Лекция 1

Введение

Функциональное и логическое программирование

КамчатГТУ, 2013 г.

Вавилон программирования

Число языков, на которых говорят люди: 2500 – 7000

Вавилон программирования

Число языков, на которых говорят люди:	2500 – 7000
Число языков программирования:	2000 – 8500

Вавилон программирования

Число языков, на которых говорят люди:	2500 – 7000
Число языков программирования:	2000 – 8500
Число парадигм программирования	$\sim$ 20

Вавилон программирования

Число языков, на которых говорят люди:	2500 – 7000
Число языков программирования:	2000 – 8500
Число парадигм программирования	$\sim$ 20
Число возможных языков программирования	$\sim$ 1 000 000

Программы, вычисляющие факториал

МК-152

```
П0 1 ИП0 x FL0 02 С/П БП 00
```

Программы, вычисляющие факториал

МК-152

```
П0 1 ИП0 x FL0 02 С/П БП 00
```

Assembler

```
global factorial
section .text

; Input in ECX register
; (greater than 0!)
; Output in EAX register
factorial:
  mov   eax, 1
.L1:
  mul   ecx
  loop  .L1
  ret
```

Программы, вычисляющие факториал

МК-152

```
П0 1 ИП0 x FL0 02 С/П БП 00
```

Assembler

```
global factorial
section .text

; Input in ECX register
; (greater than 0!)
; Output in EAX register
factorial:
  mov   eax, 1
.L1:
  mul   ecx
  loop  .L1
  ret
```

Pascal

```
function fact(n : integer) : longint;
  begin
    if n <= 1 then
      fact := 1
    else
      fact := n * fact(n - 1);
  end;
```

Программы, вычисляющие факториал

МК-152

```
П0 1 ИП0 x FL0 02 С/П БП 00
```

Assembler

```
global factorial
section .text

; Input in ECX register
; (greater than 0!)
; Output in EAX register
factorial:
  mov   eax, 1
.L1:
  mul   ecx
  loop  .L1
  ret
```

Pascal

```
function fact(n : integer) : longint;
  begin
    if n <= 1 then
      fact := 1
    else
      fact := n * fact(n - 1);
  end;
```

Scheme

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

Программы, вычисляющие факториал

APL

```
Factorial ← {α ← 1
    ω = 0 : α
    (ω − α × ω)∇1}
```

Программы, вычисляющие факториал

APL

```
Factorial ← {α ← 1
    ω = 0 : α
    (ω − α × ω)∇1}
```

Haskell

```
factorial = product . naturals

product = foldl (*) 1

naturals 1 = 1
naturals n =  n : naturals (n − 1)
```

Программы, вычисляющие факториал

APL

```
Factorial ← {α ← 1
    ω = 0 : α
    (ω − α × ω)∇1}
```

Joy

```
DEFINE fac == [1] [*] primrec.
```

Haskell

```
factorial = product . naturals

product = foldl (*) 1

naturals 1 = 1
naturals n =  n : naturals (n − 1)
```

Программы, вычисляющие факториал

APL

```
Factorial ← {α ← 1
     ω = 0 : α
     (ω − α × ω)∇1}
```

Haskell

```
factorial = product . naturals

product = foldl (*) 1

naturals 1 = 1
naturals n =  n : naturals (n − 1)
```

Joy

```
DEFINE fac == [1] [*] primrec.
```

Formica

```
(define fact
  (//. n --> (values n 1)
       0 res -->. res;
       n res --> (values (- n 1) (* n res))))
```

Программы, вычисляющие факториал

APL

```
Factorial ← {α ← 1
     ω = 0 : α
     (ω − α × ω)∇1}
```

Haskell

```
factorial = product . naturals

product = foldl (*) 1

naturals 1 = 1
naturals n =  n : naturals (n − 1)
```

Joy

```
DEFINE fac == [1] [*] primrec.
```

Formica

```
(define fact
  (//. n --> (values n 1)
       0 res -->. res;
       n res --> (values (- n 1) (* n res))))
```

Prolog

```
fact(0,1).
fact(N, F) :- fact(N − 1, X), F = N ∗ X.
```

Программы, вычисляющие факториал

Cobol

```
main-line.
  accept n
  move 1 to result.
  perform varying i from 1 by 1 until i>n
    multiply result by i giving result
  end-perform
  display "Factorial(" n ")= " result
  stop run.
```

Программы, вычисляющие факториал

Cobol

```
main-line.
  accept n
  move 1 to result.
  perform varying i from 1 by 1 until i>n
    multiply result by i giving result
  end-perform
  display "Factorial(" n ")= " result
  stop run.
```

Befunge

```
&1\>  :v v *<
   ^-1:_$>\:|
         @.$<
```

Программы, вычисляющие факториал

Cobol

```
main-line.
  accept n
  move 1 to result.
  perform varying i from 1 by 1 until i>n
    multiply result by i giving result
  end-perform
  display "Factorial(" n ")= " result
  stop run.
```

Brainf**k

```
>++++++++++>>>+>+[>>>+[-[<<<<<[+<<<<<]>>
[[-]>[<<+>+>-]<[>+<-]<[>+<-[>+<-[>+<-[>+
<-[>+<-[>+<-[>+<-[>+<-[>+<-[>[-]>>>>+>+<
<<<<<-[>+<-]]]]]]]]]]>[<+>-]+>>>>>]<<<<
<[<<<<<]>>>>>>>[>>>>>]++[-<<<<<]>>>>>>-]
+>>>>>]<[>++<-]<<<<[<[>+<-]<<<<]>>[>[-]+
+++++[<++++++++>-]>>>>]<<<<<[<[>+>+<<-]>
.<<<<<]>.>>>>]
```

Befunge

```
&1\>  :v v *<
   ^-1:_$>\:|
         @.$<
```

Программы, вычисляющие факториал

Cobol

```
main-line.
  accept n
  move 1 to result.
  perform varying i from 1 by 1 until i>n
    multiply result by i giving result
  end-perform
  display "Factorial(" n ")= " result
  stop run.
```

Brainf**k

```
>++++++++++>>>+>+[>>>+[-[<<<<<[+<<<<<]>>
[[-]>[<<+>+>-]<[>+<-]<[>+<-[>+<-[>+<-[>+
<-[>+<-[>+<-[>+<-[>+<-[>+<-[>[-]>>>>+>+<
<<<<<-[>+<-]]]]]]]]]]>[<+>-]+>>>>>]<<<<
<[<<<<<]>>>>>>>[>>>>>]++[-<<<<<]>>>>>>-]
+>>>>>]<[>++<-]<<<<[<[>+<-]<<<<]>>[>[-]+
+++++[<++++++++>-]>>>>]<<<<<[<[>+>+<<-]>
.<<<<<]>.>>>>]
```

