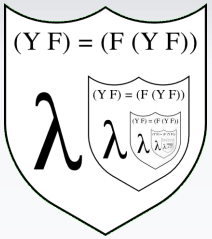
Функциональное программирование
Вводная часть

А.Н. Непейвода

2023 г.

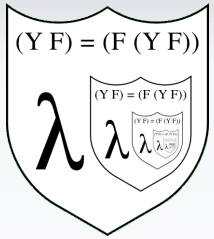


Побочные эффекты

```
bool flag;
int f (int n) {
  int retVal;
  if flag {retVal = 2*n;}
      else retVal = n;
  flag != flag;
  return retVal;
 }
```

```
void test () {
  flag = true;
  printf("f(1)+f(2) = %d\n",
            f(1)+f(2));
  printf("f(2)+f(1) = %d\n",
            f(2)+f(1));
 }
```

Что вернет функция test?



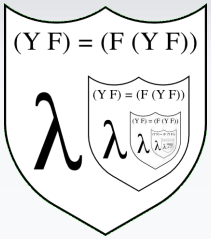
Побочные эффекты

```
bool flag;
int f (int n) {
  int retVal;
  if flag {retVal = 2*n;}
      else retVal = n;
  flag != flag;
  return retVal;
 }
```

```
void test () {
  flag = true;
  printf("f(1)+f(2) = %d\n",
            f(1)+f(2));
  printf("f(2)+f(1) = %d\n",
            f(2)+f(1));
 }
```

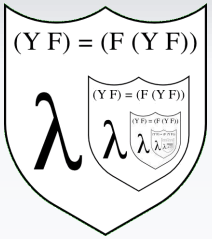
Что вернет функция test?

```
f(1)+f(2) = 4
f(2)+f(1) = 5
```



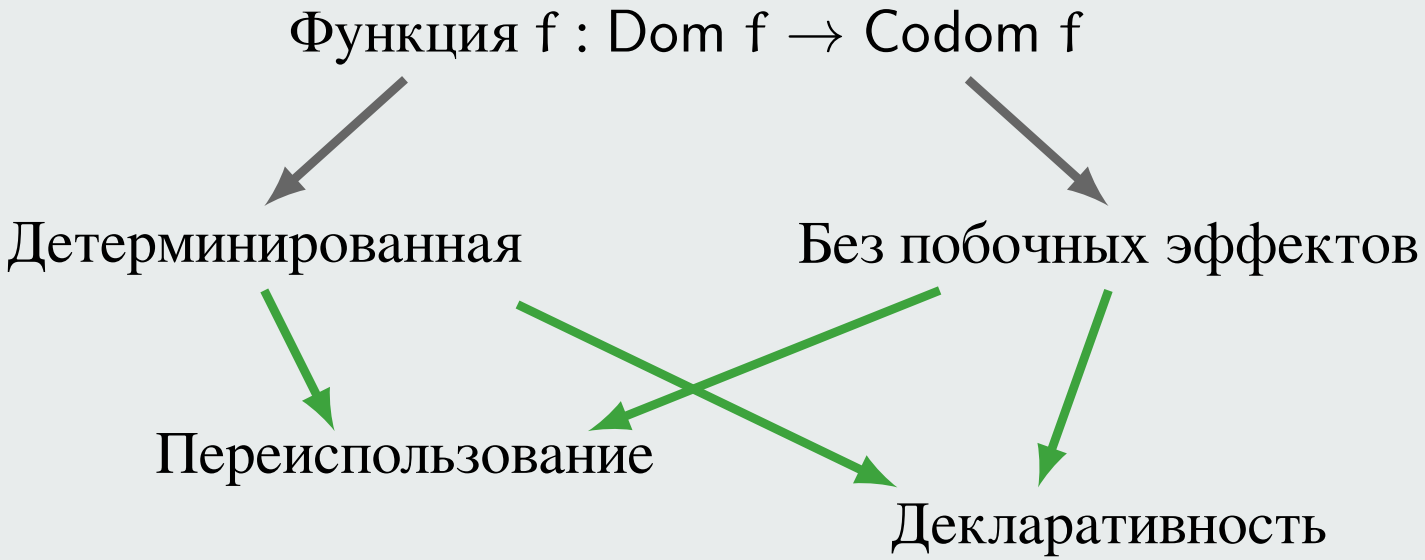
Основное определение

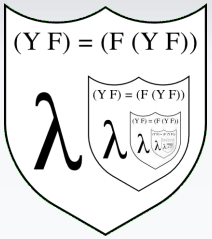
ФП (парадигма программирования) — основной вычислительный процесс рассматривается как вычисление значений математических функций.



Основное определение

ФП (парадигма программирования) — основной вычислительный процесс рассматривается как вычисление значений математических функций.



