Лекция 4

Абстракция данных

Функциональное и логическое программирование

КамчатГТУ, 2013 г.

Структура курса

Лекционный курс

Абстракция данных

Определение

Абстракция — выделение существенных, закономерных признаков объектов, которые отличают их от всех других предметов или явлений.

Абстракция данных

Определение

Абстракция — выделение существенных, закономерных признаков объектов, которые отличают их от всех других предметов или явлений.

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Абстракция данных

Определение

Абстракция — выделение существенных, закономерных признаков объектов, которые отличают их от всех других предметов или явлений.

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Стек

Объект S, для которого определены функции push, pop и top, удовлетворяющие следующим уравнениям

$$\mathrm{pop}(\mathrm{push}(v, S)) = S$$
$$\mathrm{top}(\mathrm{push}(v, S)) = v$$
$$\mathrm{pop}(\varnothing) = \varnothing$$
$$\mathrm{top}(\varnothing) = \bot$$

Тип данных в программировании

Тип данных определяет

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

- Оптимизация скомпилированных программ,

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

- Оптимизация скомпилированных программ,
- повышение надёжности (способ верификации),

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

- Оптимизация скомпилированных программ,
- повышение надёжности (способ верификации),
- повышение степени абстракции,

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

- Оптимизация скомпилированных программ,
- повышение надёжности (способ верификации),
- повышение степени абстракции,
- документирование кода,

Тип данных в программировании

Тип данных определяет

- множество допустимых значений,
- набор операций, которые можно применять к таким значениям,
- способ реализации хранения значений и выполнения операций.

Назначение систем типизации

- Оптимизация скомпилированных программ,
- повышение надёжности (способ верификации),
- повышение степени абстракции,
- документирование кода,
- стандартизация и согласованность интерфейсов.

Тип данных в программировании

Контроль типов и системы типизации:

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции
  (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket **и т.п.**),

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и **т.п.**),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и **т.п.**).

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.
- **слабая**— совместимость типов никак транслятором не контролируется.

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.
- **слабая**— совместимость типов никак транслятором не контролируется.

Способы определения совместимости типов:

- **номинативная** — определяется явно указанными именами типов.

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.
- **слабая**— совместимость типов никак транслятором не контролируется.

Способы определения совместимости типов:

- **номинативная** — определяется явно указанными именами типов.
- **структурная** — определяется структурой типа, а не его именем.

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.
- **слабая**— совместимость типов никак транслятором не контролируется.

Способы определения совместимости типов:

- **номинативная** — определяется явно указанными именами типов.
- **структурная** — определяется структурой типа, а не его именем.
- **аналитическая** — определяется предикатной функцией. При этом тип определён для данных, а не для переменных.

Тип данных в программировании

Контроль типов и системы типизации:

- **статическая** — контроль типов осуществляется при компиляции (C, Eiffel, Fortran, Java, Pascal, F#, Haskell, ML, OCaml, Scala, Typed Racket и т.п.),
- **динамическая** — контроль типов осуществляется во время выполнения (APL, Clojure, Erlang, JavaScript, Lisp, Perl, PHP, Prolog, Python, Ruby, Racket, Smalltalk и т.п.).
- **сильная** — совместимость типов автоматически контролируется транслятором.
- **слабая**— совместимость типов никак транслятором не контролируется.

Способы определения совместимости типов:

- **номинативная** — определяется явно указанными именами типов.
- **структурная** — определяется структурой типа, а не его именем.
- **аналитическая** — определяется предикатной функцией. При этом тип определён для данных, а не для переменных.
- **утиная** — определяется и реализуется общим интерфейсом доступа к данным типа.

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

  произвольное число однородных данных;

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

  произвольное число однородных данных;
- структуры и записи, объекты:

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

  произвольное число однородных данных;
- структуры и записи, объекты:

  фиксированное число неоднородных данных;

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

  произвольное число однородных данных;
- структуры и записи, объекты:

  фиксированное число неоднородных данных;
- функциональные и алгебраические типы данных

Средства абстракции данных

Простые типы

- числа
- логические константы
- символы
- указатели и т.п.

Составные типы

- векторы, матрицы;
- пары: комплексные, рациональные числа;
- строки;
- динамические структуры: кортежи, стеки, очереди;
- деревья, выражения и т.п.

Средства комбинирования данных

- массивы статические:

  фиксированное число однородных данных;
- массивы динамические:

  произвольное число однородных данных;
- структуры и записи, объекты:

  фиксированное число неоднородных данных;
- функциональные и алгебраические типы данных

  произвольное число неоднородных данных.

Типы данных в функциональном программировании

Программа

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Способы определения множеств

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Способы определения множеств

1. определение перечислением;

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Способы определения множеств

1. определение перечислением;
2. аналитический;

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Способы определения множеств

1. определение перечислением;
2. аналитический;
3. индуктивный (алгоритмический).

Типы данных в функциональном программировании

Программа

Функциональная программа представляет собой композицию чистых функций.

Процесс

Вычисление функционального выражения состоит в подстановке фактических аргументов вместо формальных в теле функции.

Тип данных

Понятие типа данных концептуально совпадает с понятием множества.

Определение

Функция — это отображение множества определения во множество значений.

Определение

Множество — совокупность объектов, объединённая по некоторому общему признаку.