Befunge

```
&1\>  :v v *<
   ^-1:_$>\:|
         @.$<
```

Piet

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов
продукционная	правило

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов
продукционная	правило

Различные парадигмы позволяют оптимизировать различные аспекты:

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов
продукционная	правило

Различные парадигмы позволяют оптимизировать различные аспекты:

- затраты ресурсов машины (память, время вычисления);

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов
продукционная	правило

Различные парадигмы позволяют оптимизировать различные аспекты:

- затраты ресурсов машины (память, время вычисления);
- затраты человеческих ресурсов (проектирование, написание, отладка);

Парадигмы программирования

Определение

Парадигма программирования — это совокупность идей и понятий, определяющих подход к написанию программ, их стиль, структуры данных и инструментарий.

Определяется **базовой программной единицей** и принципом достижения модульности программы.

Примеры

парадигма	**базовая единица**
императивная	действие
объектно-ориентированная	объект и событие
декларативная	определение
функциональная	функция
логическая	отношение
автоматная	диаграмма переходов
продукционная	правило

Различные парадигмы позволяют оптимизировать различные аспекты:

- затраты ресурсов машины (память, время вычисления);
- затраты человеческих ресурсов (проектирование, написание, отладка);
- надёжность, универсальность и расширяемость программ.

Императивная парадигма

Вычислительный процесс

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

Программа

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n

Программа

```
i := n;
```

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n
2. инициализируй целое res := 1

Программа

```
i := n;
res := 1;
```

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n
2. инициализируй целое res := 1
3. пока i > 0 повторяй шаги 4, 5

Программа

```
i := n;
res := 1;
while i > 0 do


end;
```

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n
2. инициализируй целое res := 1
3. пока i > 0 повторяй шаги 4, 5
4. res := res * i

Программа

```
i := n;
res := 1;
while i > 0 do
  res = i * res;

end;
```

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n
2. инициализируй целое res := 1
3. пока i > 0 повторяй шаги 4, 5
4. res := res * i
5. i := i - 1

Программа

```
i := n;
res := 1;
while i > 0 do
  res = i * res;
  i := i − 1;
end;
```

Императивная парадигма

Вычислительный процесс

Последовательность изменения состояний информационной среды.

Базовые единицы

Действие, инструкция.

Программа

Последовательность инструкций

Данные

Состояние памяти и регистров, значения переменных.

Алгоритм вычисления n!

1. инициализируй целое i := n
2. инициализируй целое res := 1
3. пока i > 0 повторяй шаги 4, 5
4. res := res * i
5. i := i - 1
6. результат — в переменной res

Программа

```
i := n;
res := 1;
while i > 0 do
  res = i * res;
  i := i − 1;
end;
return res;
```

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Программа

```
fact 0 = 1
fact n = product 1 n
```

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Программа

```
fact 0 = 1
fact n = product 1 n
```

Дополнительные определения

```
product a a = a
product a b = a * product (a + 1) b
```

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Программа на языке HASKELL

```
fact = product . naturals
```

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Программа на языке HASKELL

```
fact = product . naturals
```

Дополнительные определения

```
naturals 0 = []
naturals n = n : naturals (n − 1
```

Функциональная парадигма

Вычислительный процесс

Вычисление функций: подстановка аргументов в тело функций.

Базовые единицы

Функции.

Программа

Определение функций.

Данные

Функции и их аргументы.

Определение факториала

Факториал — это произведение ряда натуральных чисел

Программа на языке HASKELL

```
fact = product . naturals
```

Дополнительные определения

```
naturals 0 = []
naturals n = n : naturals (n − 1
product [] = 1
product h : t = h * product t
```

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Данные

Перечислимые множества над которыми заданы отношения.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Данные

Перечислимые множества над которыми заданы отношения.

Отношение n!

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Данные

Перечислимые множества над которыми заданы отношения.

Отношение n!

1. Число 1 является факториалом нуля.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Данные

Перечислимые множества над которыми заданы отношения.

Отношение n!

1. Число 1 является факториалом нуля.
2. Число F является факториалом числа N если существует такое X, что X является факториалом $N-1$ и выполняется равенство $F = N \cdot X$.

Логическая парадигма

Вычислительный процесс

Поиск решения, удовлетворяющего заданным условиям.

Базовые единицы

Отношения.

Программа

Определение отношений и условий.

Данные

Перечислимые множества над которыми заданы отношения.

Отношение n!

1. Число 1 является факториалом нуля.
2. Число F является факториалом числа N если существует такое X, что X является факториалом $N-1$ и выполняется равенство $F = N \cdot X$.

Программа на языке PROLOG

```
fact(0, 1).
fact(N,F) :- fact(N − 1, X),
             F = N * X
```

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.
- Возможность существенной оптимизации кода на этапе трансляции.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.
- Возможность существенной оптимизации кода на этапе трансляции.
- Высокая предметная и аспектная ориентированность кода.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.
- Возможность существенной оптимизации кода на этапе трансляции.
- Высокая предметная и аспектная ориентированность кода.
- Простота переносимости, расширемости, распараллеливания.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.
- Возможность существенной оптимизации кода на этапе трансляции.
- Высокая предметная и аспектная ориентированность кода.
- Простота переносимости, расширемости, распараллеливания.
- Возможность runtime-отладки программы.

Достоинства и недостатки ФП

Достоинства

- Надёжность функциональных программ и систем.
- Возможность строго доказательства корректности программы.
- Высокая степень абстракции и модульности.
- Экономия человеческих ресурсов.
- Возможность существенной оптимизации кода на этапе трансляции.
- Высокая предметная и аспектная ориентированность кода.
- Простота переносимости, расширемости, распараллеливания.
- Возможность runtime-отладки программы.
- Программирование, как обучение и изучение предметной области.

Достоинства и недостатки ФП

Недостатки

- Достаточно высокий уровень вхождения:

Достоинства и недостатки ФП

Недостатки

- Достаточно высокий уровень вхождения:
  - необычная семантика,

Достоинства и недостатки ФП

Недостатки

- Достаточно высокий уровень вхождения:
  - необычная семантика,
  - сильная опора на теоретическую базу.

