

Федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
Национальный исследовательский университет «МЭИ»

Институт автоматики и вычислительной техники  
Кафедра прикладной математики

Курсовой проект по дисциплине  
«Параллельные системы и параллельное программирование»  
**Руководство пользователя FRTL на многоядерных  
компьютерных системах**

Студент:  
Кобец С. Ю., А-13-09

Преподаватель:  
д.т.н., профессор Кутепов В.П.

Проверил:  
Шамаль П.Н.

## Оглавление

1. Общие сведения о языке .....	3
2. Встроенные типы данных и кортежи.....	7
3. Описание операций композиции.....	9
4. Задание функциональных уравнений .....	13
4.1. Функциональные уравнения .....	13
4.2. Функционалы .....	15
5. Задание данных при помощи конструкторов языка.....	16
5.1. Абстрактные типы данных .....	16
5.2. Перечисляемые типы данных.....	18
5.3. Параметризованные абстрактные типы данных .....	19
6. Структура FPTL-программы.....	21
7. Проведение экспериментов.....	26
7.1. Запуск программы .....	26
7.2. Критерии эффективности выполнения программы .....	27
7.3. Эффективность выполнения типичных задач .....	29
Список литературы .....	39
Приложение .....	40
А. Встроенные функции языка.....	40
В. Грамматика FPTL в БНФ .....	41
С. Листинг тестируемых программ.....	45

## 1. Общие сведения о языке

Программы на традиционных языках программирования, таких как C, Pascal, Java и т.п. состоят из последовательности модификаций значений некоторого набора переменных, который называется состоянием. Во время исполнения каждая команда изменяет состояние; следовательно, состояние проходит через некоторую конечную последовательность значений:

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_n = \sigma',$$

где  $\sigma_0$  – начальное состояние, а  $\sigma'$  – конечное. Такой стиль программирования называют императивным или процедурным.

Функциональное программирование представляет парадигму, в корне отличную от представленной выше модели. Функциональная программа представляет собой некоторое выражение, а выполнение программы означает вычисление значения этого выражения. С учетом приведенных выше обозначений, можно сказать, что финальное или любое промежуточное состояние представляет собой некоторую в математическом смысле функцию от начального состояния, т.е.  $\sigma_i = f(\sigma_0)$ ,  $i = 1, 2, \dots, n$ .

При сравнении функционального и императивного подхода к программированию можно заметить следующие свойства функциональных программ:

- Функциональные программы не используют переменные в том смысле, в котором это слово употребляется в императивном программировании. В частности в функциональных программах не используется оператор присваивания.
- Как следствие из предыдущего пункта, в функциональных программах нет циклов.
- Выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.

- Функциональные программы используют функции гораздо более замысловатыми способами. Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и даже в общем случае проводить вычисления, результатом которого будет функция.
- Вместо циклов функциональные программы широко используют рекурсивные функции.

Следует также сделать замечание относительно употребления термина «функция» в таких языках как Си, Java и т.п. В математическом смысле «функции» языка Си не являются функциями, поскольку:

- Их значение может зависеть не только от аргументов;
- Результатом их выполнения могут быть разнообразные побочные эффекты (например, изменение значений глобальных переменных)
- Два вызова одной и той же функции с одними и теми же аргументами могут привести к различным результатам.

Вместе с тем функции в функциональных программах действительно являются функциями в том смысле, в котором это понимается в математике. Соответственно, те замечания, которые были сделаны выше, к ним не применимы. Из этого следует, что вычисление любого выражения не может иметь никаких побочных эффектов, и значит, порядок вычисления его подвыражений не оказывает влияния на результат. Таким образом, функциональные программы легко поддаются распараллеливанию, поскольку отдельные компоненты выражений могут вычисляться одновременно.

Ясно, что для любой  $n$ -арной функции  $f(x_1, x_2, \dots, x_n)$ ,  $n \geq 1$ , можно одновременно вычислять значения функций, подставленных вместо ее аргументов. Условный оператор `if P(x) then f1(x) else f2(x)` также может рассматриваться как специальная тернарная функция  $F(P(x),$

$f1(x)$ ,  $f2(x)$ ), из чего следует возможность одновременного вычисления функций  $P(x)$ ,  $f1(x)$ ,  $f2(x)$ .

Функциональное параллельное программирование можно разделить:

- Основанное на *lambda*-исчислении (Lisp, Haskell, ML и т.д.)
- Основанное на операциях композиции (*FPTL*)

Язык функционального параллельного программирования *FPTL* (Functional Parallel Typified Language), в котором в отличие от функциональных языков ML, Haskell и других, основанных на *lambda*-исчислении, используются традиционные математические формы задания функций. Введенные в этом языке операции композиции функций позволяют в схемной форме отражать параллелизм в функциональной программе. В языке *FPTL* можно определять абстрактные типы данных, использовать библиотечные функции, организовывать модульные функциональные программы. Функции и данные в *FPTL* определяются в общем случае в виде систем функциональных и реляционных уравнений, а поскольку рекурсия является более мощным средством задания параллелизма по сравнению с определением функций посредством циклических конструкций, это придает языку большую выразительную силу.

Параллелизм в этом языке отражается в функциональной программе явно через используемые в ней операции композиции функций. В свою очередь же в расширении языков Haskell, ML, последовательных языках с целью задания в программе параллелизма используют специальные примитивы задания параллелизма (векторные команды, оператор `fork-join`) или программисту предоставлена возможность по тексту программы вводить комментарии, какие ее фрагменты и каким образом должны быть распараллелены на стадии компиляции (OpenMP).

В качестве примера, далее приведена программа вычисления факториала на языке Haskell:

Fact n = if n==0 then 1 else n\*Fact(n-1)

и *FPTL*:

```
Scheme Fact
{
    Fact = (id * 0).equal -> 1,
           ((id * 1).sub.Fact * id).mul;
}
```

## 2. Встроенные типы данных и кортежи

В *FPTL* объявлены четыре встроенных типа данных:

- `int` – целое 32-х битное число со знаком
- `real` – вещественное число двойной точности
- `string` – строка ASCII-символов
- `bool` – `true` либо `false`

Все операции в языке производят над *кортежами типов*, или же просто над *кортежами*. В математике кортеж – это упорядоченный конечный набор длины  $n$  (где  $n \in \mathbb{N} \cup \{0\}$ ), каждый элемент которого  $a_i$  принадлежит некоторому множеству  $D_i$ .

В *FPTL* этими множествами являются стандартные типы данных языка. Кортеж строится как декартово произведение исходных типов. Используемые типы и порядок их следования определяют сигнатуру кортежа; порядок следования объектов в создаваемом кортеже сохраняется на протяжении его времени жизни согласно заданной сигнатуре.

Например, `(“hello”, 42, true)` – это кортеж длины 3, который состоит из элементов типа `string`, `int` и `bool`.

Для интерпретации в *FPTL* данного кортежа, используется следующая запись:

`“hello” * 42 * true`

Длина кортежа строго фиксирована. Для кортежа, длина которого равна нулю, вводится множество  $D^{(0)} = \{\lambda\}$ , содержащее единственный элемент, обозначаемый  $\lambda$ , который удовлетворяет условию  $\lambda\alpha = \alpha\lambda = \alpha$  для любого кортежа  $\alpha$ .