Способы определения множеств

1. определение перечислением;
2. аналитический;
3. индуктивный (алгоритмический).
4. алгебраический;

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

1. пересечение: $A \cap B$

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

1. пересечение: $A \cap B$
2. дополнение: $A \,/\, B$

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

1. пересечение: $A \cap B$
2. дополнение: $A \setminus B$
3. объединение (сумма): $A \cup B$

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

1. пересечение: $A \cap B$
2. дополнение: $A \,/\, B$
3. объединение (сумма): $A \cup B$
4. декартово произведение: `(f` A B`)`, `(g` A B C`)`

   здесь `f` и `g` — *конструкторы типов*.

Способы определения типов данных

Определение перечислением соответствует размеченному объединению типов:

```
Five ::= 5
Bool ::= True ∪ False
Direction ::= "S" ∪ "E" ∪ "N" ∪ "W"
```

Аналитическое определение использует предикатные функции:

```
Segm ::= {x | 0 ≤ x ≤ 1}
Even ::= (divisible-by? 2)
```

Индуктивные определения используют рекурсию

```
Nat     ::= 0  ∪ 1 + Nat
Expr    ::= Atom  ∪ List
List    ::= null  ∪ Any : List
```

Алгебраические операции

1. пересечение: $A \cap B$
2. дополнение: $A \,/\, B$
3. объединение (сумма): $A \cup B$
4. декартово произведение: `(f` A B`)`, `(g` A B C`)`

   здесь `f` и `g` — *конструкторы типов*.

Определение

Типы, образуемые с помощью сумм и произведений других типов, называются *алгебраическими*.

Конструкторы абстрактных типов

Определение

Конструктор типа — это функция, которая возвращает экземпляр типа, представляющего собой произведение типов её аргументов.

Конструкторы абстрактных типов

Определение

Конструктор типа — это функция, которая возвращает экземпляр типа, представляющего собой произведение типов её аргументов.

Конструкторы типов, как правило, не производят вычислений. Это средство *композиции* и *декомпозициии* данных.

Конструкторы абстрактных типов

Определение

Конструктор типа — это функция, которая возвращает экземпляр типа, представляющего собой произведение типов её аргументов.

Конструкторы типов, как правило, не производят вычислений. Это средство *композиции* и *декомпозициии* данных.

Определение

Декомпозиция составного типа состоит в связывании его частей с заданными символами.

Построение абстрактоного стека

Функциональный стек

- Определение типа:

  `Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`
  - `top(∅) = ∅`
  - `pop(∅) = ∅`

Построение абстрактоного стека

Функциональный стек

- Определение типа:

  `Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`
  - `top(∅) = ∅`
  - `pop(∅) = ∅`

Реализация

```
Stack ::= null  ∪ (push Any Stack)

top (push v S) = v
top ∅          = ∅
```

Построение абстрактоного стека

Функциональный стек

- Определение типа:

  `Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`
  - `top(∅) = ∅`
  - `pop(∅) = ∅`

Реализация

```
Stack ::= null  ∪ (push Any Stack)

top (push v S) = v
top ∅          = ∅

pop (push v S) = S
pop ∅          = ∅
```

Определение рациональных чисел

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Определение рациональных чисел

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Реализация

```
Rat ::= rat Int Nat

make-rat n d =  let x = (gcd n d) in (rat n/x d/x)
```

Определение рациональных чисел

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Реализация

```
Rat ::= rat Int Nat

make-rat n d =  let x = (gcd n d) in (rat n/x d/x)

add (rat a b) (rat c d) = make-rat (a*d + b*c) b*d
```

Определение рациональных чисел

Рациональное число

Упорядоченная пара, $(a, b), a \in Z, b \in N$, для которой выполняется уравнение $\gcd(a, b) = 1$, а также определены операции сложения, умножения и деления:

$$(a, b) + (a', b') = (ab' + a'b, bb')$$
$$(a, b) \times (a', b') = (aa', bb')$$
$$(a, b)/(a', b') = (ab', ba')$$

Реализация

```
Rat ::= rat Int Nat

make-rat n d =  let x = (gcd n d) in (rat n/x d/x)

add (rat a b) (rat c d) = make-rat (a*d + b*c) b*d

mul (rat a b) (rat c d) = make-rat a*c  b*d
```

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S  :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- **Базовые операции:**
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- **свойства:**
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`

Функциональные типы данных

Определение

Объект принадлежит к **функциональному** (неизменяемому) типу данных, если не допускается прямое изменение уже существующего объекта. Все изменения функциональных данных производятся с помощью функций, принимающих старый объект и возвращающих новый — с изменёнными свойствами.

Изменяемый стек

- Базовые операции:
  - `push x S :: Any Stack → void`
  - `pop S :: Stack → Any`
  - `create :: void → Stack`
- свойства:
  - `(push S x); y := (pop S)` ⇔ `y:=x`

Функциональный стек

`Stack ::= ∅ | push Any Stack`

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
```

Состояние:

S: $\varnothing$

Эквивалентная функциональная реализация:

```
S = []
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
```

Состояние:

S: $\langle a \rangle$

Эквивалентная функциональная реализация:

```
S = push a []
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
```

Состояние:

S: $\langle b\ a \rangle$

Эквивалентная функциональная реализация:

```
S = push b (push a [])
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
push c S;
```

Состояние:

S: $\langle c\ b\ a \rangle$

Эквивалентная функциональная реализация:

```
S = push c (push b (push a []))
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
push c S;
x := pop S;
```

Состояние:

S: $\langle b\ a \rangle$
x: $\langle a \rangle$

Эквивалентная функциональная реализация:

```
S = pop (push c (push b (push a [])))
x = top (push c (push b (push a [])))
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
push c S;
x := pop S;
T := create
```

Состояние:

S: $\langle b\ a \rangle$
x: $\langle a \rangle$
T: $\varnothing$

Эквивалентная функциональная реализация:

```
S = pop (push c (push b (push a [])))
x = top (push c (push b (push a [])))
T = []
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
push c S;
x := pop S;
T := create
push 1 T;
```

Состояние:

S: $\langle b\ a \rangle$
x: $\langle a \rangle$
T: $\langle 1 \rangle$

Эквивалентная функциональная реализация:

```
S = pop (push c (push b (push a [])))
x = top (push c (push b (push a [])))
T = push 1 []
```

Функциональные типы данных

Операции с изменяемым стеком:

```
S := create;
push a S;
push b S;
push c S;
x := pop S;
T := create
push 1 T;
y := pop T;
```

Состояние:

S: $\langle b\ a \rangle$
x: $\langle a \rangle$
T: $\langle\rangle$
y: 1

Эквивалентная функциональная реализация:

```
S = pop (push c (push b (push a [])))
x = top (push c (push b (push a [])))
T = pop (push 1 [])
y = top (push 1 T)
```

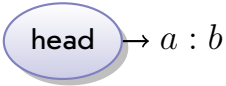
Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).

$$a : b$$

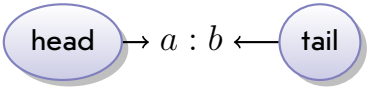
Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



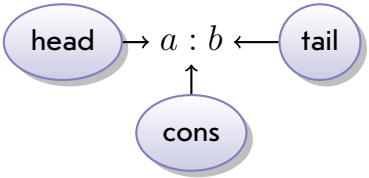
Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



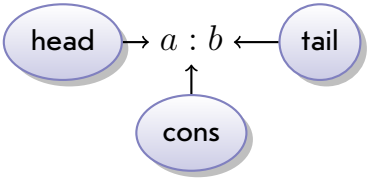
Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



Точечная пара

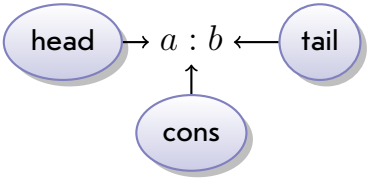
Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



$cons$ — **конструктор** пары
$head$, $tail$ — **селекторы** пары

Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).

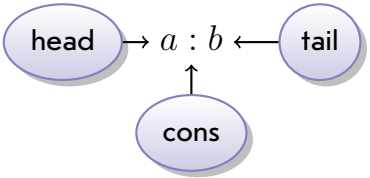


$cons$ — **конструктор** пары
$head$, $tail$ — **селекторы** пары

$$head\ (a : b) = a,$$
$$tail\ (a : b) = b,$$
$$(head\ p) : (tail\ p) = p.$$

Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



$cons$ — **конструктор** пары
$head$, $tail$ — **селекторы** пары

$$head\ (a : b) = a,$$
$$tail\ (a : b) = b,$$
$$(head\ p) : (tail\ p) = p.$$

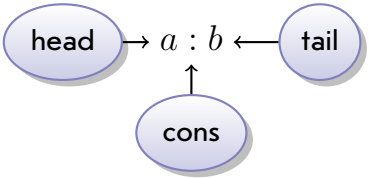
Свойства точечной пары:

- несимметрична:

$$a : b \neq b : a, \text{ если } a \neq b;$$

Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



$cons$ — **конструктор** пары
$head$, $tail$ — **селекторы** пары

$$head\ (a : b) = a,$$
$$tail\ (a : b) = b,$$
$$(head\ p) : (tail\ p) = p.$$

Свойства точечной пары:

- несимметрична:

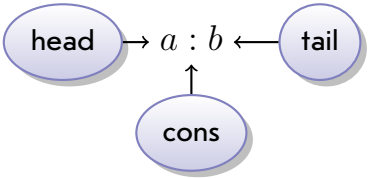
$$a : b \neq b : a, \text{ если } a \neq b;$$

- левоассоциативна:

$$a : b : c = a : (b : c) \neq (a : b) : c;$$

Точечная пара

Элементарным средством динамического комбинирования разнородных данных является **точечная пара** (cons-ячейка, cons-пара).



$cons$ — **конструктор** пары
$head$, $tail$ — **селекторы** пары

$$head\ (a : b) = a,$$
$$tail\ (a : b) = b,$$
$$(head\ p) : (tail\ p) = p.$$

Свойства точечной пары:

- несимметрична:

$$a : b \neq b : a, \text{ если } a \neq b;$$

- левоассоциативна:

$$a : b : c = a : (b : c) \neq (a : b) : c;$$

- правило эквивалентности:

$$a : b = a' : b' \ \Leftrightarrow \ a = a' \wedge b = b'$$

Комбинирование данных с помощью cons-пар

С помощью точечных пар мы можем создавать различные структуры данных, например:

Комбинирование данных с помощью cons-пар

С помощью точечных пар мы можем создавать различные структуры данных, например:

- линейные списочные структуры:

$$a : b : c : d = a : (b : (c : d))$$

Комбинирование данных с помощью cons-пар

С помощью точечных пар мы можем создавать различные структуры данных, например:

- линейные списочные структуры:

$$a : b : c : d = a : (b : (c : d))$$

- списки ключ-значение:

$$(k_1 : v_1) : (k_2 : v_2) : (k_3 : v_3) : ...$$

Комбинирование данных с помощью cons-пар

С помощью точечных пар мы можем создавать различные структуры данных, например:

- линейные списочные структуры:

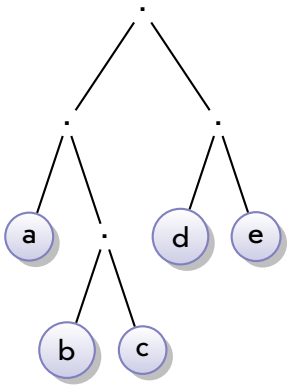
$$a : b : c : d = a : (b : (c : d))$$

- списки ключ-значение:

$$(k_1 : v_1) : (k_2 : v_2) : (k_3 : v_3) : \ldots$$

- двоичные деревья:

$$((a : (b : c)) : (d : e))$$



Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

`S = []`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