Достоинства и недостатки ФП

Недостатки

- Достаточно высокий уровень вхождения:
  - необычная семантика,
  - сильная опора на теоретическую базу.
- Сложность создания трансляторов.

Достоинства и недостатки ФП

Недостатки

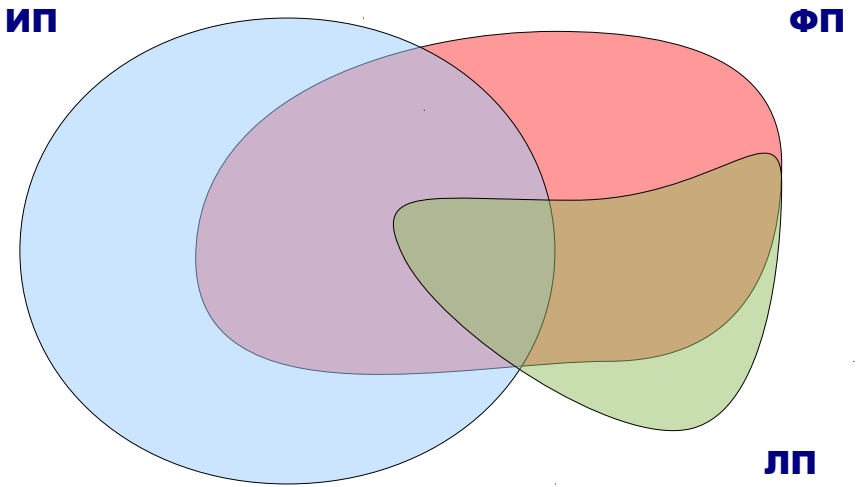
- Достаточно высокий уровень вхождения:
  - необычная семантика,
  - сильная опора на теоретическую базу.
- Сложность создания трансляторов.
- Отсутствие операции присваивания может повысить требования к машинным ресурсам.

Достоинства и недостатки ФП

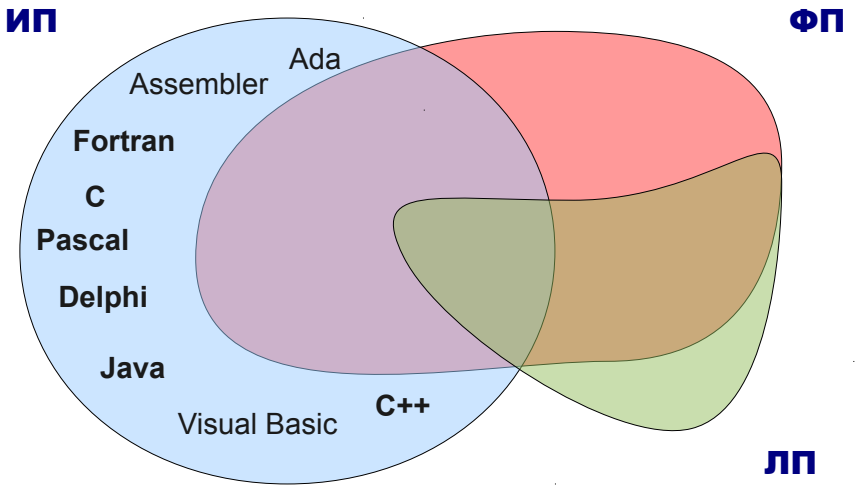
Недостатки

- Достаточно высокий уровень вхождения:
  - необычная семантика,
  - сильная опора на теоретическую базу.
- Сложность создания трансляторов.
- Отсутствие операции присваивания может повысить требования к машинным ресурсам.
- Необходимость в сложном менеджменте памяти