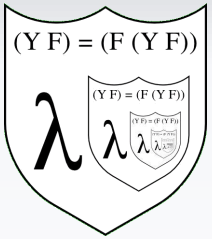


Чистое ФП

Чистая функция в ФП — функция, поведение которой аналогично математической.

Не являются чистыми:

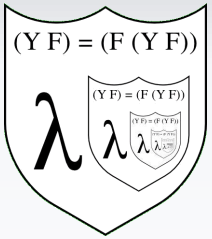
- IO-функции;
- rand-функции;
- функции, зависящие от или меняющие глобальное состояние среды.



Свойства ФП

Cons

- Фон-неймановское горлышко
- Интерактивность



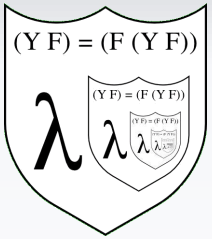
Свойства ФП

Pros

- "Носят семантику с собой"
- Верифицируемость (!!)
- Параметризуемость
- Быстрое прототипирование

Cons

- Фон-неймановское горлышко
- Интерактивность



Свойства ФП

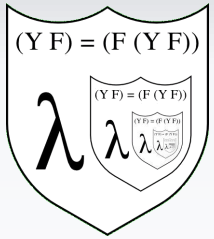
Pros

- "Носят семантику с собой"
- Верифицируемость (!!)
- Параметризуемость
- Быстрое прототипирование

Cons

- Фон-неймановское горлышко
- Интерактивность

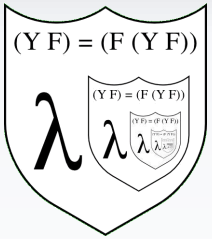
- ФЯП уже эффективны для реализации массивного параллелизма и многопоточности (Scala, Erlang).
- Интерактивность — дополнительная инкапсуляция (монадическое программирование, реактивное программирование).
- Высокоуровневые оптимизации.



Миграция в массовые ЯП

- Анонимные функции и функции высших порядков.
- Развитое сопоставление с образцом (record patterns в Java, etc).
- Алгебраические типы данных.
- Функторы и монады (асинхронные вычисления).
- Средства метапрограммирования.
- Системы типизации в стиле соответствия Карри–Ховарда.

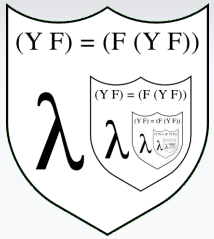
«Все там будем»: академические ФЯП — это песочница, где зарождается дизайн современных ЯП вообще.



Теоретические основы ФП как паттерны проектирования

- Инкапсуляция побочных эффектов
- Высокая параметризуемость без потери надёжности
- Перенос оптимизаций на абстрактный уровень

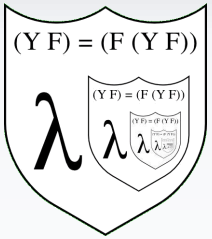
После 2010 года — стремительный рост публикаций о применении теории категорий в программной инженерии. Внедрение паттернов на основе алгебраических понятий в дисциплину программирования.



Структура курса

- Два языка: Рефал (разминочный), Haskell (центральный).
- Бестиповое и типизированное лямбда-исчисление.
- Начала теории категорий.

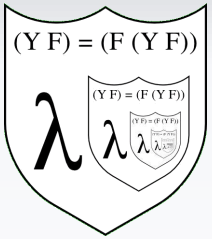
- 5 лабораторных работ $\times$ 8 баллов
- 2 РК $\times$ 15 баллов



Модели вычислений («системы координат»)

- Машины Тьюринга (автоматы) — максимально императивны, манипуляция строками
- Рекурсивные функции Клини (структурное программирование) — всё ещё императивны, манипуляция функциями
- Алгорифмы Маркова (системы переписывания термов) — декларативны, манипуляция строками
- Лямбда-исчисление (системы функций высшего порядка) — декларативны, манипуляция функциями

Манипуляция строками $\Rightarrow$ программа как «белый ящик»
Манипуляция функциями $\Rightarrow$ программа как «чёрный ящик»



Алгорифмы Маркова

Синтаксис

Два вида правил:

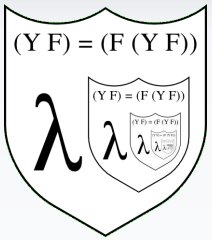
$$\begin{array}{lll} \Phi_i & \to & \Psi_i \quad \text{(нефинальные)} \\ \Phi'_i & \to_* & \Psi'_i \quad \text{(финальные)} \end{array}$$

Семантика

- Данные — единственная строка.
- Правила просматриваются сверху вниз по списку.
- Выбирается самое левое вхождение подстроки Φ_i, после чего в случае финального правила оно просто заменяется на Ψ_i, а в случае нефинального — заменяется на Ψ_i, и операция повторяется над изменённой строкой.