```
S = push a [] = a:[]
```

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

```
S = push b (push a []) = b:a:[]
```

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

```
S = push c (push b (push a [])) = c:b:a:[]
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

```
S = push c (push b (push a [])) = c:b:a:[]
top(S) = c
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

```
S = push c (push b (push a [])) = c:b:a:[]
top(S) = c
pop(S) = b:a:[]
```

Построение стека с помощью точечной пары

Функциональный стек

- Базовые операции:
  - `push :: Any Stack → Stack`
  - `top :: Stack → Any`
  - `pop :: Stack → Stack`
- свойства:
  - `top (push x S) = x`
  - `pop (push x S) = S`

Свойства функций для обработки стека совпадают со свойствами конструктора и селекторов точечной пары:

Точечная пара

- свойства:
  - `head (cons x y) = x`
  - `tail (cons x z) = z`

Следовательно, возможна такая реализация:

```
push = cons  top = head  pop = tail
```

```
S = push c (push b (push a [])) = c:b:a:[]
top(S) = c
pop(S) = b:a:[]
pop (pop (pop S)) = []
```

Списки

Определение

Списками называются:

Списки

Определение

Списками называются:
- символ [] **(пустой список)**,

Списки

Определение

Списками называются:
- символ [] **(пустой список)**,
- либо линейная списочная структура, самым левым элементом которой является [].

Списки

Определение

Списками называются:
- символ $[\,]$ **(пустой список)**,
- либо линейная списочная структура, самым левым элементом которой является $[\,]$.

списочные структуры:

$$a : b$$
$$a : b : c : d$$

Списки

Определение

Списками называются:
- символ $[\,]$ **(пустой список)**,
- либо линейная списочная структура, самым левым элементом которой является $[\,]$.

списочные структуры:

$$a : b$$
$$a : b : c : d$$

Списочная структура

`ListStr ::= Atom:Atom ∪ Atom:ListStr`

Списки

Определение

Списками называются:
- символ $[\,]$ **(пустой список)**,
- либо линейная списочная структура, самым левым элементом которой является $[\,]$.

списочные структуры:

$$a : b$$
$$a : b : c : d$$

списки:

$$null \qquad [\,]$$
$$a : null \qquad [a]$$
$$a : b : c : null \qquad [a\ b\ c]$$

Списочная структура

`ListStr ::= Atom:Atom ∪ Atom:ListStr`

Списки

Определение

Списками называются:
- символ $[\,]$ **(пустой список)**,
- либо линейная списочная структура, самым левым элементом которой является $[\,]$.

списочные структуры:

$$a : b$$
$$a : b : c : d$$

списки:

$null$	$[\,]$
$a : null$	$[a]$
$a : b : c : null$	$[a\ b\ c]$

Списочная структура

`ListStr ::= Atom:Atom ∪ Atom:ListStr`

Список

`List ::= [] ∪ Atom:List`

Использование списков

С помощью списков можно также конструировать различные составные объекты:

- векторы и массивы

Использование списков

С помощью списков можно также конструировать различные составные объекты:

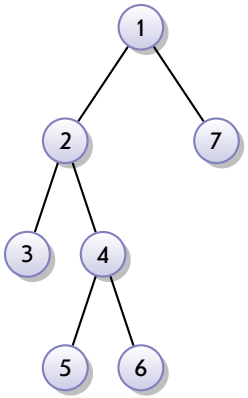
- векторы и массивы
- функциональные типы: очереди, стеки и т.п.

Использование списков

С помощью списков можно также конструировать различные составные объекты:

- векторы и массивы
- функциональные типы: очереди, стеки и т.п.
- размеченные деревья:

[1 [2 [3 [] []] [4 [5 [] []] [6 [] []]]] [7 [] []]]



Использование списков

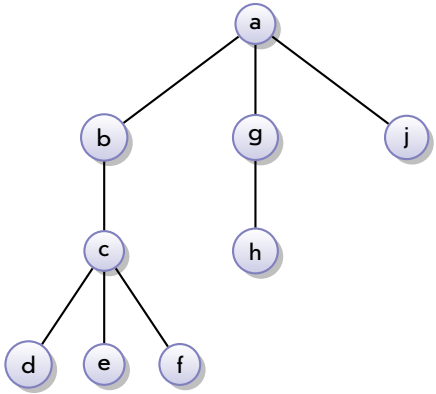
С помощью списков можно также конструировать различные составные объекты:

- векторы и массивы
- функциональные типы: очереди, стеки и т.п.
- размеченные деревья:

$$[1\ [2\ [3\ [\,]\ [\,]]\ [4\ [5\ [\,]\ [\,]]\ [6\ [\,]\ [\,]]]]\ [7\ [\,]\ [\,]]]$$

- произвольные деревья (S-выражения):

$$[a\ [b\ [c\ d\ e\ f]]\ [g\ h]\ j]$$



Использование списков

С помощью списков можно также конструировать различные составные объекты:

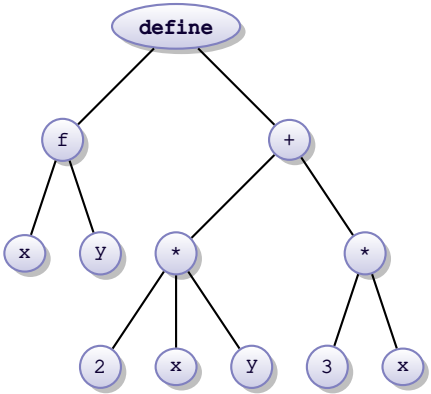
- векторы и массивы
- функциональные типы: очереди, стеки и т.п.
- размеченные деревья:

$$[1\ [2\ [3\ [\,]\ [\,]]\ [4\ [5\ [\,]\ [\,]]\ [6\ [\,]\ [\,]]]]\ [7\ [\,]\ [\,]]]$$

- произвольные деревья (S-выражения):

$$[a\ [b\ [c\ d\ e\ f]]\ [g\ h]\ j]$$