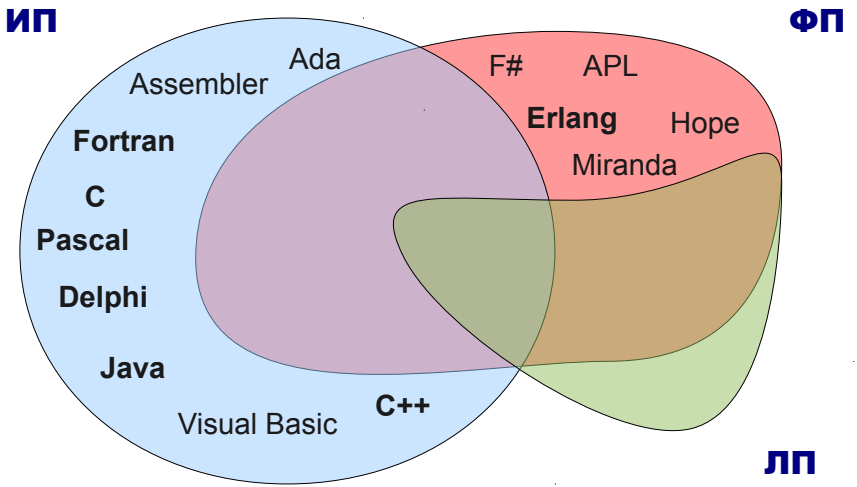
Роль и место ФиЛП среди прочих парадигм



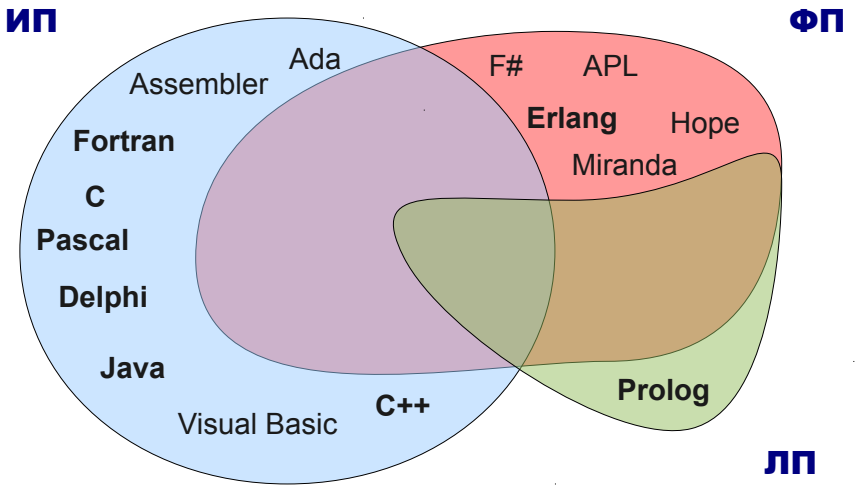
Роль и место ФиЛП среди прочих парадигм



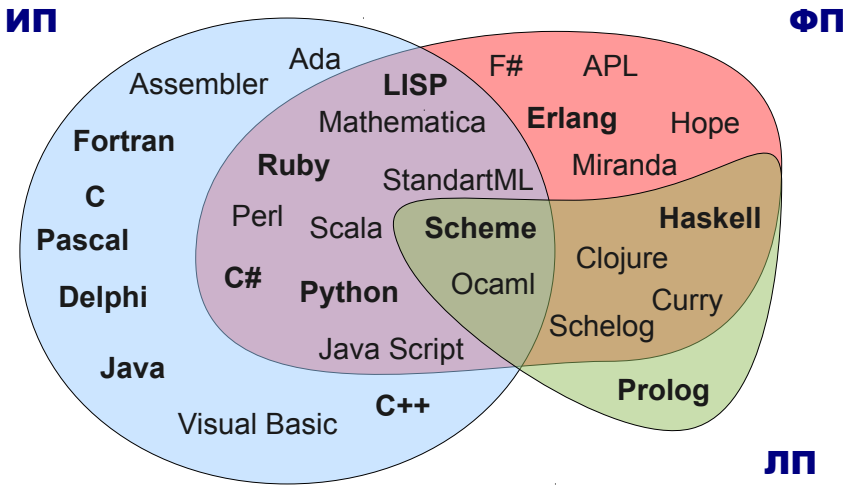
Роль и место ФиЛП среди прочих парадигм



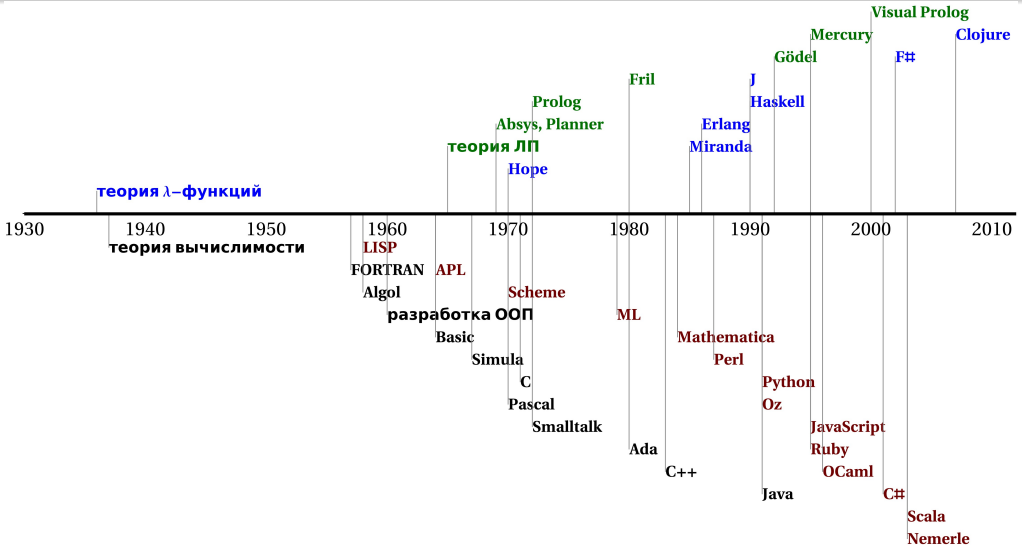
Роль и место ФиЛП среди прочих парадигм



Роль и место ФиЛП среди прочих парадигм



История развития языков программирования



Структура курса

Лекционный курс

Лекция 1. Введение

Структура курса

Лекционный курс

Лекция 1. Введение

Лекция 2. Функции и их свойства

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия
Лекция 4. Абстракция данных

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия
Лекция 4. Абстракция данных
Лекция 5. Абстракция процедур

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия
Лекция 4. Абстракция данных
Лекция 5. Абстракция процедур
Лекция 6. Абстракция структурной рекурсии

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия
Лекция 4. Абстракция данных
Лекция 5. Абстракция процедур
Лекция 6. Абстракция структурной рекурсии
Лекция 7. Ленивые вычисления

Структура курса

Лекционный курс

Лекция 1. Введение
Лекция 2. Функции и их свойства
Лекция 3. Рекурсия
Лекция 4. Абстракция данных
Лекция 5. Абстракция процедур
Лекция 6. Абстракция структурной рекурсии
Лекция 7. Ленивые вычисления
Лекция 8. Модульность и расширяемость

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

Структура курса

Лекционный курс

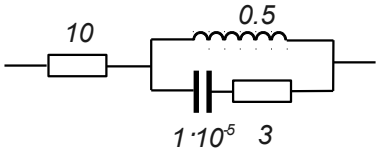
Примеры, которые мы будем рассматривать

Теория и практика:

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,




```
(define circ
  '(- (R 10)
      (|| (L 0.5)
          (- (R 3) (C 1e-5)))))

 > ((impedance circ) 100)
 11.234543+54.3456i
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,

```
(define ADD1
  (Turing-Machine
    '(Start 1) --> '(Start 1 l)
    '(Start 0) --> '(Start 0 l)
    '(Start ()) --> '(Adder () r)
    '(Adder 0) --> '(End 1 p)
    '(Adder 1) --> '(Adder 0 r)
    '(Adder ()) --> '(End 1 p)))

> (ADD1 (tape '(1 0 1)))}
 Program starts
 Start ((1) 0 1)
 Start (1 (0) 1)
 Start (1 0 (1))
 Start (1 0 1 ())
 Adder (1 0 (1))
 Adder (1 (0) 0)
 End (1 (1) 0)
 '(Tape (1) 1 (0))
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,

```
> (begin
    (for ([i 5]) (display "h"))
    (for ([i 5]) (display "e"))
    (for ([i 5]) (display "y")))
hhhhheeeeeyyyyy

> (begin-fork
    (for ([i 5]) (display "h") (pause))
    (for ([i 5]) (display "e") (pause))
    (for ([i 5]) (display "y") (pause)))
heyheyheyheyhey
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,
- принципы Web-программирования.

```
> (define f (applet))
> (define g (applet))
> (g 'start)
Application started.
> (g 'resume)
Input the first number 5
Application stopped.
> (f 'start)
Application started.
> (f 'resume)
Input the first number 6
Application stopped.
> (g 'resume)
Input the second number 2
Web output: 7
> (f 'resume)
Input the second number 4
Web output: 10
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,
- принципы Web-программирования.

Математические задачи:

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,
- принципы Web-программирования.