По сути — рекурсивный вызов функции, разбивающей строку по шаблону $(.^*)\Phi_i(.^*)$. С именованными группами — как-то так:

$(?\texttt{<x1>}.^*?)\Phi_i(?\texttt{<x2>}.^*) \to \texttt{group(x1)} \mathbin{+\!\!+} \Psi_i \mathbin{+\!\!+} \texttt{group(x2)}$

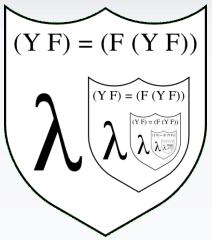


Алгорифмы Маркова как рекурсивная функция

- С учётом того, что группы всегда соответствуют выражениям $.^*$ (на самом деле — $.*?$), заменим их просто именами.
- Конкатенацию сделаем неявной не только в регулярках, но и в результате.
- Результат: рекурсивная функция следующего вида.

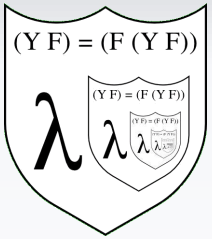
$$
\begin{array}{lcl}
\mathsf{f}(\mathsf{x}_1\ \Phi_1\ \mathsf{x}_2) & = & \mathsf{f}(\mathsf{x}_1\ \Psi_1\ \mathsf{x}_2) \\
 & \dots & \\
\mathsf{f}(\mathsf{x}_1\ \Phi_1'\ \mathsf{x}_2) & = & \mathsf{x}_1\ \Psi_1'\ \mathsf{x}_2 \\
\mathsf{f}(\mathsf{x}_1\ \Phi_\mathsf{k}\ \mathsf{x}_2) & = & \mathsf{f}(\mathsf{x}_1\ \Psi_\mathsf{k}\ \mathsf{x}_2) \\
 & \dots &
\end{array}
$$

Все левые части содержат один и тот же вызов, так что упоминание об f там избыточно. Получился примитивный ЯП.



Рефал: засахаренные алгорифмы Маркова

- Свободные образцы: вместо примитивного образца — всё множество возможных образцов над свободным моноидом, в том числе с повторными переменными.
- «Много функций»: введение идентификатора функции, стоящего в образце самым первым и определяющего выбор подмножества правил переписывания.
- Вложенные структуры: использование лесов строк (дополнительного конструктора в моноиде) в качестве данных программы.



Над тезисом Чёрча–Тьюринга

От нормальных алгорифмов к машинам Тьюринга

- Введём символ «головка машины» τ и символы состояний ξ_j. Потребуем Φ_i и Ψ_i всегда начинаться с символа τ, за которым следует ξ_j и единственная буква строки.
- Правила становятся коммутирующими; ленивое сопоставление также больше не актуально.

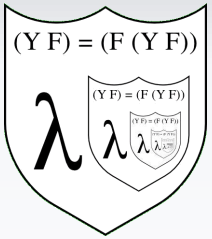
Обратно

- Воспользуемся расширенным тезисом Чёрча–Тьюринга...

Наблюдение

Переход от алгорифмов к другим моделям вычислений очень прост — почти все преобразования такого типа сводимы к применению системы переписывания термов, которая выражается ~~нормальным алгорифмом~~ Рефал-программой.

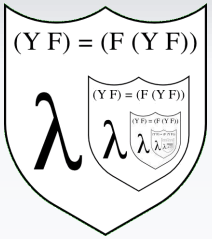
Условно, нормальные алгорифмы являются «абсолютной комонадой» (полностью «белый ящик»).



Декларативные объявления

```
f [] = ...
f [x] = ...
f (x : _) = ...
```

- Частичный разбор терма начиная с внешнего конструктора.
- Сопоставление сверху вниз (аналогично цепочке `if–elif–else`).
- Не перестановочные предложения: третье в том числе описывает и случай, когда список одноэлементный (т.к. алгебраически список определяется конструкторами `:` (т.е. `cons`) и `[]`, т.е. `nil`).



Регулярные выражения

Проблема неоднозначности разбора строковых данных уже решалась в рамках построения механизма сопоставления с регулярными выражениями.

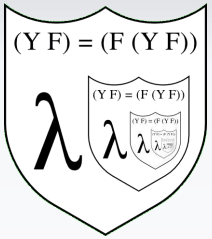
- Жадные квантификаторы — забирают максимально длинную подстроку

```
match = re.search(r'\( .* \)', r'0( (12)3)4(56 )7')
# match[0] = ((12)3)4(56)
```

- Ленивые квантификаторы — забирают максимально короткую подстроку

```
match = re.search(r'\( .*? \)', r'0( (12 )3)4(56)7')
# match[0] = ((12)
```

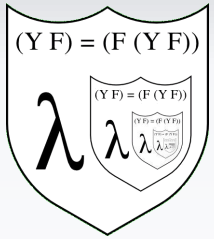
Здесь `.` — произвольная буква, `*` — квантификатор: повторять образец 0 или больше раз, `\(` и `\)` — экранированные представления круглых скобок.