```
(define (f x y) (+ (* 2 x y) (* 3 x)))
```



Базовые функции для работы со списками

- $list$ — конструктор

$$list\ a\ b\ c\ = [a\ b\ c]$$
$$(list) = [\,]$$

Базовые функции для работы со списками

- $list$ — **конструктор**

$$list\ a\ b\ c\ = [a\ b\ c]$$
$$(list) = [\,]$$

- $first$, $second$, ..., $rest$, ref — **селекторы**

$$first\ [a\ b\ c] = a$$
$$second\ [a\ b\ c] = b$$
$$rest\ [a\ b\ c] = [b\ c]$$
$$ref\ 3\ [a\ b\ c] = c$$
$$first = head, \quad rest = tail$$

Базовые функции для работы со списками

- $list$ — конструктор

$$list\ a\ b\ c\ = [a\ b\ c]$$
$$(list) = [\,]$$

- $first$, $second$, ..., $rest$, ref — селекторы

$$first\ [a\ b\ c] = a$$
$$second\ [a\ b\ c] = b$$
$$rest\ [a\ b\ c] = [b\ c]$$
$$ref\ 3\ [a\ b\ c] = c$$
$$first = head, \quad rest = tail$$

- $list?$, $empty?$ — квантификаторы

$$list?\ [a\ b\ c] = true \qquad empty?\ [a\ b\ c] = false$$
$$list?\ a : b = false \qquad empty?\ [\,] = true$$
$$list?\ [\,] = true \qquad empty? = null?$$

Базовые функции для работы со списками

- $list$ — конструктор

$$list\ a\ b\ c\ = [a\ b\ c]$$
$$(list) = [\,]$$

- $first$, $second$, ..., $rest$, ref — селекторы

$$first\ [a\ b\ c] = a$$
$$second\ [a\ b\ c] = b$$
$$rest\ [a\ b\ c] = [b\ c]$$
$$ref\ 3\ [a\ b\ c] = c$$
$$first = head, \quad rest = tail$$

- $list?$, $empty?$ — квантификаторы

$$list?\ [a\ b\ c] = true \qquad empty?\ [a\ b\ c] = false$$
$$list?\ a : b = false \qquad empty?\ [\,] = true$$
$$list?\ [\,] = true \qquad empty? = null?$$

- $length$ — длина списка

$$length\ [a\ b\ c] = 3$$
$$length\ [\,] = 0$$

Базовые функции для работы со списками

- $list$ — конструктор

$$list\ a\ b\ c\ = [a\ b\ c]$$
$$(list) = [\,]$$

- $first$, $second$, ..., $rest$, ref — селекторы

$$first\ [a\ b\ c] = a$$
$$second\ [a\ b\ c] = b$$
$$rest\ [a\ b\ c] = [b\ c]$$
$$ref\ 3\ [a\ b\ c] = c$$
$$first = head, \quad rest = tail$$

- $list?$, $empty?$ — квантификаторы

$$list?\ [a\ b\ c] = true \quad empty?\ [a\ b\ c] = false$$
$$list?\ a : b = false \qquad empty?\ [\,] = true$$
$$list?\ [\,] = true \qquad empty? = null?$$

- $length$ — длина списка

$$length\ [a\ b\ c] = 3$$
$$length\ [\,] = 0$$

- $append$ — конкатенация двух или более списков

$$append\ [a\ b\ c]\ [d\ e] = [a\ b\ c\ d\ e]$$

Что ещё можно делать со списками?

- Анализ содержимого

$$find\ odd?\ [1\ 2\ 3\ 4] = 2$$
$$member\ 3\ [1\ 2\ 3\ 4] = [3\ 4]$$
$$every?\ number?\ [1\ 2\ 3\ 4] = true$$
$$count\ x\ [1\ x\ 3\ x\ x\ 5] = 3$$
$$max\ [1\ 3\ 5\ 2\ 4] = 5$$

Что ещё можно делать со списками?

- **Анализ содержимого**

$$find\ odd?\ [1\ 2\ 3\ 4] = 2$$
$$member\ 3\ [1\ 2\ 3\ 4] = [3\ 4]$$
$$every?\ number?\ [1\ 2\ 3\ 4] = true$$
$$count\ x\ [1\ x\ 3\ x\ x\ 5] = 3$$
$$max\ [1\ 3\ 5\ 2\ 4] = 5$$

- **Отображения, свёртки и обобщённые произведения**

$$map\ (x \mapsto x^2)\ [a\ b\ c] = [a^2\ b^2\ c^2]$$
$$map\ +\ [2\ 3\ 4]\ [1\ 2\ 3] = [3\ 5\ 7]$$
$$sum\ [1\ 2\ 3\ 4] = 10$$
$$product\ [1\ 2\ 3\ 4] = 24$$
$$inner\ +\ \times\ [a\ b\ c]\ [x^2\ x\ 1] = ax^2 + bx + c$$

Что ещё можно делать со списками?

- **Анализ содержимого**

$$find\ odd?\ [1\ 2\ 3\ 4] = 2$$
$$member\ 3\ [1\ 2\ 3\ 4] = [3\ 4]$$
$$every?\ number?\ [1\ 2\ 3\ 4] = true$$
$$count\ x\ [1\ x\ 3\ x\ x\ 5] = 3$$
$$max\ [1\ 3\ 5\ 2\ 4] = 5$$

- **Отображения, свёртки и обобщённые произведения**

$$map\ (x \mapsto x^2)\ [a\ b\ c] = [a^2\ b^2\ c^2]$$
$$map\ +\ [2\ 3\ 4]\ [1\ 2\ 3] = [3\ 5\ 7]$$
$$sum\ [1\ 2\ 3\ 4] = 10$$
$$product\ [1\ 2\ 3\ 4] = 24$$
$$inner\ +\ \times\ [a\ b\ c]\ [x^2\ x\ 1] = ax^2 + bx + c$$

- **Фильтрация**

$$filter\ odd?\ [1\ 2\ 3\ 4] = [2\ 4]$$
$$remove\ odd?\ [1\ 2\ 3\ 4] = [1\ 3]$$

Что ещё можно делать со списками?

- **Анализ содержимого**

$$find\ odd?\ [1\ 2\ 3\ 4] = 2$$