В *FPTL* все кортежи являются линейными, и для доступа их элементам используется операция  $\pi_i^m$  – базисная операция выбора  $i$ -го аргумента из кортежа длиной  $m$ :

$$\pi_i^m(x_1, x_2, \dots, x_m) = x_i, m > 0, i = 1, 2, \dots, m,$$

$$\pi_0^m(x_1, x_2, \dots, x_m) = \lambda,$$

В коде программы эта операция осуществляется путем заключения в квадратные скобки номера выбираемого элемента. К примеру, следующие записи будут интерпретироваться таким образом:

`("hello" * 42 * true).[1] ⊢ hello`

`("hello" * 42 * true).([2] * [3]) ⊢ 42, true`

Здесь знак « $\vdash$ » означает результат выполнения предшествующей ему записи и непосредственного отношения к *FPTL* никакого не имеет. Базисные операции языка « $\cdot$ » и « $*$ », используемые в предыдущих примерах, более подробно будут рассмотрены в следующем разделе.



### 3. Описание операций композиции

В *FPTL* используются четыре простые операции композиции функций, являющиеся в некотором смысле редукцией общепринятых в математической практике способов задания функций путем разбора случаев, подстановки вместо аргументов известной функции других функций. Четыре рассматриваемых далее операции композиции функций, оператор задания функций посредством систем функциональных уравнений и “извлекаемое” из определения абстрактного типа данных множество функций-конструкторов и обратных им функций-деструкторов образуют универсальную сигнатуру, позволяющую выразить любую вычислимую функцию над рассматриваемым типом данных.

В *FPTL* заданы следующие четыре бинарных ассоциативных операции композиции функций: «•» — последовательная композиция, «\*» — конкатенация кортежей-значений функции (или параллельная композиция), «→» — условная композиция и «⊕» — объединение графиков ортогональных функций (или ортогональная композиция). Нужно отметить, что в текущей реализации *FPTL* операция ортогональной композиции не реализована. При написании программ, которые должен распознать интерпретатор языка, первым трем реализованным операциям будут советоваться следующие записи: «.», «\*», «-».

Ниже, определяя синтаксис и семантику операций композиции функций, используются символы  $\alpha, \beta, \gamma, \dots$ , возможно с индексами, для обозначения кортежей. Функцию  $f^{(m,n)}$  однозначно представляет ее график, который обозначается  $\tilde{f}$  и задается следующим образом:

$$\tilde{f} = \{(\alpha, \beta) \mid f(\alpha) = \beta\}$$

где  $f(\alpha)$  обозначает результат применения функции  $f$  к кортежу  $\alpha$ .

Следующий порядок старшинства операций композиции функции:  $\bullet$ ,  $*$ ,  $\rightarrow$ ,  $\oplus$ , убывающий слева направо, позволяет далее опускать ряд скобок в представлении функций. Необходимо заметить, что операции композиции  $\bullet$ ,  $*$ ,  $\rightarrow$  ассоциативны,  $\oplus$  – ассоциативна и коммутативна. Все операции композиции являются бинарными, и их определение дается в инфиксной форме.

1. Операция последовательной композиции ( $\bullet$ ):

$$f^{(m,n)} = f_1^{(m,k)} \bullet f_2^{(k,m)};$$

$$\tilde{f}^{(m,n)} = \{(\alpha, \beta) \mid \exists \gamma: (\alpha, \gamma) \in \tilde{f}_1^{(m,k)} \wedge (\gamma, \beta) \in \tilde{f}_2^{(k,m)}\};$$

$$f^{(m,n)}(\alpha) = f_2^{(k,m)}(f_1^{(m,k)}(\alpha)).$$

2. Операция конкатенации кортежей-значений функций ( $*$ ):

$$f^{(m,n1+n2)} = f_1^{(m,n1)} * f_2^{(m,n2)};$$

$$\tilde{f}^{(m,n1+n2)} = \{(\alpha, \beta_1\beta_2) \mid (\alpha, \beta_1) \in \tilde{f}_1^{(m,n1)} \wedge (\alpha, \beta_2) \in \tilde{f}_2^{(m,n2)}\};$$

$$f^{(m,n1+n2)}(\alpha) = f_1^{(m,n1)}(\alpha) f_2^{(m,n2)}(\alpha).$$

3. Операция условной композиции ( $\rightarrow$ ):

$$f^{(m,n)} = f_1^{(m,n1)} \rightarrow f_2^{(m,n)};$$

$$\tilde{f}^{(m,n)} = \{(\alpha, \beta) \mid (\alpha, \beta) \in \tilde{f}_2^{(m,n)} \wedge \exists \gamma: (\alpha, \gamma) \in \tilde{f}_1^{(m,n1)}\};$$

$f^{(m,n)}(\alpha) = f_2^{(m,n)}(\alpha)$ , если значение  $f_1$  определено;  $f^{(m,n)}(\alpha) = \omega$ , если  $f_1^{(m,n1)}(\alpha) = \omega$  или  $f_2^{(m,n)}(\alpha) = \omega$ ; если вычисление значения  $f_1^{(m,n1)}$  или  $f_2^{(m,n)}$  длится бесконечно долго, то вычисление значения  $f^{(m,n)}(\alpha)$  также продолжается бесконечно.

Чтобы согласовать эту условную конструкцию с общепринятой, когда  $f_1^{(m,n1)}$  – пропозициональная функция, принимающая два значения: «истина» и «ложь», положим, что значение «ложь» тождественно  $\omega$ .

4. Операция объединения графикой ортогональных функций ( $\oplus$ ):

$$f^{(m,n)} = f_1^{(m,n)} \oplus f_2^{(m,n)};$$

$$\tilde{f}^{(m,n)}(\alpha) = \tilde{f}_1^{(m,n)}(\alpha) \cup \tilde{f}_2^{(m,n)}(\alpha);$$

$f^{(m,n)}(\alpha) = f_1^{(m,n)}(\alpha)$  или  $f^{(m,n)}(\alpha) = f_2^{(m,n)}(\alpha)$  в зависимости от того, определено ли значение  $f_1^{(m,n)}(\alpha)$  или  $f_2^{(m,n)}(\alpha)$  соответственно.

Для того чтобы сохранялось свойство функциональности для  $f^{(m,n)}$  операция  $\oplus$  должна применяться только к совместным или ортогональным функциям.

Функции  $f_1^{(m,n)}$  и  $f_2^{(m,n)}$  *совместны* тогда и только тогда, когда:

$$\forall \alpha \forall \beta_1 \forall \beta_2 : (\alpha, \beta_1) \in \tilde{f}_1^{(m,n)} \wedge (\alpha, \beta_2) \in \tilde{f}_2^{(m,n)} \supset \beta_1 = \beta_2$$

Функции  $f_1$  и  $f_2$  *ортогональны*, если они совместны и не существует кортежей  $\beta_1$  и  $\beta_2$  таких, что  $(\alpha, \beta_1) \in \tilde{f}_1^{(m,n)}$  и  $(\alpha, \beta_2) \in \tilde{f}_2^{(m,n)}$  для всякого  $\alpha$ . Другими словами, функции ортогональны, если пересечение их графиков пусто.