Ассоциативные образцы

Для краткости ++ опускается. Например: `x'A'y` — сокращённая запись для `x ++ ['A'] ++ y`.

- Сопоставление ленивое, сверху вниз.
- Эффективный доступ к строке с начала, с конца или с середины (например, при использовании суффиксного массива). Условный "шаг" — вызов сопоставления. С точки зрения реализации, "шаг" имеет сложность, большую, чем $O(1)$.
- Строки — очевидно. Как добавить леса с доступом "с середины"?

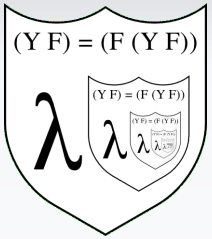


Ассоциативные образцы

- Сопоставление ленивое, сверху вниз.
- Эффективный доступ к строке с начала, с конца или с середины (например, при использовании суффиксного массива). Условный "шаг" — вызов сопоставления. С точки зрения реализации, "шаг" имеет сложность, большую, чем $O(1)$.
- Строки — очевидно. Как добавить леса с доступом "с середины"? Решение: дополнительный конструктор, добавляющий глубину вложенности.

Дан список списков. Выделить список, содержащий букву `'A'`.

`(list x1 (list z1 'A' z2) x2)` — с тегами (АТД);
`(x1 (z1 'A' z2) x2)` — без тегов (ака Рефал);



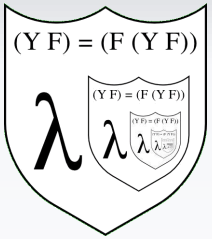
Структурные скобки

```
[Data] ::= [Letter] | [Identifier] | [MacroDigit]
         | ([Data]) | [Data] [Data]
```

Круглые скобки (не закавыченные) — дополнительный конструктор в полугруппе с конкатенацией.

- Имитация многоместных функций.
- Более общо — моделирование древовидных структур.

```
[Tree] ::= (Node [Info] (Children [Tree]+)) | (Leaf [Info])
```

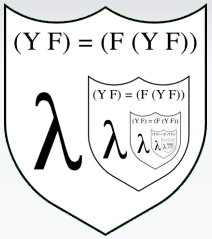


Образцы в Рефале

Образец — это строка в объединённом алфавите констант и переменных. Понятие языка образца (слов, которые могут сопоставиться с образцом) появилось только в 80-е годы, т.е. теория рефал-данных — одна из самых «молодых» в мире ЯП (и появилась она не в связи с рефалом, а в связи с вопросами машинного обучения).

Далее предполагаем все образцы «ленивыми» (переменные, начиная от самой левой, принимают как можно более короткие значения).

- Образец, находящий две одинаковые закавыченные последовательности?
- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?



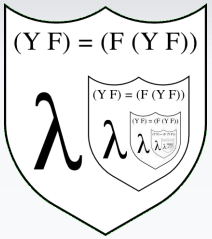
Образцы в Рефале

- Образец, находящий две одинаковые закавыченные последовательности?

  `x1 " x2 " x3 " x2 " x4`

  Заметим, что если внутри значения `x2` окажутся хотя бы одни кавычки, то существует более «ленивое» сопоставление (с той же длиной значения `x1`, но меньшей — `x2`).

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?



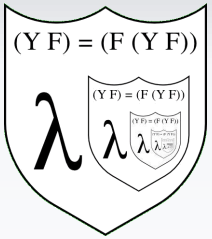
Образцы в Рефале

- Образец, находящий две одинаковые закавыченные последовательности?

  `x1 " x2 " x3 " x2 " x4`

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?

  Первая проба: `x1 x2 x2 x3`



Образцы в Рефале

- Образец, находящий две одинаковые закавыченные последовательности?

  `x1` `" x2 "` `x3` `" x2 "` `x4`

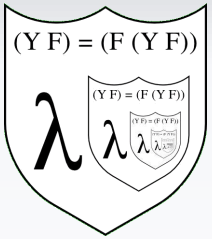
- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?

  Первая проба: `x1` `x2` `x2` `x3`

  Fail! Тривиально, из-за наличия в моноиде единицы — пустого слова. Требуется подчеркнуть непустоту — сказать, что в квадрат входит хотя бы один символ.

  Разделим переменные образца на два класса:

  - `e.name` — expression, сопоставляется с чем угодно;
  - `s.name` — symbol, сопоставляется только с символом.