$$member\ 3\ [1\ 2\ 3\ 4] = [3\ 4]$$
$$every?\ number?\ [1\ 2\ 3\ 4] = true$$
$$count\ x\ [1\ x\ 3\ x\ x\ 5] = 3$$
$$max\ [1\ 3\ 5\ 2\ 4] = 5$$

- **Отображения, свёртки и обобщённые произведения**

$$map\ (x \mapsto x^2)\ [a\ b\ c] = [a^2\ b^2\ c^2]$$
$$map\ +\ [2\ 3\ 4]\ [1\ 2\ 3] = [3\ 5\ 7]$$
$$sum\ [1\ 2\ 3\ 4] = 10$$
$$product\ [1\ 2\ 3\ 4] = 24$$
$$inner\ +\ \times\ [a\ b\ c]\ [x^2\ x\ 1] = ax^2 + bx + c$$

- **Фильтрация**

$$filter\ odd?\ [1\ 2\ 3\ 4] = [2\ 4]$$
$$remove\ odd?\ [1\ 2\ 3\ 4] = [1\ 3]$$

- **Структурные преобразования**

$$sort\ [2\ 1\ 5\ 4] = [1\ 2\ 4\ 5]$$
$$partition\ 2\ [1\ 2\ 3\ 4\ 5\ 6] = [[1\ 2]\ [3\ 4]\ [5\ 6]]$$
$$transpose\ [[1\ 2]\ [3\ 4]\ [5\ 6]] = [[1\ 3\ 5]\ [2\ 4\ 6]]$$
$$flatten\ [[1\ 2\ []]\ 3\ [4\ [5\ 6]]] = [1\ 2\ 3\ 4\ 5\ 6]$$
$$union\ [1\ 2\ 3\ 2\ 3\ 2] = [1\ 2\ 3]$$
$$split\ [a\ a\ a\ b\ b\ a\ c] = [[a\ a\ a]\ [b\ b]\ [a]\ [c]]$$
$$insert\ x\ [1\ 2\ 3\ 4] = [1\ x\ 2\ x\ 3\ x\ 4]$$
$$rotate\ [1\ 2\ 3\ 4] = [4\ 1\ 2\ 3]$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: **0**
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: **0**
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: **0**
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

3

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$[a\ b\ c]$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: **0**
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: **0**
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 2$$
$$\downarrow$$
$$(g\ 3$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \longrightarrow [b\ c]$$
$$\downarrow$$
$$(g\ a$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 2$$
$$\downarrow$$
$$(g\ 3\ (g\ 2$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \rightarrow [b\ c]$$
$$\downarrow$$
$$(g\ a\ (g\ b$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 1 + 1 \longrightarrow 1$$
$$\downarrow$$
$$(g\ 3\ (g\ 2$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \to b : [c] \longrightarrow [c]$$
$$\downarrow$$
$$(g\ a\ (g\ b$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 1 + 1 \longrightarrow 1$$
$$\downarrow$$
$$(g\ 3\ (g\ 2\ (g\ 1$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \to b : [c] \longrightarrow [c]$$
$$\downarrow$$
$$(g\ a\ (g\ b\ (g\ c$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 1 + 1 \longrightarrow 1 + 0 \longrightarrow 0$$
$$\downarrow$$
$$(g\ 3\ (g\ 2\ (g\ 1$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \to b : [c] \longrightarrow c : [\,] \longrightarrow [\,]$$
$$\downarrow$$
$$(g\ a\ (g\ b\ (g\ c$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 1 + 1 \longrightarrow 1 + 0 \longrightarrow 0 \downarrow$$
$$(g\ 3\ (g\ 2\ (g\ 1\ x_0)))$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \to b : [c] \longrightarrow c : [\,] \longrightarrow [\,] \downarrow$$
$$(g\ a\ (g\ b\ (g\ c\ x_0)))$$

Обработка списков

Обработка натурального числа.

- определение: `N ::= 0 | (1 + N)`
- база: 0
- конструктор: $succ = (x \mapsto 1 + x)$
- деструктор: $pred = (1 + x \mapsto x)$

$$F\ 0 = x_0$$
$$F\ (1 + n) = g\ (1 + n)\ (F\ n)$$

$$1 + 2 \longrightarrow 1 + 1 \longrightarrow 1 + 0 \longrightarrow 0 \downarrow$$
$$(g\ 3\ (g\ 2\ (g\ 1\ x_0)))$$

Поэлементная обработка списка.

- определение: `List ::= [] | Atom : List`
- база: $[\,]$
- конструктор: $succ = l \mapsto (A : l)$
- деструктор: $pred = (A : t) \mapsto t$

$$F\ [\,] = x_0$$
$$F\ h : t = g\ h\ (F\ t)$$

$$a : [b\ c] \to b : [c] \longrightarrow c : [\,] \longrightarrow [\,] \downarrow$$
$$(g\ a\ (g\ b\ (g\ c\ x_0)))$$

Примеры обработки списков

Длина списка

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

Примеры обработки списков

Длина списка

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