График функции  $f^{(m,n)}$ , определенной посредством операции  $\oplus$ , теперь имеет следующую семантику:

- $f^{(m,n)}(\alpha) = f_1^{(m,n)}(\alpha)$  или  $f^{(m,n)}(\alpha) = f_2^{(m,n)}(\alpha)$  в зависимости от того, определено ли значение  $f_1^{(m,n)}(\alpha)$  или  $f_2^{(m,n)}(\alpha)$  соответственно;
- если значение  $f_1^{(m,n)}(\alpha)$  и  $f_2^{(m,n)}(\alpha)$  оба не определены,  $f^{(m,n)}(\alpha)$  также не определено;
- если значения  $f_1^{(m,n)}(\alpha)$  и  $f_2^{(m,n)}(\alpha)$  оба определены, то  $f^{(m,n)}(\alpha)$  принимает значение той функции, которая была вычислена первой.

Заметим, что первые две операции композиции позволяют выразить известный оператор подстановки, используемый в языке рекурсивных функций и в обычной математической нотации:

$$f(x_1, x_2, \dots, x_n) = g(f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)),$$

где  $g$  –  $m$ -арная, а  $f_1, f_2, \dots, f_m$  –  $n$ -арные заданные функции. Через базисные функции *FPTL* это записывается в следующем виде:

$$f = (f_1 * f_2 * \dots * f_m).g$$

Через операцию условной композиции можно легко задать широко используемую в программировании условную конструкцию

if  $p$  then  $f_1$  else  $f_2$

В *FPTL* эта запись имеет следующий вид:

$$p \rightarrow f_1, f_2$$

Для выражения  $p \rightarrow f_1$  «ложь» для  $p$  можно трактовать как неопределенное значение, т.е.  $p = \omega$ .

## 4. Задание функциональных уравнений

Основными семантическими объектами в языке *FPTL* наравне с данными выступают определяемые на них в общем случае частичные функции. В *FPTL* рассматриваются типизированные  $(m, n)$ -арные,  $m \geq 0, n \geq 0$  функции, отображающие кортежи типизированных элементов длины  $m$  в кортежи типизированных элементов длины  $n$ . Таким образом, любая  $(m, n)$ -арная функция  $f^{(m,n)}$ , имеющая тип  $t_1, t_2, \dots, t_m \rightarrow t_1', t_2', \dots, t_n'$ , есть однозначное в общем случае частичное отображение:

$$f^{(m,n)} : D_{t_1} \times D_{t_2} \times \dots \times D_{t_m} \rightarrow D_{t_1'} \times D_{t_2'} \times \dots \times D_{t_n'}$$

где  $D_i$  – непустое множество элементов типа  $t_j$ .

При этом предполагается, что  $D_i$  содержит вычисляемое неопределенное значение, обозначаемое  $\omega$ . Неопределённое значение функции в *FPTL* используется в двух смыслах: оно либо отражает неограниченный процесс вычислений, либо является вычислимым и в этом случае обозначается  $\omega$ . Поэтому значение функции не определено, если вычисление значения хотя бы одного элемента её входного кортежа длится неограниченно долго или оно вычислено и имеет значение  $\omega$ .

Функции в *FPTL* определяются в общем случае посредством систем функциональных уравнений в заданных сигнатурах, в записи которых используются описанные в предыдущем разделе операции композиции.

### 4.1. Функциональные уравнения

Общей формой задания функций в *FPTL* являются системы функциональных уравнений вида

$$x_i = \tau_i, i = 1, 2, \dots, n, \quad (1)$$

где  $x_i$  – функциональная переменная,  $\tau_i$  – терм той же арности, что и  $x_i$ .

Неявно предполагается, что каждая функциональная переменная или базисная функция имеет заданную арность.

Термом в сигнатуре операций композиции называется любая конструкция, построенная из базисных функций и функциональных переменных путём конечного числа применений операций композиции.

Множество термов строится индуктивно:

1. Функциональная переменная или базисная функция есть терм той же ариности и того же типа, что и функциональная переменная или базисная функция.
2. Если  $\tau_1$  и  $\tau_2$  – термы, то  $(\tau_1 \Delta \tau_2)$  – также термы, где  $\Delta \in \{\bullet, *, \rightarrow, \oplus\}$ . При этом требуется согласование ариностей термов  $\tau_1$  и  $\tau_2$  для применяемой операции композиции, как это было указано выше.
3. Других термов нет.

Вернемся к примеру функции вычисления факториала:

```
Scheme Fact
{
    Fact = (id * 0).equal -> 1,
           ((id * 1).sub.Fact * id).mul;
}
```

Данная функция может быть разбита на термы. В результате будет получена следующая эквивалентная запись:

```
Scheme Fact
{
    Fact = P -> F1, F2;
    P = (id * 0).equal;
    F1 = 1;
    F2 = (id * DEC.Fact).mul;
    DEC = (id * 1).sub;
}
```

Выбор описанных в этом разделе операций композиции функции, помимо требования универсальности, в большой степени был предопределен общепринятой математической практикой определения функции в виде рекурсивных равенств, в правой части которых может использоваться опера-

ция разбора случаев и подстановки (подстановки функции вместо переменных известной функции).

## 4.2. Функционалы

В *FPTL* есть возможность использования функционалов - функций, принимающих в качестве одного из параметров другую функцию. Имя функции, являющейся параметром функционала, записывается в квадратных скобках после имени функционала.

Примером функционала может служить функция численного вычисления интеграла, принимающая в качестве входного аргумента интегрируемую функцию. Пример вычисления интеграла методом трапеции на элементарном отрезке  $[a, b]$  записывается в *FPTL* следующим образом:

```
Fun Integrate[Fn]
{
    Integrate = ((([1].Fn * [2].Fn).add * ([2] *
                [1]).sub).mul * 2.0).div;
}
```

## 5. Задание данных при помощи конструкторов языка

### 5.1. Абстрактные типы данных

В *FPTL* можно представлять любой абстрактный тип данных.

Примером синонима к существующему типу данных `int` может являться следующая простая конструкция:

```
Data d
{
    d = int.c_succ;
}
```

Чуть более сложным примером может служить описание типа данных-пары из строковой и вещественной переменной:

```
Data Pair
{
    Pair = string * int.c_pair;
}
```

Как видно из примеров, типы данных, как и функции, в *FPTL* определяются через систему реляционных уравнений с использованием введенных ранее операций композиции функций и конструкторов.

Для любого определяемого абстрактного типа данных базисными функциями являются функции-конструкторы и функции-деструкторы (обратные функции), вводимые при определении абстрактного типа. Более того, используемые при его построении функции-конструкторы и функции-деструкторы вместе с функциями выбора аргумента  $\pi_i^m$ ,  $m > 0$ ,  $i = 1, 2, \dots, m$ , образуют полный набор базисных функций в том смысле, что любая вычислимая функция над данным рассматриваемого типа может быть выражена средствами языка *FPTL*. С каждым конструктором и деструктором связана их арность:  $(n_i, 1)$  и  $(1, n_i)$  соответственно.

Например, мы можем определить в *FPTL* как абстрактный тип данных множество для представления натуральных чисел NAT:



$$\text{NAT}^{(0,1)} = 0^{(0,1)} \oplus \text{NAT}^{(0,1)} \cdot \text{succ}^{(1,1)},$$

где введены конструкторы  $0^{(0,1)}$  и  $\text{succ}^{(1,1)}$  указанных арностей, такие что:

$$\tilde{0}(\lambda) = 0,$$

$$\widetilde{\text{succ}}(x) = x \cdot \text{succ}$$

Одновременно неявно вводятся сопровождающие их функции-деструкторы:  $(0^{-1})^{(0,1)}$  и  $(\text{succ}^{-1})^{(1,1)}$  такие, что:

$$\tilde{0}^{-1}(x) = \begin{cases} \lambda, & \text{если } x = 0 \\ \omega, & \text{если } x \neq 0 \end{cases}$$

$$\widetilde{\text{succ}}^{-1}(x) = \begin{cases} y, & \text{если } x = y \\ \omega, & \text{если } x \neq y \end{cases}$$

При написании программ на *FPTL* по соглашению принято, что имена конструкторов начинаются с приставки «с\_». Имя деструктора получается путем добавления «~» в начале имени конструктора.