Образцы в Рефале

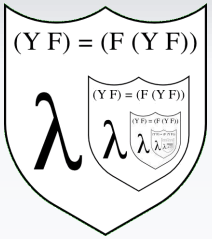
- Образец, находящий две одинаковые закавыченные последовательности?

  `x1 " x2 " x3 " x2 " x4`

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?
  Разделим переменные образца на два класса:
  - `e.name` — expression, сопоставляется с чем угодно;
  - `s.name` — symbol, сопоставляется только с символом.

  Тогда решение задачи выглядит так:

  `e.x1 s.y e.x2 s.y e.x2 e.x3`

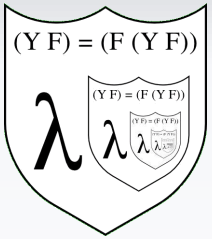


Функции над образцами

Когда спроектирован образец, построить соответствующую функцию совсем просто. В правых частях можно использовать любые переменные образца, в любом количестве.

- Функция, находящая в аргументе две одинаковые закавыченные последовательности, и возвращающая одну из них?

- Функция, находящая в слове квадрат (т.е. дважды повторяющееся подслово)?



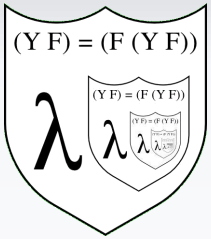
Функции над образцами

- Функция, находящая в аргументе две одинаковые закавыченные последовательности, и возвращающая одну из них?

```
F {e.x1 '\"' e.x2 '\"' e.x3 '\"' e.x2 '\"'
e.x4 = e.x2;}
```

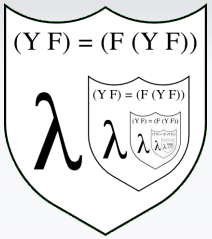
- Функция, находящая в слове квадрат (т.е. дважды повторяющееся подслово)?

```
F {e.x1 s.y e.x2 s.y e.x2 e.x3 = s.y e.x2;}
```



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

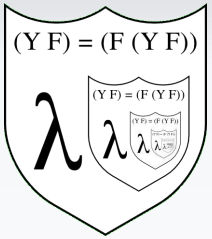


Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Решение «в лоб»:

```
F { 0 0 0 = 0 0;
    0 0 1 = 0 1;
    0 1 0 = 0 1;
    0 1 1 = 1 0;
    1 0 0 = 0 1;
    1 0 1 = 1 0;
    1 1 0 = 1 0;
    1 1 1 = 1 1; }
```



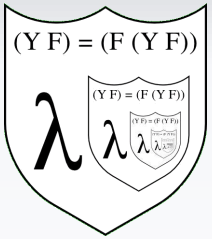
Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Решение «в лоб»:

```
F { 0 0 0 = 0 0;
    0 0 1 = 0 1;
    0 1 0 = 0 1;
    0 1 1 = 1 0;
    1 0 0 = 0 1;
    1 0 1 = 1 0;
    1 1 0 = 1 0;
    1 1 1 = 1 1; }
```

- Два нуля влекут результат `0 1`;
- Две единицы влекут результат `1 0`;
- Осталось разобраться с `t t t`. Но такие данные порождают всегда `t t`.

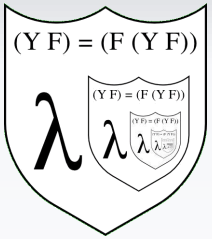


Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Второе приближение:

```
F { t t t = t t;
    x1 0 x2 0 x3 = 0 1;
    x1 1 x2 1 x3 = 1 0;
  }
```



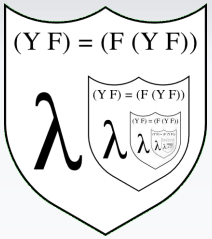
Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Второе приближение:

```
F { t t t = t t;
    x1 0 x2 0 x3 = 0 1;
    x1 1 x2 1 x3 = 1 0;
  }
```

- Видно, что вторая и третья строчки почти одинаковы — на первом месте в результате стоит выделенное значение, на втором — другое;
- Длина строки `x1 x2 x3` всегда равна 1, и это именно то другое значение!

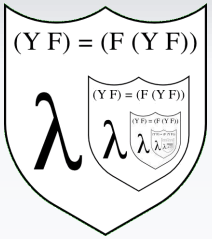


Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Почти решение:

```
F { t t t = t t;
    x1 t x2 t x3
      = t x1 x2 x3;
  }
```



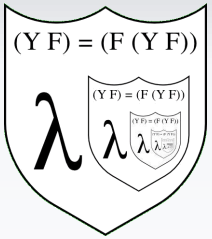
Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Почти решение:

```
F { t t t = t t;
    x1 t x2 t x3
      = t x1 x2 x3;
  }
```

- А теперь видно, что и первая строчка не нужна — в ней делается то же, что и во второй.
- Итоговая функция содержит всего одну строчку:
  `x1 t x2 t x3 = t x1 x2 x3`.
  Причём `xi` справа от знака `=` можно располагать как угодно относительно друг друга, ответ всё равно будет правильным.