Математические задачи:

- решение произвольных алгебраических и дифф. уравнений,

```
> (bisection (λ (x) (= x (cos x))) 0 1)
0.7390851332151603
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,
- принципы Web-программирования.

Математические задачи:

- решение произвольных алгебраических и дифф. уравнений,
- оперирование бесконечными последовательностями,

```
> (bisection (λ (x) (= x (cos x))) 0 1)
0.7390851332151603

> (define (enum n) (cons n (enum n)))
> (enum 1)
1 2 3 4 5 6 7 ...
> (mult~ (enum 1) (enum 1))
1 4 9 16 25 36 49 ...
```

Примеры, которые мы будем рассматривать

Теория и практика:

- язык для описания электрических схем,
- язык для машины Тьюринга,
- создание ООП, многопоточности,
- принципы Web-программирования.

Математические задачи:

- решение произвольных алгебраических и дифф. уравнений,
- оперирование бесконечными последовательностями,
- статистическое моделирование дорожного движения.

```
> (bisection (λ (x) (= x (cos x))) 0 1)
0.7390851332151603

> (define (enum n) (cons n (enum n)))
> (enum 1)
1 2 3 4 5 6 7 ...
> (mult~ (enum 1) (enum 1))
1 4 9 16 25 36 49 ...
```

```
;; вероятность столкновения двух
;; агрессивных водителей за цикл светофора
(probability
  (do [c <- (Dist: ('G 30) ('Y 10) ('R 30))]
      [(a1 a2) <- (true 9/10)]
      [coll <- (collide 'Agressive
                        'Agressive c)]
      (return (and a1 a2 coll)))))
0.09060428571428572
```

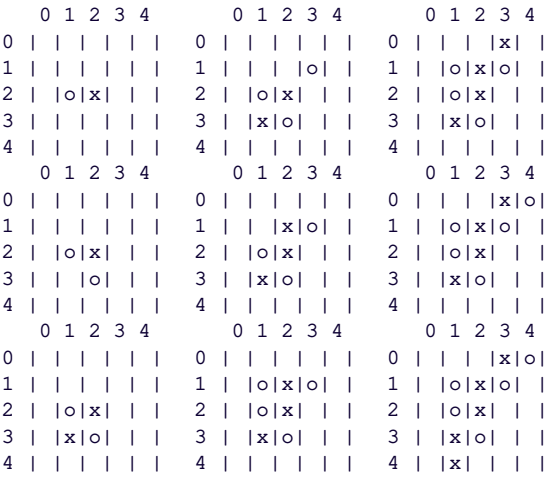
Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,



Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,
- распознавание образов, адаптивный спам-фильтр,

```
> (spam-test "Hello,
   Please call me as soon as possible!
   Sincerely, Dr Green.")
'Ok.

> (spam-test "Only here! Only today!
    Get job without trouble!!")
'spam!

> (spam-test "Hi,
    Just a reminder: don't forget your
    allergy prescription when you
    visit home today.
    Mom.")
'check.
```

Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,
- распознавание образов, адаптивный спам-фильтр,
- упрощение выражений и символьное дифференцирование,
- автоматическое доказательство теорем,

```
; упрощение выражения
(expand '((B || A) && (A || ! B))))
'A

; силлогизм
(expand '((человек => смертен) &&
          (Сократ => человек) =>
          (Сократ => смертен)))
'T

; принцип доказательства от противного
(expand '((! A => B) && (! A => ! B)
=> A))
'T
```

Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,
- распознавание образов, адаптивный спам-фильтр,
- упрощение выражений и символьное дифференцирование,
- автоматическое доказательство теорем,
- интерпретация естественного языка,

```
> (show-parsing "mi wile li moku
                 e kili mute mi.")

Простое предложение
 с выражением намерения,
 дополненное определением.

((подлежащее "mi")
 (сказуемое (намерение "moku"))
 (дополнение
  (определение
   (притяж./накл. "mi")
   (множ.число "kili"))))
```

Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,
- распознавание образов, адаптивный спам-фильтр,
- упрощение выражений и символьное дифференцирование,
- автоматическое доказательство теорем,
- интерпретация естественного языка,
- решение логических задач и головоломок.

Как следует расставить знаки арифметических операций: $+, -, \times, /$ в выражении

$$(((1\ ?\ 2)\ ?\ 3)\ ?\ 4)\ ?\ 5,$$

чтобы в результате получилось заданное число, например, 9?

```
(collect (printf "(((1 ~a 2) ~a 3) ~a 4) ~a 5 = 9\n"
                 op4 op3 op2 op1)
  [(op1 op2 op3 op4) <- '(+ - * /)]
  (= 9 (eval `((nest ,op1 ,op2 ,op3 ,op4) 1 2 3 4 5))))
```

```
(((1 + 2) - 3) + 4) + 5 = 9
(((1 + 2) / 3) * 4) + 5 = 9
(((1 / 2) + 3) * 4) - 5 = 9
```

Какие примеры мы будем рассматривать

Решение задач искусственного интеллекта:

- игра в крестики-нолики и в линии,
- распознавание образов, адаптивный спам-фильтр,
- упрощение выражений и символьное дифференцирование,
- автоматическое доказательство теорем,
- интерпретация естественного языка,
- решение логических задач и головоломок.

A говорит, что B отрицает, что C считает, что A всегда врёт, но в свою очередь, B слыхал, что A не склонен верить C. При этом C уверяет, что слышал, как A всем рассказывал о том, что B считает C лжецом. Выясните, кто из них говорит правду, а кто врёт.

```
(define says
  (/. x x --> #t
      x y --> #f))

(collect '((A ,A) (B ,B) (C ,C))
  [(A B C) <- '(#f #t)]
  (says A (says B (not (says C (not A)))))
  (says B (says A (not C)))
  (says C (says A (says B (not C)))))
```