В результате, приведенный выше пример описания типа данных NAT переписывается в приемлемом для компилятора виде следующим образом:

```
Data Nat
{
    Nat = c_null ++ Nat.c_succ;
}
```

Для описанных выше конструкторов `c_null` и `c_succ` одновременно неявно были введены деструкторы с именами `~c_null` и `~c_succ` соответственно.

Рассмотрим пример описания в *FPTL* типа данных списка целых чисел:

```
Data List
{
    List = c_nil ++ int * List.c_cons;
}
```

Для доступа к элементам такого списка необходимо использовать деструкторы. Другими словами для последовательного получения элементов списка необходимо его последовательно разрушать. К примеру, имеется следующий список.

```
a = (5 * (9 * c_nil).c_cons).c_cons
```

Тогда применение к нему деструкторов будет давать следующий результат:

```
a.~c_cons.[1] | 5
a.~c_cons.[2].~c_cons.[1] | 9
a.~c_cons -> "t", "f" | "t"
a.~c_nil -> "t", "f" | "f"
a.~c_cons.[2].~c_cons.[2].~c_nil -> "t", "f" | "t"
```

В качестве примера использования описанного типа может служить программа конкатенации двух списков:

```
Scheme Concat
{
    Concat = [1].~c_nil -> [2], [2].~c_nil -> [1],
              ([1].~c_cons.[1] * ([1].~c_cons.[2] *
                                   [2])).Concat).c_cons;
}

Application
List1 = (1 * c_nil).c_cons;
List2 = (2 * c_nil).c_cons;
% Concat(List1, List2)
```

## 5.2. Перечисляемые типы данных

Как видно из прошлого примера описания данных, в *FPTL* присутствует возможность задания перечислимых типов данных. Данный тип данных является аналогом `enum` в таких языках как Java и C#. Осуществляется эта

возможность при помощи операции ортогональной композиции. Например, такая конструкция на языке Java как

```
enum Season
{
    Winter, Spring, Summer, Autumn
}
```

будет описываться в *FPTL* следующим образом:

```
Data Season
{
    Season = c_Winter ++ c_Spring ++ c_Summer ++
            c_Autumn;
}
```

### 5.3. Параметризованные абстрактные типы данных

Создаваемые в *FPTL* типы данных могут быть параметризованы типовыми переменными. Фактически, это аналог шаблонов из C или *generics* из Java и C#.

Имена типовых параметров должны начинаться с апострофа. Их нужно указать в квадратных скобках после имени типа данных в его заголовке. В результате для того, чтобы задать параметризованный список предыдущий пример нужно модифицировать следующим образом:

```
Data List['x]
{
    List = c_nil ++ 'x * List['x].c_cons;
}
```

Тип элементов, которые будет содержать список, определяется дальше в программе (явно или неявно).

Также при описании параметризованных абстрактных типов данных можно использовать несколько параметров. Например, параметризованная пара описывается следующим образом:

```
Data TPair['x','y']  
{  
    TPair = 'x * 'y.c_pair;  
}
```

В качестве типовых параметров возможно использование и типов данных, которые сами имеют типовые параметры. Например, создание списка списков для хранения элементов матрицы.

## 6. Структура FRTL-программы

Любая *FRTL*-программа имеет следующую структуру:

- Представление данных
- Схема
  - Функциональная переменная
  - Вложенная функция
- Описание применения схемы

Сама программа находится в блоке **Scheme**. Представление функции, в котором рассматривается только композиционная ее структура с точностью до введенных операций композиции, т.е. без задания интерпретации базисных функций, называется *схемой*. Схема как самостоятельный элемент в задании функции, наглядно отражает ее композиционную структуру и упрощает ее анализ. Именно схема функции является основой построения асинхронной модели вычисления значений функций.

Схема состоит из нескольких функциональных уравнений. Все функциональные уравнения описываются с новой строки. Точкой входа в программу является функциональная переменная, с именем, совпадающим с именем схемы. Поэтому в схеме всегда должны содержать хотя бы одно объявление функциональной переменной с именем, таким же как и конструкции **Scheme**. В программе может быть использован символ “@” – он является синонимом функциональной переменной с именем, совпадающим с именем схемы.

Рассмотрим пример описания структуры программы вычисления факториала:

Scheme Fact

```
{
    @ = P -> F1, F2;
    P = (id * 0).equal;
    F1 = 1;
    F2 = (id * DEC.@).mul;
    DEC = (id * 1).sub;
}
```

В описании функциональной переменной мы можем использовать введенные ранее операции композиции, функциональные переменные, определенные в нашей схеме, а также все встроенные функции языка. В программе явно не указываются аргументы функциональных переменных. По сути, они передаются неявно. Например, аргументы функциональной переменной **Fact** будут переданы в функциональные переменные **P**, **F1**, **F2** при их вызовах из **Fact**. Получить значение аргументов можно либо при помощи функции **id**, результатом которой являются все ее входные аргументы, т.е.

$$id(\alpha) = \alpha,$$

либо при помощи функции  $\pi_i^m$ ,  $m > 0$ ,  $i = 1, 2, \dots, m$ , – выбор  $i$ -го аргумента из  $m$  входных аргументов. Т.е. поскольку описанная выше программа имеет одно входное значение, то ее можно записать следующим эквивалентным способом:

Scheme Fact

```
{
    @ = P -> F1, F2;
    P = ([1] * 0).equal;
    F1 = 1;
    F2 = ([1] * DEC.@).mul;
    DEC = ([1] * 1).sub;
}
```

Внутри блока **Scheme** могут заключаться вложенные функции, которые располагаются внутри блоков **Fun**. При необходимости, в **Scheme** может присутствовать несколько блоков **Fun**, причем каждый из которых так же

может содержать вложенные функции. Основным назначением блока `Fun` является разграничение имен функциональных переменных. Рассмотрим пример:

```
Scheme MyScheme
{
    MyScheme = 1;
    F = 2;

    Fun MyFun1
    {
        MyFun1 = 1;
        F = 3;
    }

    Fun MyFun2
    {
        MyFun2 = 2;
        F = 4;
    }
}
```

В данной схеме каждая из функциональных переменных `F` имеет область видимости только в пределах своего блока.

В структуре программы схеме предшествует описание данных, которое заключается в блок `Data`. Одним из уже рассмотренных ранее примеров описания данных может служить описание параметризованного списка:

```
Data List['x]
{
    List = c_nil ++ 'x * List['x].c_cons;
}
```

Нужно отметить, что в данном блоке может быть несколько описаний данных подряд.