```
length [a b c]
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
```

```
1 + length [b c]
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

```
insert x [a b c]
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

```
insert x [a b c]
a : x : (insert x [b c])
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

```
insert x [a b c]
a : x : (insert x [b c])
a : x : b : x : (insert x [c])
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

```
insert x [a b c]
a : x : (insert x [b c])
a : x : b : x : (insert x [c])
a : x : b : x : [c])
```

Примеры обработки списков

Длина списка

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

```
length [a b c]
1 + length [b c]
1 + 1 + length [c]
1 + 1 + 1 + length []
1 + 1 + 1 + 0
```

insert

```
insert _ [x] = [x]
insert x (h : t) = h : x : (insert x t)
```

```
insert x [a b c]
a : x : (insert x [b c])
a : x : b : x : (insert x [c])
a : x : b : x : [c])
[a x b x c]
```

Сравнение массивов и списков по эффективности

Операция	Статический массив	Динамический массив		Список
Поэлементная обработка	$\Theta(n)$	$\Theta(n)$	$=$	$\Theta(n)$
Получение значения i-того элемента	$\Theta(1)$	$\Theta(1)$	$<$	$\Theta(i)$
Изменение значения i-того элемента	$\Theta(1)$	$\Theta(1)$		—
Получение значения длины массива	$\Theta(1)$	$\Theta(1)$	$<$	$\Theta(n)$
Вставка/удаление первого элемента	—	$\Theta(n)$	$>$	$\Theta(1)$
Вставка/удаление i-того элемента	—	$\Theta(n-i)$	$\sim$	$\Theta(i)$
Перерасход памяти	0	$\Theta(n)$	$\lesssim$	$\Theta(n)$

Использование массивов и списков

Использование массивов и списков

Использование массивов и списков

Использование массивов

Использование массивов и списков

Использование массивов

- хранение индексированных данных

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:
    - синтаксические деревья,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:
    - синтаксические деревья,
    - деревья вычислений,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:
    - синтаксические деревья,
    - деревья вычислений,
    - XML,

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:
    - синтаксические деревья,
    - деревья вычислений,
    - XML,
    - словари;

Использование массивов и списков

Использование массивов

- хранение индексированных данных
  - векторы и матрицы;
  - строки;
  - таблицы:
    - прямоугольные таблицы произвольной размерности,
    - таблицы с фиксированными структурными записями,
    - ассоциативные таблицы;
  - последовательности;
  - коллекции с произвольным доступом.

Использование списков

- хранение структурированных данных:
  - последовательности и коллекции;
  - упорядоченные множества:
    - ассоциативные таблицы;
    - списки с пропусками,
    - кучи (heaps),
    - словари;
  - иерархические структуры:
    - синтаксические деревья,
    - деревья вычислений,
    - XML,
    - словари;
  - построение функциональных типов данных.

Списки в Scheme

Lisp – **L**ots of **i**dioticly **s**miling **p**arenthesis

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,

```
> (cons 1 2)
(1 . 2)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,

```
> (cons 1 (cons 2 3))
(1 2 . 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,

```
> ((cons 1 2) (cons 3 4))
((1 . 2) . (3 . 4))
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,

```
> (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,

```
> (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,

```
> (car (cons 1 2))
1
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,

```
> (car (cons (cons 1 2) 3))
(1 . 2)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.

```
> (car (cons (cons 1 2) 3))
(1 . 2)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.

```
> (cdr (cons 1 2))
2
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.

```
> (cdr (cons 1 (cons 2 3)))
(2 . 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.