```
((A #f) (B #t) (C #t))
```

Что даёт изучение ФП

- Понимание глубоких основ программирования:

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:
  - элементы ФП включаются во все современные ЯП: Perl, Python, Ruby, C#, JavaScript, и т.д.

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:
  - элементы ФП включаются во все современные ЯП: Perl, Python, Ruby, C#, JavaScript, и т.д.
- Выработка хорошего стиля и навыков технологии программирования:

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:
  - элементы ФП включаются во все современные ЯП: Perl, Python, Ruby, C#, JavaScript, и т.д.
- Выработка хорошего стиля и навыков технологии программирования:
  - расширяемость и сопровождаемость программ

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:
  - элементы ФП включаются во все современные ЯП: Perl, Python, Ruby, C#, JavaScript, и т.д.
- Выработка хорошего стиля и навыков технологии программирования:
  - расширяемость и сопровождаемость программ
  - юнит-тестирование и верификация

Что даёт изучение ФП

- Понимание глубоких основ программирования:
  - абстракция данных,
  - абстракция процедур
  - абстракция циклических процессов
  - абстракция времени
  - абстракция вычислений
- Повышение профессионального уровня и расширение инструментария:
  - элементы ФП включаются во все современные ЯП: Perl, Python, Ruby, C#, JavaScript, и т.д.
- Выработка хорошего стиля и навыков технологии программирования:
  - расширяемость и сопровождаемость программ
  - юнит-тестирование и верификация
  - изящество, как принцип построения программы.

Язык программирования Formica

Lisp (1957) — первая реализация ФП, S-выражения, макропрограммирование.

Scheme (1972) — Lisp + гигиеничные макросы, хвостовая рекурсия, единое пространство имён и функций, продолжения.

Racket (2008) — Scheme + JIT, неизменяемые данные, система модулей, сопоставление с образцом, контракты, возможность статической типизации.

Formica (2012) — Racket + формальные функции, подстановки, монады, сечения функций, контрактная типизация, обобщённая композиция.

Язык программирования Racket

Язык для исследования и разработки
новых концепций, и новых языков программирования.

Язык для внедерения новейших технологий
в повседневную практику.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.
- Обработка текстовых и символьных данных.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.
- Обработка текстовых и символьных данных.
- Лексический анализ и трансляция.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.
- Обработка текстовых и символьных данных.
- Лексический анализ и трансляция.
- Задачи искусственного интеллекта.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.
- Обработка текстовых и символьных данных.
- Лексический анализ и трансляция.
- Задачи искусственного интеллекта.
- Системное программирование.

Язык программирования Racket

Общая характеристика

- Является расширением языка SCHEME
- Потомок языка LISP
- Использует префиксную скобочную нотацию.
- Мультипарадигменный.
- Интерпретируемый, кроссплатформенный.
- Типизация: динамическая, сильная.
- Поощряет функциональный стиль.

Использование

- Изучение и разработка различных парадигм и концепций программирования.
- Разработка новых языков программирования.
- Прототипирование.
- Web-программирование, серверы.
- Обработка текстовых и символьных данных.
- Лексический анализ и трансляция.
- Задачи искусственного интеллекта.
- Системное программирование.
- Создание графических интерфейсов.

Примеры программ

Hello world!

```
(display "Hello world!")
```

Примеры программ

Hello world!

```
(display "Hello world!")
```

Простейший web-сервер

```
#lang web-server/insta
;; A "hello world" web server
(define (start request)
  (response/xexpr
   '(html
     (body "Hello World"))))
```

Примеры программ

Hello world!

```
(display "Hello world!")
```

Простейший web-сервер

```
#lang web-server/insta
;; A "hello world" web server
(define (start request)
  (response/xexpr
   '(html
     (body "Hello World"))))
```

Игра в угадайку

```
#lang racket/gui
(define f (new frame% [label "Guess"]))

(define (check i)
  (let [(n (random 5))]
    (λ (btn evt)
      (message-box "."
        (if (= i n) "Yes" "No")))))

(for ([i 5])
  (make-object button%
    (format "~a" i) f (check i)))

(send f show #t)
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

Pascal

```
Program NameOnList (Input, Output) ;
Type
  ListType = ^NodeType;
  NodeType = Record
                  First : String;
                  Rest : ListType
                End;
Var
  List : ListType;
  Name : String;

Procedure GetList (Var List: ListType; fname : String);
...

Function Guest (Name : String; List : ListType) : String;
Begin
  If List = nil
     Then Guest := "no"
     Else If Name = List^.First
        Then Guest := "yes"
        Else Guest := Guest ( Name, List^.Rest)
End;
Begin
  Readln ( Name );
  GetList ( List, "guests.txt" );
  Writeln (Guest ( Name, List ) )
End.
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ 'null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

Pascal

```
Program NameOnList (Input, Output) ;
Type
  ListType = ^NodeType;
  NodeType = Record
                  First : String;
                  Rest : ListType
                End;
Var
  List : ListType;
  Name : String;

Procedure GetList (Var List: ListType; fname : String);
...
Function InList (Name : String; List : ListType) : String;
Begin
  If List = nil
     Then InList := "no"
     Else If Name = List^.First
        Then InList := "yes"
        Else InList := InList ( Name, List^.Rest)
End;
Begin
  Readln ( Name );
  GetList ( List, "guests.txt" );
  Writeln (InList ( Name, List ) )
End.
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define-formal cons)

(define in-list?
  (/. _ 'null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

Pascal

```
Program NameOnList (Input, Output) ;
Type
  ListType = ^NodeType;
  NodeType = Record
                   First : String;
                   Rest : ListType
                 End;
Var
  List : ListType;
  Name : String;

Procedure GetList (Var List: ListType; fname : String);
...
Function InList (Name : String; List : ListType) : String;
Begin
  If List = nil
     Then InList := "no"
     Else If Name = List^.First
        Then InList := "yes"
        Else InList := InList ( Name, List^.Rest)
End;
Begin
  Readln ( Name );
  GetList ( List, "guests.txt" );
  Writeln (InList ( Name, List ) )
End.
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

C#

```
public bool InList(string name; string[] list)
{
    bool iAfterE = list.Any(w => w.Contains(name));

    if (iAfterE)
    {
       Console.WriteLine("yes")
    }
    else
    {
       Console.WriteLine("no")
    };
}
```

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

C#

```
public bool InList(string name; string[] list)
{
    bool iAfterE = list.Any(w => w.Contains(name));

    if (iAfterE)
    {
       Console.WriteLine("yes")
    }
    else
    {
       Console.WriteLine("no")
    };
}
```

Использует функциональную библиотеку LINQ.

Почему Formica?

Задача

Выяснить, содержится ли заданное имя в списке гостей.
Длина списка произвольна и заранее неизвестна.

Formica

```
(define in-list?
  (/. _ null --> 'no
      name (cons name _) --> 'yes
      name (cons _ rest) --> (in-list? name (rest))))

(define (names p)
  (cons (read p) (λ () (names p))))

(in-list? (read)
          (names (open-input-file "guests.txt")))
```

C#

```
public bool InList(string name; string[] list)
{
    bool iAfterE = list.Any(w => w.Contains(name));

    if (iAfterE)
    {
       Console.WriteLine("yes")
    }
    else
    {
       Console.WriteLine("no")
    };
}
```

Использует функциональную библиотеку LINQ.