После блока `Scheme` следует описание применения схемы, где описывается начальный набор данных, к которым будет применяться схема. Вызов

программы, описанной в схеме, производится из этой секции. Данный раздел имеет следующий синтаксис:

```
Application
/* Блок констант */
% Имя_схемы(параметры)
```

Примером описания применения схемы программы, использующей в качестве входного значения список, служит следующая запись:

```
Application
a = (5 * (9 * (2 * (7 * (3 * (4 * (1 * (6 *
c_nil).c_cons).c_cons).c_cons).c_cons).c_cons).c_cons).c_cons);
% Sort(a)
```

Ниже приведен пример описания применения схемы вычисления факториала:

```
Application
% Fact(7)
```

Нужно заметить, что результат выполнения этой программы не будет выведен на экран. Для вывода на экран результатов необходимо произвести модификацию программы при помощи встроенной функции вывода в консоль `print`. В результате текст программы вместе с описанием ее вызова будет иметь следующий вид:

```
Scheme PrintFact
{
    PrintFact = id.Fact.print;
    Fact = P -> F1, F2;
    P = ([1] * 0).equal;
    F1 = 1;
    F2 = ([1] * DEC.Fact).mul;
    DEC = ([1] * 1).sub;
}

Application
% PrintFact(7)
```



Далее описано то, как меняется аргумент функциональных переменных в процессе выполнения данной функциональной программы.

В `PrintFact` передается одно целое число типа `int`, которое тут же передается в `Fact`, а оттуда в `P`, `F1` и `F2`. В `P` берется аргумент функциональной переменной (`[1]` возвращает целое число) и приписывается к нему справа еще одно целое число – константа ноль. Этот кортеж из двух чисел становится аргументом встроенной функции `equal` (проверка на равенство). Результат функции `equal` (булевское значение) становится результатом функциональной переменной `P`.

В `F1` не используется входной параметр, а просто возвращается целочисленная константа как результат функции. Целое число будет и типом результата функциональной переменной `Fact`, т.к. тип условного оператора определяется типом любой из ветвей (эти типы должны совпадать), а результатом одной из ветвей является результат функции `F1`.

В `F2` берется входной параметр (целое число) и к нему справа приписывается результат вызова функций `DEC.Fact`. Результатом функциональной переменной `DEC` будет целое число (см. ниже). Оно передается на вход функциональной переменной `Fact`. На выходе из нее также получается целое число (см. выше).

Соответственно, результатом вызова пары `DEC.Fact` будет одно целое число. Оно приписывается справа к аргументу функции `F2`, и полученная пара целых чисел передается на вход функции `mul` (умножение). Ее результат (целое число) будет являться результатом функциональной переменной `F2`.

В `DEC` берется аргумент функциональной переменной (целое число), приписывается к нему справа целочисленная константа `1`, и полученная пара чисел передается на вход функции `sub` (вычитание). Полученное целое число будет результатом функциональной переменной.

Результат функции `Fact` (целое число) передается на вход встроенной функции `print`, которая выведет полученное значение на экран.

## 7. Проведение экспериментов

### 7.1. Запуск программы

Для того, чтобы выполнить программу на своем компьютере пользователю необходимо выполнить следующие действия:

1. Поместить текст *FPTL*-программы в текстовый файл в пользовательской директории компьютера.
2. Перейти в директорию с исполняемым файлом интерпретатора *FPTL* с помощью команды консоли:

```
>cd /d путь_к_файлу_интерпретатора
```

3. Запустить выполнение своей программы с помощью команды консоли:

```
>fptl путь_к_файлу_программы/имя_файла_с_расширением N
```

где `fptl` – имя файла интерпретатора, *N* – число используемых при вычислении нитей.

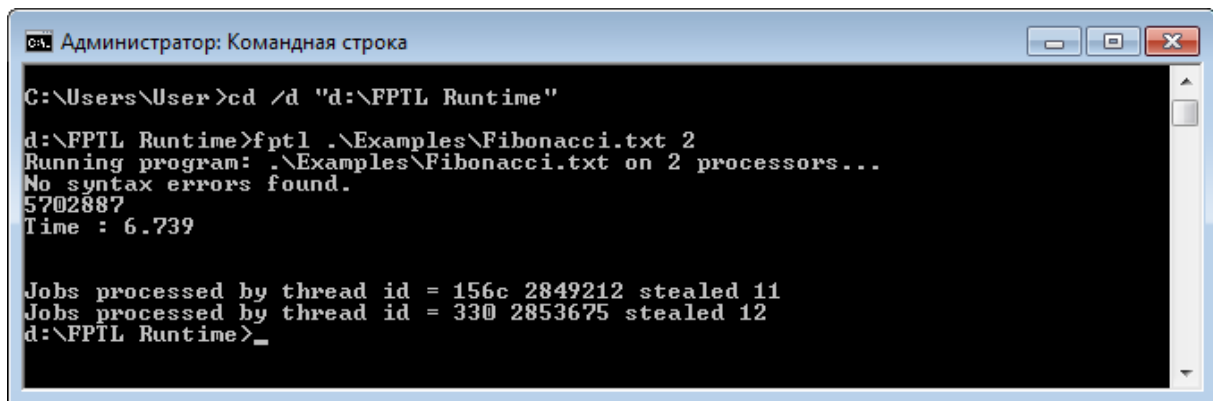
Рассмотрим пример программы вычисления *N*-го числа Фибоначчи:

```
Scheme Fibonacci
{
    @ = Fib.print;

    Fib = ([1] * 0).equal -> 1,
          ([1] * 1).equal -> 1,
          ((([1] * 2).sub.Fib * ([1] * 1).sub.Fib).add);
}

Application
% Fibonacci(33)
```

Текст данной программы находится в текстовом файле `Fibonacci.txt`, зададим ее выполнение на двух нитях:



```
Администратор: Командная строка

C:\Users\User>cd /d "d:\FPTL Runtime"

d:\FPTL Runtime>fptl .\Examples\Fibonacci.txt 2
Running program: .\Examples\Fibonacci.txt on 2 processors...
No syntax errors found.
5702887
Time : 6.739

Jobs processed by thread id = 156c 2849212 stealed 11
Jobs processed by thread id = 330 2853675 stealed 12
d:\FPTL Runtime>_
```

Как видно, в результате выполнения программы получен ответ равный 5702887, а также выведено время выполнения программы, равное 6,739с.

## 7.2. Критерии эффективности выполнения программы

При выполнении параллельной программы прежде всего необходимо измерять время ее выполнения. Пусть  $T_1(n)$  – время выполнения наилучшего последовательного алгоритма решения некоторой задачи, а  $T_p(n)$  – время выполнения параллельного алгоритма решения этой задачи на  $p$  исполнителях, где  $n$  – размерность задачи. Тогда ускорение для данного параллельного алгоритма рассчитывается по следующей формуле:

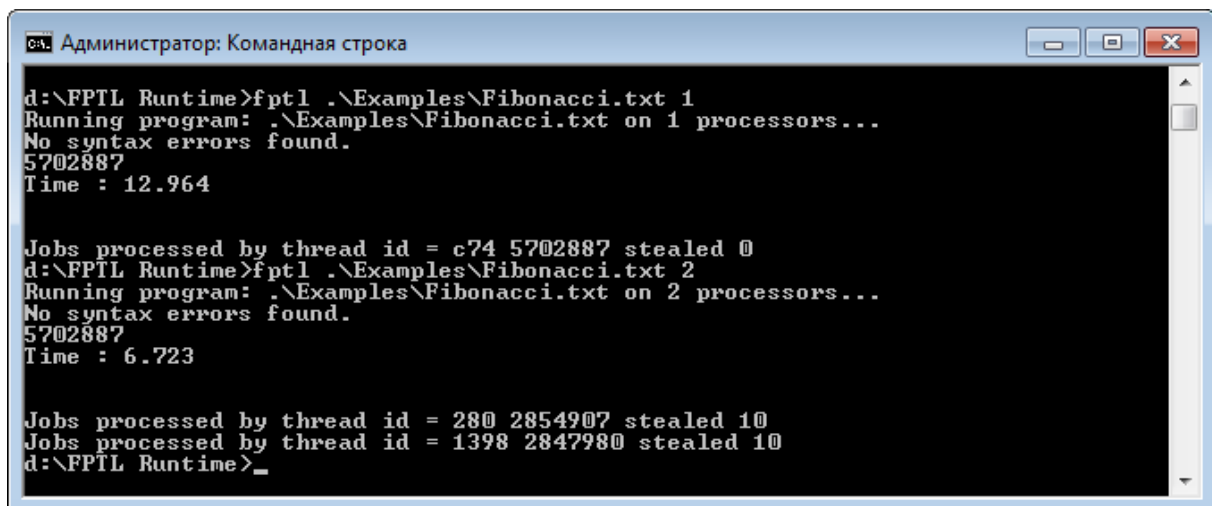
$$\xi = T_1(n) / T_p(n)$$

Чем ближе значение ускорения к  $p$ , тем лучше параллельный алгоритм и, соответственно, программа. И потому данный параметр является основным показателем эффективности параллельной программы.

Язык *FPTL* создавался с целью формализации общепринятой математической нотации задания функций и одновременно явного отражения параллелизма вычислений в этом задании на основе свойств операций композиции функций. С вычислительной точки зрения только для операции последовательной композиции требуется строго последовательное вычисление значений функций, к которым она применена. Для операций параллельной и условной композиции значения функций могут вычисляться в любом порядке, в том числе параллельно.

Интересно провести параллель между понятием информационной независимости компонентов программы и его аналогией в *FPTL*. Очевидно, что только операция последовательной композиции отражает зависимость по данным между функциями, к которым она применяется. Остальные операции говорят об информационной независимости функций, которые участвуют в композициях, определяемых посредством этих операций. Как естественное следствие этого, значения этих функций можно вычислять параллельно, а операции параллельной и условной композиции являются основным источником параллелизма *FPTL*-программ

Рассмотрим пример выполнения приведенной ранее программы вычисления 33-го числа Фибоначчи. При выполнении данной программы на одном и двух потоках получены следующие результаты:



```
Администратор: Командная строка

d:\FPTL Runtime>fptl .\Examples\Fibonacci.txt 1
Running program: .\Examples\Fibonacci.txt on 1 processors...
No syntax errors found.
5702887
Time : 12.964

Jobs processed by thread id = c74 5702887 stealed 0
d:\FPTL Runtime>fptl .\Examples\Fibonacci.txt 2
Running program: .\Examples\Fibonacci.txt on 2 processors...
No syntax errors found.
5702887
Time : 6.723

Jobs processed by thread id = 280 2854907 stealed 10
Jobs processed by thread id = 1398 2847980 stealed 10
d:\FPTL Runtime>_
```

Получаем следующее значение ускорения при выполнении на двух потоках:

$$\xi = T_1(33) / T_2(33) = 12,964 / 6,723 = 1,928$$

Значение ускорения очень близко к значению количества исполнителей, а значит данный алгоритм хорошо распараллеливает выполнение задачи на данном числе потоков.

### 7.3. Эффективность выполнения типичных задач

Все тесты выполнялись на многопроцессорной системе со следующей конфигурацией:

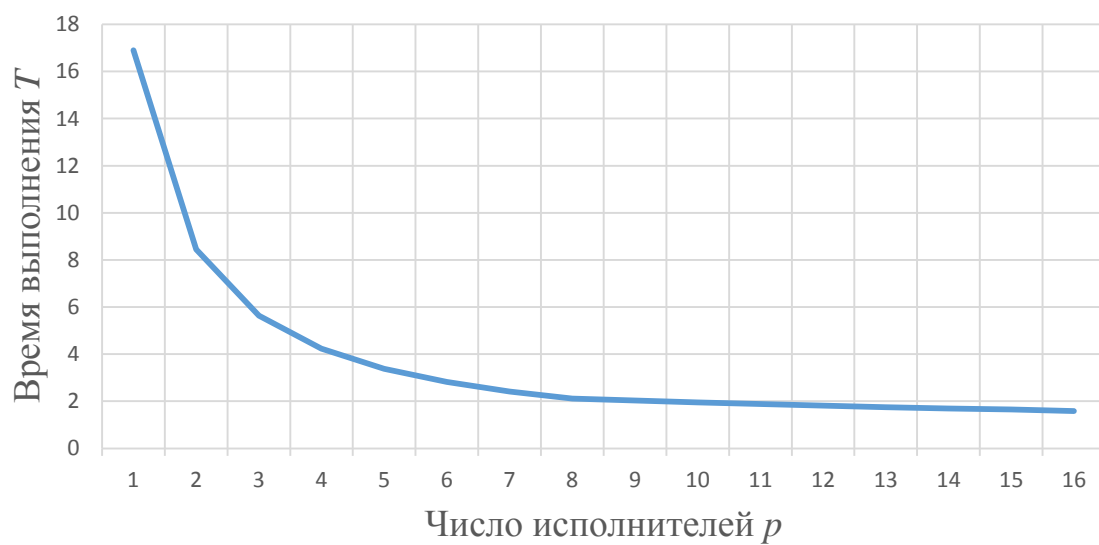
- 8×2 ЦПУ 2.8ГГц
- 14Гб ОЗУ
- Стек рекурсии 100мб
- ОС Ubuntu 12

#### Численное интегрирование

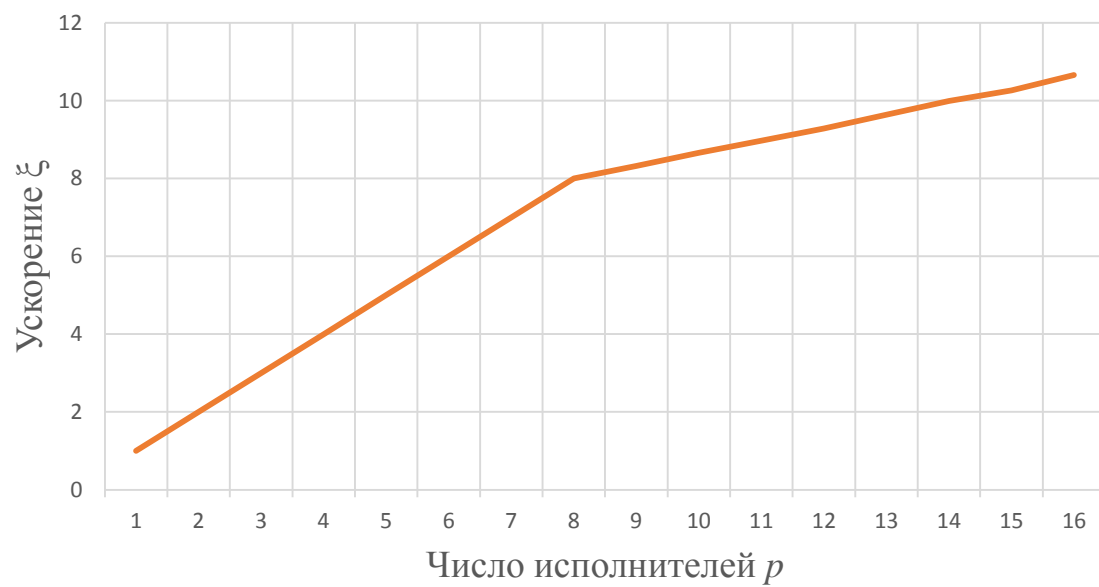
Интегрирование методом трапеций на отрезке  $[0, 10]$  с точностью  $\varepsilon = 10^{-6}$  функции  $f(x) = \sin(0,5 \cdot x^3) \cdot \sin(0,25 \cdot x^2) \cdot \sin(0,125 \cdot x)$ :