```
> (car (cdr (cons 1 (cons 2 3))))
2
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (cons 1 (cons 2 3)))
#t
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`

```
> (list 1 2 3)
'(1 2 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`

```
> '(1 2 3)
'(1 2 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`

```
> '(1 2 3)
'(1 2 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`

```
> (car '(1 2 3))
1
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`

```
> (cdr '(1 2 3))
'(2 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`

```
> (cdr '(1 2 3))
'(2 3)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`

```
> null
'()
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`

```
> (cons 2 null)
'(2)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`
- квантификатор списка: `list?`

```
> (cons 2 null)
'(2)
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`
- квантификатор списка: `list?`

```
> (list? '(cons 2 null))
#t
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`
- квантификатор списка: `list?`

```
> (list? '(cons 2 3))
#f
```

Списки в Scheme

Lisp – **Lis**t **P**rocessing language — «язык обработки списков»

- списки представляют в нём **данные**,
- и списки представляют **программы**;
- в этом языке нет ничего, кроме символов, списков и составляющих их точечных пар.

Синтаксис использования пар

- конструктор: `cons`,
- селектор правой части: `car`,
- селектор левой части: `cdr`.
- квантификатор пары: `pair?`

```
> (pair? (car (cons 1 (cons 2 3))))
#f
```

Синтаксис использования списков

- конструктор: `list`, `'`
- селектор головы: `car`, `first`
- селектор хвоста: `cdr`, `rest`
- пустой список: `null`, `empty`, `'()`
- квантификатор списка: `list?`
- квантификатор пустого списка: `empty?`, `null?`

```
> (list? '(cons 2 3))
#f
```

	pair?	list?	null?
`(a . b)`	+		
`'(a b)`	+	+	
`null`		+	+

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,
  - аггрегирование (свёртку).

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,
  - аггрегирование (свёртку).
- Запросы функциональны, они не меняют переданные им данные, а возвращают новые.

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,
  - аггрегирование (свёртку).
- Запросы функциональны, они не меняют переданные им данные, а возвращают новые.
- Реализует функциональную композицию запросов

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,
  - аггрегирование (свёртку).
- Запросы функциональны, они не меняют переданные им данные, а возвращают новые.
- Реализует функциональную композицию запросов

Списки в C#

LINQ (Language Integrated Quiery) — набор функциональных возможностей языка C#.

- Работает с объектами, реализующими интерфейс `IEnumerable<T>`.
- Предоставляет около 40 методов обработки коллекций (запросов):
  - фильтрацию,
  - проецирование (отображения),
  - структурные преобразования,
  - аггрегирование (свёртку).
- Запросы функциональны, они не меняют переданные им данные, а возвращают новые.
- Реализует функциональную композицию запросов

Пример:

```
using System;
using System.Linq;
class LinqDemo
{
  static void Main ()
  {
    string[] names =
      { "Tom", "Harry", "Dick", "Mary" };
    IEnumerable<string> query = names
      .Where     (n => n.EndsWith("y"))
      .OrderedBy (n => n.Length)
      .Select    (n => n.ToUpper( ));
    foreach (string name in query)
      Console.Write(name+" ");
  }}
```

Результат: `MARY HARRY`

Абстракция данных

Преимущества использования АТД

Абстракция данных

Преимущества использования АТД

- Выразительность:

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.
- Надёжность

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.
- Надёжность
  - наличие доказываемых свойств и инвариантов.

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.
- Надёжность
  - наличие доказываемых свойств и инвариантов.
  - построение денотационной семантики для данных и операций над ними,

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.
- Надёжность
  - наличие доказываемых свойств и инвариантов.
  - построение денотационной семантики для данных и операций над ними,

Определение

Денотационная семантика объектам в программе ставит в соответствие настоящие математические объекты.

Абстракция данных

Преимущества использования АТД

- Выразительность:
  - высокая предметная ориентированность,
  - простота аспектной ориентированности.
- Модульность:
  - логическая замкнутость,
  - выстраивание барьера абстракции.
- Надёжность
  - наличие доказываемых свойств и инвариантов.
  - построение денотационной семантики для данных и операций над ними,

Определение

Денотационная семантика объектам в программе ставит в соответствие настоящие математические объекты.

Определение

Операционная семантика описывает каким образом синтаксически верная программа интепретируется в виде последовательности вычислений.

Заключение

Рассмотренная система типов позволяет

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.
- Обеспечивают замкнутость операции комбинирования.

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.
- Обеспечивают замкнутость операции комбинирования.

Списки позволяют:

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.
- Обеспечивают замкнутость операции комбинирования.

Списки позволяют:

- создавать произвольные динамические кортежи;

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Списки позволяют:

- создавать произвольные динамические кортежи;
- оперировать кортежем, как единым объектом;

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.
- Обеспечивают замкнутость операции комбинирования.

Заключение

Рассмотренная система типов позволяет

- осуществлять абстракцию данных, пользуясь математическими приёмами и инструментами;
- создавать функциональные типы данных для исключения побочных эффектов;
- определять расширяемые алгебраические и индуктивные типы данных.

Точеченые пары позволяют

- комбинировать данные произвольных типов;
- конструировать произвольные динамически расширяемые структуры данных.
- Обеспечивают замкнутость операции комбинирования.

Списки позволяют:

- создавать произвольные динамические кортежи;
- оперировать кортежем, как единым объектом;
- строить выражения произвольной структуры.