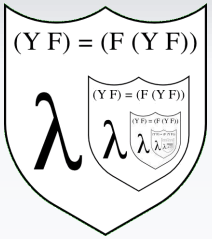


Задача Корлюкова ++

Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.



Задача Корлюкова ++

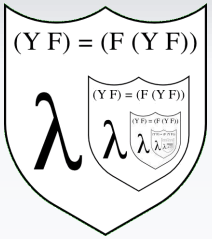
Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.

Как описать образцы для элементов образцов?

[Образец](, [Выражение] : [Образец])*



Задача Корлюкова ++

Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

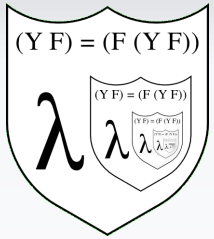
Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.

Как описать образцы для элементов образцов?

[Образец](, [Выражение] : [Образец])*

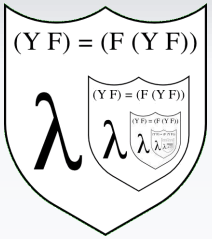
Решение задачи Корлюкова++ в этих терминах:

```
F {1 1 1 1 = 1 0 0;
   x1 0 x2
   , x1 x2 : z1 t z2 t z3 = 0 t z1 z2 z3; }
```



Другие образцы

- Слово содержит как букву ’A’, так и букву ’B’.
- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).
- Слово не содержит ничего, кроме (произвольного числа) букв ’A’.
- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

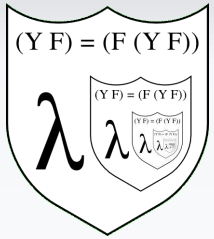
- Слово содержит как букву ’A’, так и букву ’B’.

  Ответ: образец `x1 'A' x2, x1 x2: z1 'B' z2`
  (поскольку ’A’ точно не содержит ничего от ’B’)

- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).

- Слово не содержит ничего, кроме (произвольного числа) букв ’A’.

- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

- Слово содержит как букву ’A’, так и букву ’B’.

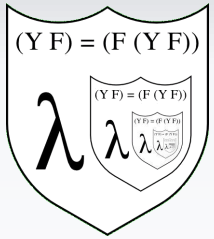
  Ответ: образец `x1 'A' x2, x1 x2: z1 'B' z2`
  (поскольку ’A’ точно не содержит ничего от ’B’)

- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).

  Тут решение тривиальное: `(x1 x2) (x2 x1)`.

- Слово не содержит ничего, кроме (произвольного числа) букв ’A’.

- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

- Слово содержит как букву ’A’, так и букву ’B’.

  Ответ: образец `x1 ’A’ x2, x1 x2: z1 ’B’ z2`
  (поскольку ’A’ точно не содержит ничего от ’B’)

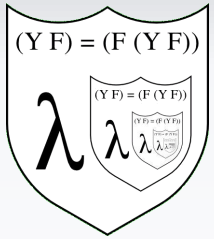
- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).

  Тут решение тривиальное: `(x1 x2) (x2 x1)`.

- Слово не содержит ничего, кроме (произвольного числа) букв ’A’.

  Решение нетривиальное: `x, ’A’ x : x ’A’`.

- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

- Слово содержит как букву ’A’, так и букву ’B’.

  Ответ: образец `x1 ’A’ x2, x1 x2: z1 ’B’ z2`
  (поскольку ’A’ точно не содержит ничего от ’B’)

- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).

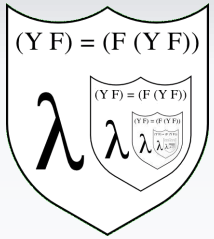
  Тут решение тривиальное: `(x1 x2) (x2 x1)`.

- Слово не содержит ничего, кроме (произвольного числа) букв ’A’.

  Решение нетривиальное: `x, ’A’ x : x ’A’`.

- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).

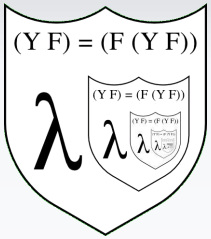
  По аналогии с предыдущим: `(x1) (x2), x1 x2 : x2 x1`.



Ещё несколько задач

Какие языки описываются следующими образцами?

- `x1, x1 A B : B A x1, x1 : x2 x2`
- `x1, x1 x1: t1 x2 x1 x3, x1: t1 x2 t2 x4`



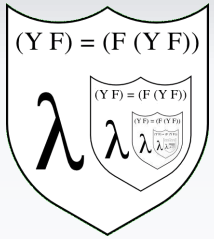
Ещё несколько задач

Какие языки описываются следующими образцами?