Число исполнителей $p$	Время выполнения $T$	Ускорение $\xi$
1	16,900	1,000
2	8,450	2,000
3	5,630	3,002
4	4,230	3,995
5	3,377	5,004
6	2,815	6,004
7	2,415	6,998
8	2,113	7,998
9	2,030	8,325
10	1,952	8,658
11	1,883	8,975
12	1,821	9,281
13	1,753	9,641
14	1,691	9,994
15	1,646	10,267
16	1,586	10,656

**Зависимость времени выполнения от числа исполнителей**



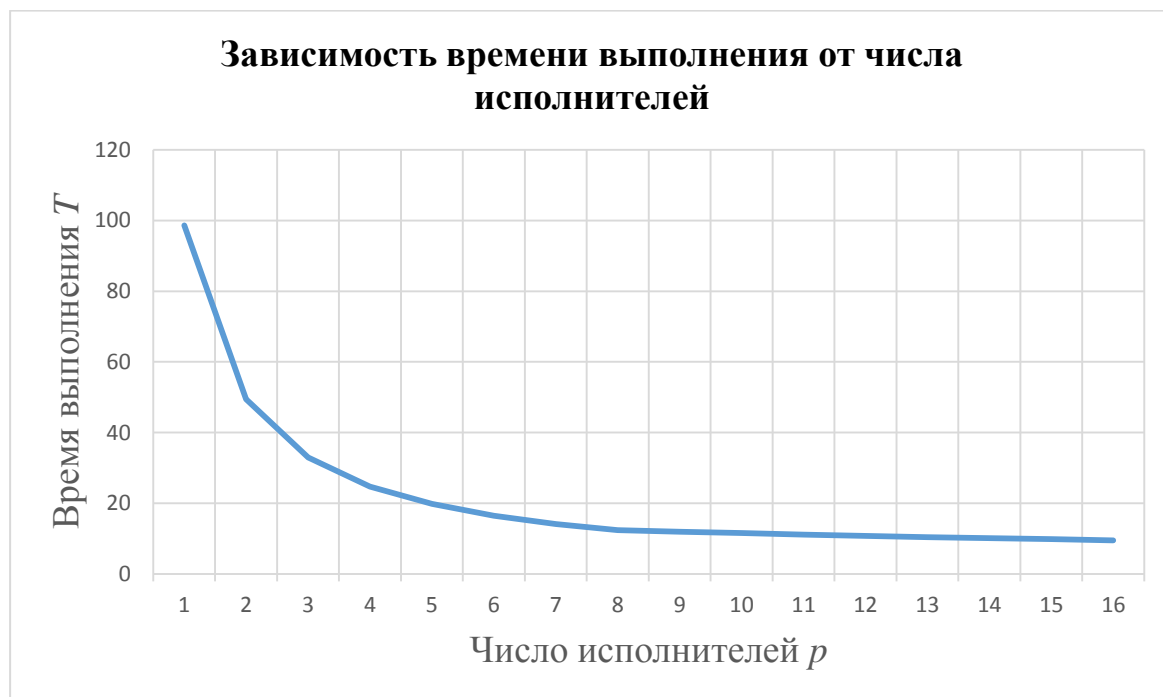
**Зависимость ускорения от числа исполнителей**

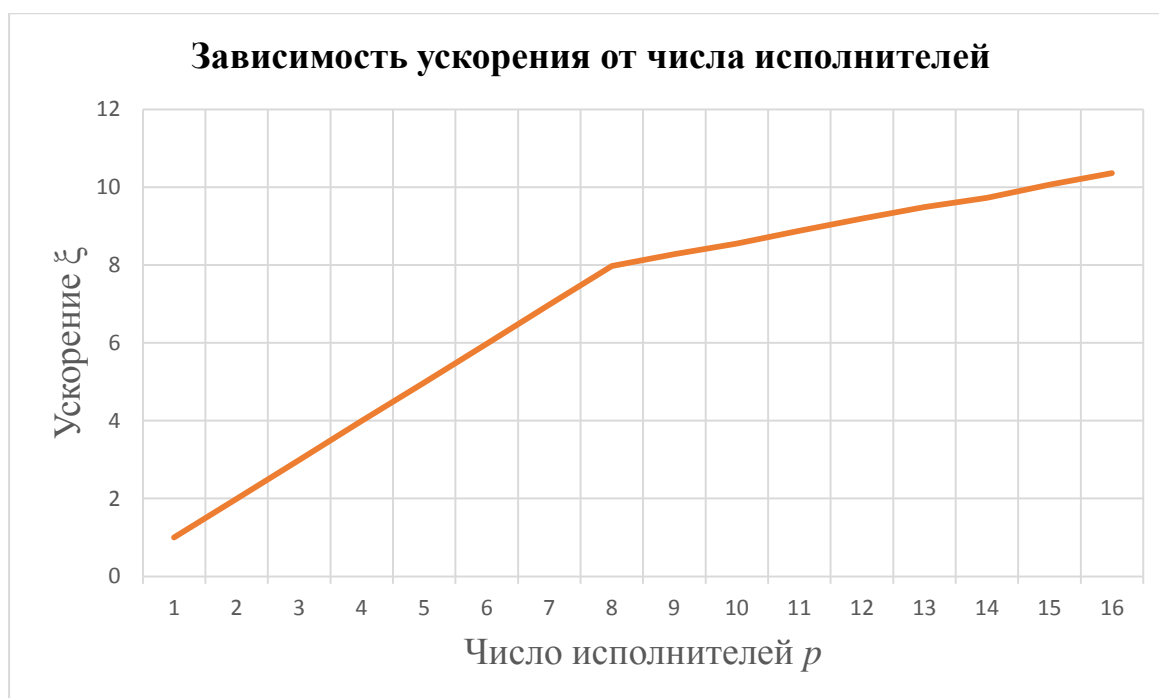


## Нахождение числа Фибоначчи

Вычисление  $n = 37$  числа Фибоначчи:

Число исполнителей $p$	Время выполнения $T$	Ускорение $\xi$
1	98,606	1,000
2	49,447	1,994
3	32,938	2,994
4	24,683	3,995
5	19,789	4,983
6	16,489	5,980
7	14,127	6,980
8	12,362	7,977
9	11,912	8,278
10	11,526	8,555
11	11,099	8,884
12	10,726	9,193
13	10,387	9,493
14	10,132	9,732
15	9,796	10,066
16	9,513	10,365