```
(define (in-list? n l)
  (display (if (member n l) 'yes 'no)))
```

Почему Formica?

Задача

Имеется четыре списка слов:

- `("the" "that" "a")`
- `("frog" "elephant" "thing" "turtle")`
- `("walked" "eats" "treaded" "grows")`
- `("slowly" "quickly" "salad")`

Необходимо максимально эффективным способом найти, какие фразы можно составить, используя ровно по одному слову из всех этих списков, при двух условиях: 1) слова должны следовать в том же порядке, в котором указаны списки и 2) в каждой последовательной паре слов последняя буква первого слова должна совпадать с первой буквой второго.

Почему Formica?

Задача

Имеется четыре списка слов:

- `("the" "that" "a")`
- `("frog" "elephant" "thing" "turtle")`
- `("walked" "eats" "treaded" "grows")`
- `("slowly" "quickly" "salad")`

Необходимо максимально эффективным способом найти, какие фразы можно составить, используя ровно по одному слову из всех этих списков, при двух условиях: 1) слова должны следовать в том же порядке, в котором указаны списки и 2) в каждой последовательной паре слов последняя буква первого слова должна совпадать с первой буквой второго.

Haskell

```
import Control.Monad

joins left right = last left == head right

do
  w1 <- ["the", "that", "a"]
  w2 <- ["frog", "elephant", "thing"]
  unless (joins w1 w2) []
  w3 <- ["walked", "treaded", "grows"]
  unless (joins w2 w3) []
  w4 <- ["slowly", "quickly"]
  unless (joins w3 w4) []
  return (unwords [w1, w2, w3, w4])
```

Почему Formica?

Задача

Имеется четыре списка слов:

- `("the" "that" "a")`
- `("frog" "elephant" "thing" "turtle")`
- `("walked" "eats" "treaded" "grows")`
- `("slowly" "quickly" "salad")`

Необходимо максимально эффективным способом найти, какие фразы можно составить, используя ровно по одному слову из всех этих списков, при двух условиях: 1) слова должны следовать в том же порядке, в котором указаны списки и 2) в каждой последовательной паре слов последняя буква первого слова должна совпадать с первой буквой второго.

Haskell

```
import Control.Monad

joins left right = last left == head right

do
  w1 <- ["the", "that", "a"]
  w2 <- ["frog", "elephant", "thing"]
  unless (joins w1 w2) []
  w3 <- ["walked", "treaded", "grows"]
  unless (joins w2 w3) []
  w4 <- ["slowly", "quickly"]
  unless (joins w3 w4) []
  return (unwords [w1, w2, w3, w4])
```

Использует библиотеку `Control.Monad`

Почему Formica?

Задача

Имеется четыре списка слов:

- `("the" "that" "a")`
- `("frog" "elephant" "thing" "turtle")`
- `("walked" "eats" "treaded" "grows")`
- `("slowly" "quickly" "salad")`

Необходимо максимально эффективным способом найти, какие фразы можно составить, используя ровно по одному слову из всех этих списков, при двух условиях: 1) слова должны следовать в том же порядке, в котором указаны списки и 2) в каждой последовательной паре слов последняя буква первого слова должна совпадать с первой буквой второго.

Racket

```
(define (joins? a b)
  (eq? (last (string->list a))
       (first (string->list b))))

(do
  [w1 <- '("the" "that" "a")]
  [w2 <- '("frog" "elephant" "thing")]
  (guard  (joins? w1 w2))
  [w3 <- '("walked" "treaded" "grows")]
  (guard  (joins? w2 w3))
  [w4 <- '("slowly" "quickly")]
  (guard  (joins? w3 w4))
  (list (list w1  w2  w3  w4)))
```

Почему Formica?

Задача

Имеется четыре списка слов:

- `("the" "that" "a")`
- `("frog" "elephant" "thing" "turtle")`
- `("walked" "eats" "treaded" "grows")`
- `("slowly" "quickly" "salad")`

Необходимо максимально эффективным способом найти, какие фразы можно составить, используя ровно по одному слову из всех этих списков, при двух условиях: 1) слова должны следовать в том же порядке, в котором указаны списки и 2) в каждой последовательной паре слов последняя буква первого слова должна совпадать с первой буквой второго.

Racket

```
(define (joins? a b)
  (eq? (last (string->list a))
       (first (string->list b))))

(do
  [w1 <- '("the" "that" "a")]
  [w2 <- '("frog" "elephant" "thing")]
  (guard  (joins? w1 w2))
  [w3 <- '("walked" "treaded" "grows")]
  (guard  (joins? w2 w3))
  [w4 <- '("slowly" "quickly")]
  (guard  (joins? w3 w4))
  (list (list w1  w2  w3  w4)))

(define-syntax do
  (syntax-rules (<-)
    [(do (b) r) (bind b r)]
    [(do (x <- p) r) (bind p (λ (x) r))]
    [(do b bs ... r) (do b (do bs ... r))]))

(define (bind x f)
  (fold (nest append f) (list) x)

(define (guard expr)
  (if expr (list null) (list))
```

Почему не C#?

```
using System;
using System.Collections.Generic;

public class Amb : IDisposable
{
    List <IValueSet> streams = new List<IValueSet>();
    List <IAssertOrAction> assertsOrActions = new List<IAssertOrAction>();
    volatile bool stopped = false;

    public IAmbValue<T> DefineValues<T>(params T[] values)
    {
        return DefineValueSet(values);
    }

    public IAmbValue<T> DefineValueSet<T>(IEnumerable<T> values)
    {
        ValueSet<T> stream = new ValueSet<T>();
        stream.Enumerable = values;
        streams.Add(stream);
        return stream;
    }

    public Amb Assert(Func<bool> function)
    {
        assertsOrActions.Add(new AmbAssert()
        {
            Level = streams.Count,
            IsValidFunction = function
        });
        return this;
    }
```

Почему не C#?