- `x1, x1 A B : B A x1, x1 : x2 x2`

  Пустой язык. Поскольку уравнение (а это именно уравнение) `x1 A B = B A x1` имеет решения вида `B (A B)*`, а они все — нечётной длины.

- `x1, x1 x1: t1 x2 x1 x3, x1: t1 x2 t2 x4`



Ещё несколько задач

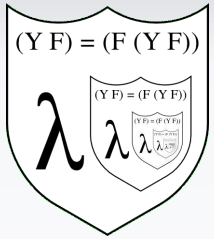
Какие языки описываются следующими образцами?

- `x1, x1 A B : B A x1, x1 : x2 x2`

  Пустой язык. Поскольку уравнение (а это именно уравнение) `x1 A B = B A x1` имеет решения вида `B (A B)*`, а они все — нечётной длины.

- `x1, x1 x1: t1 x2 x1 x3, x1: t1 x2 t2 x4`

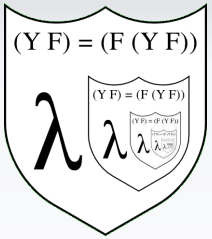
  Язык слов вида w^s, где $s \geqslant 2$. Почему они подходят, понять легко, а вот почему не подходят другие слова, проще понять, изучив основы комбинаторики слов.



Стандартные Рефал-образцы

- Три типа переменных: `e`-переменные (типа выражение), `t`-переменные (типа терм: буква или выражение в скобках), `s`-переменные (типа буква).
- Константы без кавычек — идентификаторы (например, `AAA`).
- Константы в одинарных кавычках — строки (например, `'AAA'`).

```
F {
 e.1 '(' e.2 ')' e.3
 , <G e.2> : True = <F e.1 e.3>;
 e.1 s.x e.2
 , '()' : e.z1 s.x e.z2 = UNBALANCE;
 e.Z = BALANCE;  }
G {
  e.z1 '(' e.z2 = False;
  e.x = True;   }
```

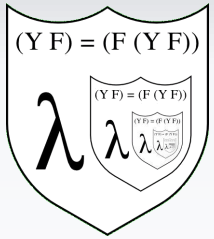


Идентификаторы и строки

Преобразования типов

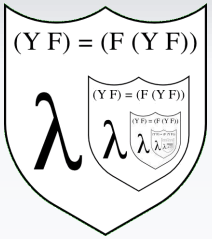
- Идентификатор в строку: `<Explode [Identifier]>`;
- Строку в идентификатор: `<Implode [String]>`;
- Макроцифру в строку: `<Symb [Number]>`;
- Строку в макроцифру: `<Numb [String]>`.

- Идентификатор и макроцифра всегда сопоставляются с единственной s-переменной. То есть, `True : s.x` и `1001 : s.x` — успешно, `'True' : s.x` — неудачно.
- С учётом `Implode`, идентификаторы образуют бесконечный алфавит констант.
- Это позволяет удобно использовать идентификаторы в качестве управляющих конструкций без использования «лишних» структурных скобок.



Пример: переименовка Рефал-функции

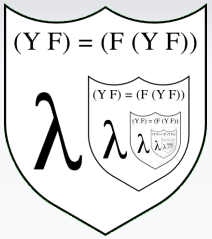
```
Rename {
  ((e.Name) e.NewName) e.1 e.Name e.2'{'e.3
  , <Meaningless e.1 e.2> : True
     = e.1 e.NewName e.2'{'
       <Rename ((e.Name) e.NewName) e.3>;
  ((e.Name) e.NewName) e.1 '<'e.Name' 'e.2
     = e.1'<'e.NewName' '
       <Rename ((e.Name) e.NewName) e.2>;
  (e.Names) e.Other = e.Other;   }
Meaningless {
  ' 'e.x = <Meaningless e.x>;
  e.x' ' = <Meaningless e.x>;
  /* ПУСТО */ = True;
  '/*'e.x'*/'e.xx = <Meaningless e.xx>;
  e.z = False;   }
```



Ввод & Вывод

```
[Открытие потока] ::= <Open [Mode] [Number] [FileName]>
[Закрытие потока] ::= <Close [Number]>
[Чтение (построчное)] ::= <Get [Number]>
[Запись (построчная с выгрузкой)]
        ::= <Putout [Number] [Expression]>
[Запись (построчная без выгрузки)]
        ::= <Put [Number] [Expression]>
```

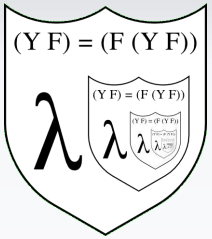
- Поток с номером 0 всегда открыт — это консоль.
- При чтении результат — строка символов.
- При записи преобразование всех термов в символы происходит автоматически.