```
using System;
using System.Collections.Generic;

public class Amb : IDisposable
{
    List <IValueSet> streams = new List<IValueSet>();
    List <IAssertOrAction> assertsOrActions = new List<IAssertOrAction>();
    volatile bool stopped = false;

    public IAmbValue<T> DefineValues<T>(params T[] values)
    {
        return DefineValueSet(values);
    }

    public IAmbValue<T> DefineValueSet<T>(IEnumerable<T> values)
    {
        ValueSet<T> stream = new ValueSet<T>();
        stream.Enumerable = values;
        streams.Add(stream);
        return stream;
    }

    public Amb Assert(Func<bool> function)
    {
        assertsOrActions.Add(new AmbAssert()
        {
            Level = streams.Count,
            IsValidFunction = function
        });
        return this;
    }

    public Amb Perform(Action action)
    {
        assertsOrActions.Add(new AmbAction()
        {
            Level = streams.Count,
            Action = action
        });
        return this;
    }

    public void Stop()
    {
        stopped = true;
    }

    public void Dispose()
    {
        RunLevel(0, 0);
        if (!stopped)
        {
            throw new AmbException();
        }
    }

    void RunLevel(int level, int actionIndex)
    {
        while (actionIndex < assertsOrActions.Count && assertsOrActions[actionIndex].Leve
        {
            if (!assertsOrActions[actionIndex].Invoke() || stopped)
                return;
            actionIndex++;
        }
```

Почему не C#?

```
if (level < streams.Count)
        {
            using (IValueSetIterator iterator = streams[level].CreateIterator())
            {
                while (iterator.MoveNext())
                {
                    RunLevel(level + 1, actionIndex);
                }
            }
        }
    }

  interface IValueSet
    {
        IValueSetIterator CreateIterator();
    }

    interface IValueSetIterator : IDisposable
    {
        bool MoveNext();
    }

    interface IAssertOrAction
    {
        int Level { get; }
        bool Invoke();
    }
```

Почему не C#?

```
if (level < streams.Count)
        {
            using (IValueSetIterator iterator = streams[level].CreateIterator())
            {
                while (iterator.MoveNext())
                {
                    RunLevel(level + 1, actionIndex);
                }
            }
        }
    }

    interface IValueSet
    {
        IValueSetIterator CreateIterator();
    }

    interface IValueSetIterator : IDisposable
    {
        bool MoveNext();
    }

    interface IAssertOrAction
    {
        int Level { get; }
        bool Invoke();
    }
```

```
class AmbAssert : IAssertOrAction
{
    internal int Level;
    internal Func<bool> IsValidFunction;

    int IAssertOrAction.Level { get { return Level; } }

    bool IAssertOrAction.Invoke()
    {
        return IsValidFunction();
    }
}

class AmbAction : IAssertOrAction
{
    internal int Level;
    internal Action Action;

    int IAssertOrAction.Level { get { return Level; } }

    bool IAssertOrAction.Invoke()
    {
        Action(); return true;
    }
}
```

Почему не C#?

```
    class ValueSet<T> : IValueSet, IAmbValue<T>, IValueSetIterator
    {
        internal IEnumerable<T> Enumerable;
        private IEnumerator<T> enumerator;

        public T Value { get { return enumerator.Current; } }

        public IValueSetIterator CreateIterator()
        {
            enumerator = Enumerable.GetEnumerator();
            return this;
        }

        public bool MoveNext()
        {
            return enumerator.MoveNext();
        }

        public void Dispose()
        {
            enumerator.Dispose();
        }
    }
}

public interface IAmbValue<T>
{ T Value { get; } }

public class AmbException : Exception
{
    public AmbException() : base("AMB is angry") { }
}
```

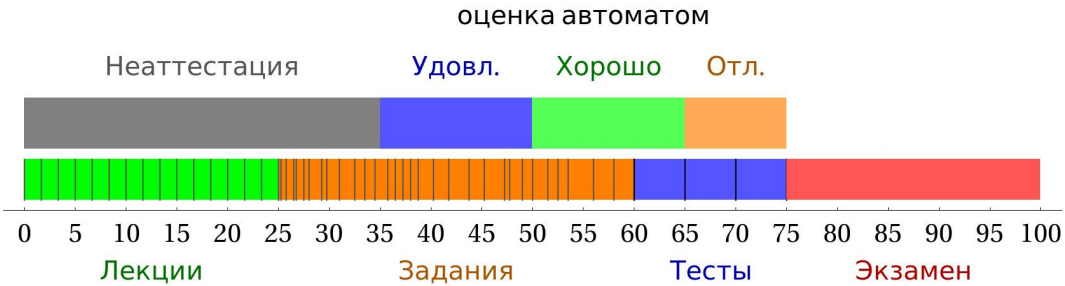
Почему не C#?

```
    class ValueSet<T> : IValueSet, IAmbValue<T>, IValueSetIterator
    {
        internal IEnumerable<T> Enumerable;
        private IEnumerator<T> enumerator;

        public T Value { get { return enumerator.Current; } }

        public IValueSetIterator CreateIterator()
        {
            enumerator = Enumerable.GetEnumerator();
            return this;
        }

        public bool MoveNext()
        {
            return enumerator.MoveNext();
        }

        public void Dispose()
        {
            enumerator.Dispose();
        }
    }
}

public interface IAmbValue<T>
{ T Value { get; } }

public class AmbException : Exception
{
    public AmbException() : base("AMB is angry") { }
}
```

```
using (Amb amb = new Amb())
{
    var set1 = amb.DefineValues("the", "that", "a");
    var set2 = amb.DefineValues("frog", "elephant", "thing");
    var set3 = amb.DefineValues("walked", "treaded", "grows");
    var set4 = amb.DefineValues("slowly", "quickly");

    amb.Assert(() => IsJoinable(set1.Value, set2.Value));
    amb.Assert(() => IsJoinable(set2.Value, set3.Value));
    amb.Assert(() => IsJoinable(set3.Value, set4.Value));

    amb.Perform(() =>
        {
            System.Console.WriteLine("{0} {1} {2} {3}",
              set1.Value, set2.Value, set3.Value, set4.Value);
            amb.Stop();
        });
}

using (Amb amb = new Amb())
{
    IAmbValue<int> x = amb.DefineValues(1, 2, 3);
    IAmbValue<int> y = amb.DefineValues(4, 5, 6);
    amb.Assert(() => x.Value * y.Value == 8);
    amb.Perform(() =>
        {
            System.Console.WriteLine("{0} {1}", x.Value, y.Value);
            amb.Stop();
        });
}
```

Вывод: разным задачам — разные парадигмы

Рейтинги и оценки



Необходимые минимумы

- Отлично = задания (все!) + лекции + 5
- Хорошо = задания + экзамен | задания + тесты
- Удовл. = лекции + тесты | задания

Доступные материалы

- Конспект лекций (слайды).
- Книга + DrRacket.
- Дополнительная литература.