Модули

- Объявление функции, видимой из другого модуля — `$ENTRY [FName];`
- Импорт функции — `$EXTERN [Fname].`

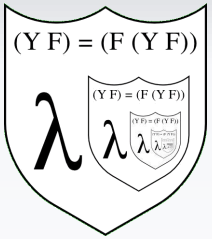
Если в нескольких модулях обнаружатся `ENTRY` — функции с одинаковыми именами, возникнет ошибка сборки.



Блоки

```
F1 {
   e.1 'B' e.2, e.1'A' : 'A' e.1 = 'Ok';
   e.Z = 'Fail'; }
F2 {
   e.1 'B' e.2
   , e.1'A' : { 'A' e.1 = 'Ok';
                e.Z = 'Fail';
              };
  e.Z = 'Fail'; }
```

Если в слове несколько вхождений буквы `'B'`, то функция `F1` будет перебирать их все, несмотря на то, что в принципе может подойти только первое из них. Блоки дают не только возможность построить локальное определение, но и возможность управлять возвратами, т.к. при заходе в блок точка возврата во внешнюю функцию удаляется.

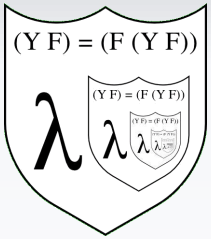


Синтаксис блоков

```
[Sentence] ::= [Pattern] (, [Expr] : [Pattern])*
               ((, [Expr] : {[Sentence]+};) | (= [Expr];))
```

Здесь круглые скобки — метасимволы, фигурные скобки — символы языка.

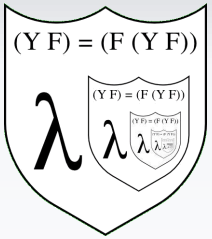
Предложение всегда заканчивается `;`, но это может произойти либо при вызове блока, либо при переходе к правой части. Блоки могут быть вложенными.



Дополнительные стеки

- Положить в стек: `<Br [String]'='[Expression]>`;
- Достать из стека: `<Dg [String]>`;
- Достать копию: `<Cp [String]>`.

Имена стеков можно порождать и вычислять как обычные строки.

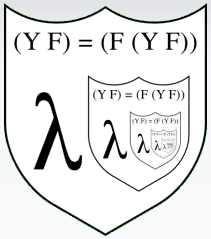


Сложность сопоставления

```
F1 {
   e.1 'A' e.2, e.2 : e.3 'B' e.4 = True;
   e.1 = False;  }
F2 {
   e.1 'A' e.2
    , e.1 e.2 : {e.3 'B' e.4 = True;
       e.Z = False;  };
   e.Z = False;  }
F3 {
   Init 'A' e.1 = <F3 Pass e.1>;
   Pass 'B' e.1 = True;
   s.Mode t.x e.1 = <F3 s.Mode e.1>;
   s.Mode e.Z = False;   }
```

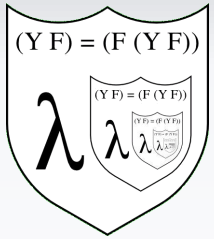
Эффективность F2 выше, чем F1 — нет удлинения переменной e.1.

Функция F3 менее всего зависит от внутренних свойств Рефал-машины (сопоставление всегда линейно), но несколько хуже отражает семантику, чем F2.



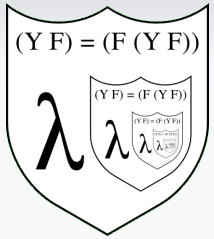
Метапрограммирование на Рефале

Вызов `<Mu [Identifier] [Expression]>` осуществляет вызов функции с именем `[Identifier]` и аргументом `[Expression]`. Функция с именем `[Identifier]` должна быть либо определена в том же модуле, либо определена в прилинкованном к нему модуле как `$ENTRY`.



Проектирование образцов

- Открытые переменные — подлежащие удлинению. Стараемся не допускать больше двух открытых переменных в образце.
- Если пустое сопоставление с переменной нас не устраивает, нужно это учесть, иначе оно обязательно когда-нибудь возникнет.
- Если некоторое обобщение образца определяет подмножество правил, которые могут быть применены к данным, то используем это обобщение для захода в блок и внутри блока уже выбираем конкретное правило.



Общие принципы проектирования

- Новая структура вложенных слов (лесов) $\Rightarrow$ вводим новую функцию.
- Другой способ обработки той же самой структуры $\Rightarrow$ удобно использовать управляющие идентификаторы внутри той же функции.
- Объект «разбирается» образцом, но используется в правой части как целое $\Rightarrow$ используем условия или блоки как безвозвратные at-конструкции.
- Порождается новый объект, который передаётся сразу нескольким функциям $\Rightarrow$ используем условия как безвозвратные let-конструкции.
- Есть редко используемые аккумуляторы $\Rightarrow$ рассматриваем возможность перемещения их в стек.