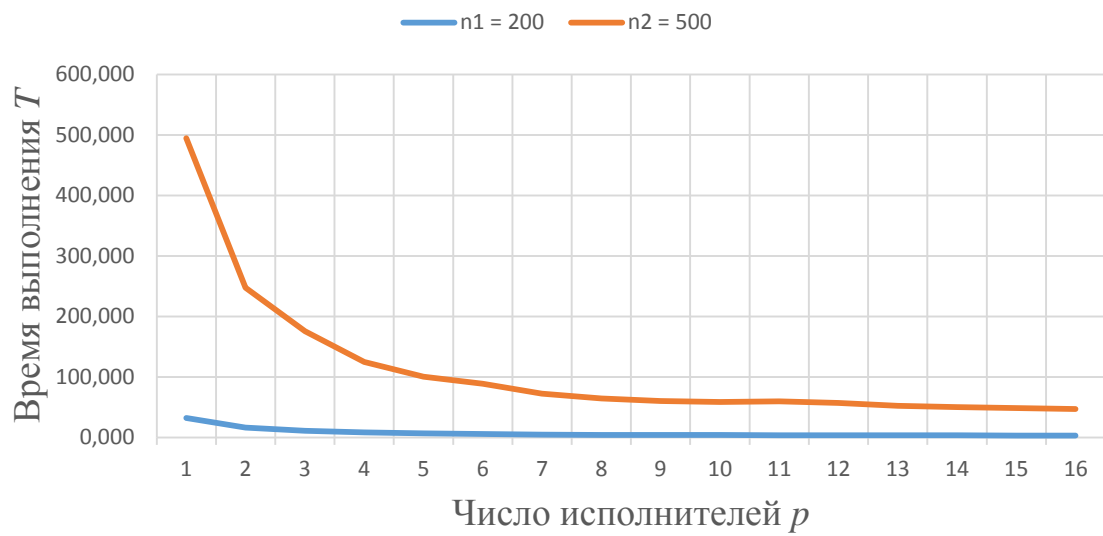
### Умножение матриц

Умножение матриц, представленных двумерными массивами, для размерностей  $n_1 = 200$  и  $n_2 = 500$ :

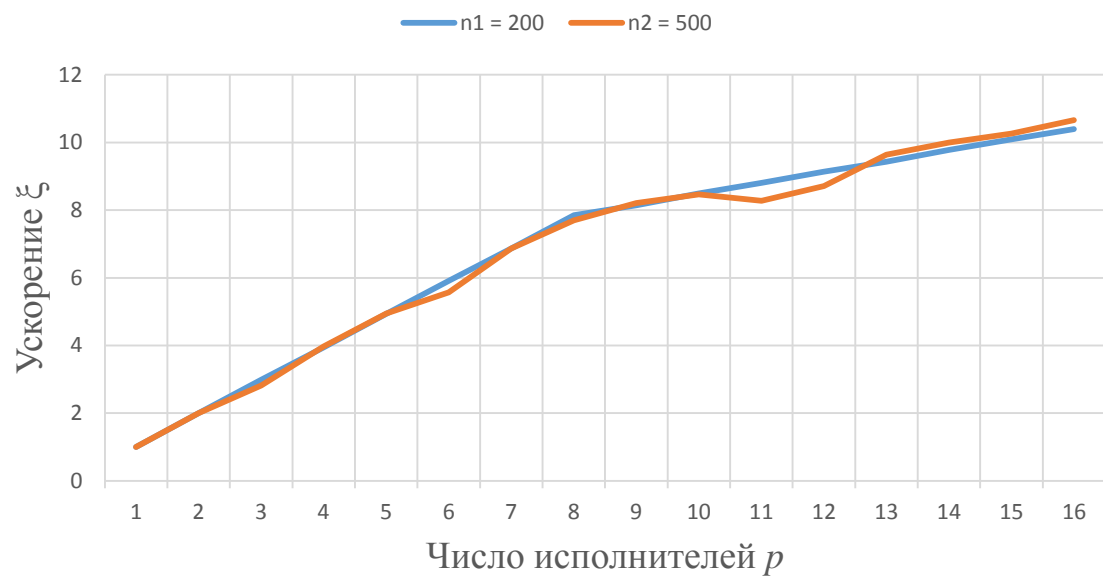
Число исполнителей $p$	Время выполнения $T_1$	Время выполнения $T_2$	Ускорение $\xi_1$	Ускорение $\xi_2$
1	32,256	494,732	1,000	1,000
2	16,128	247,287	2,000	2,001
3	10,808	175,581	2,984	2,818
4	8,183	124,759	3,942	3,966
5	6,535	100,061	4,936	4,944
6	5,458	88,808	5,910	5,571
7	4,701	72,069	6,862	6,865
8	4,110	64,241	7,848	7,701
9	3,962	60,236	8,141	8,213
10	3,799	58,454	8,491	8,464
11	3,663	59,762	8,806	8,278
12	3,531	56,793	9,135	8,711
13	3,419	52,196	9,434	9,641
14	3,298	50,119	9,780	9,994
15	3,196	48,485	10,093	10,267
16	3,104	46,949	10,392	10,656



### Зависимость времени выполнения от числа исполнителей

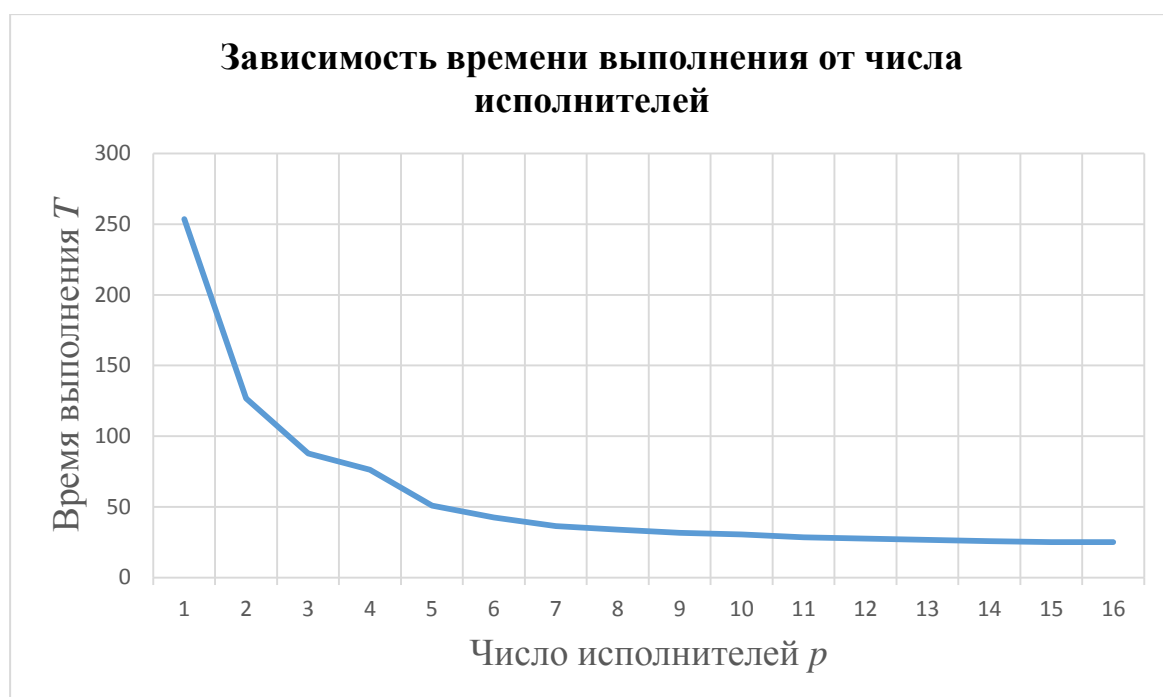


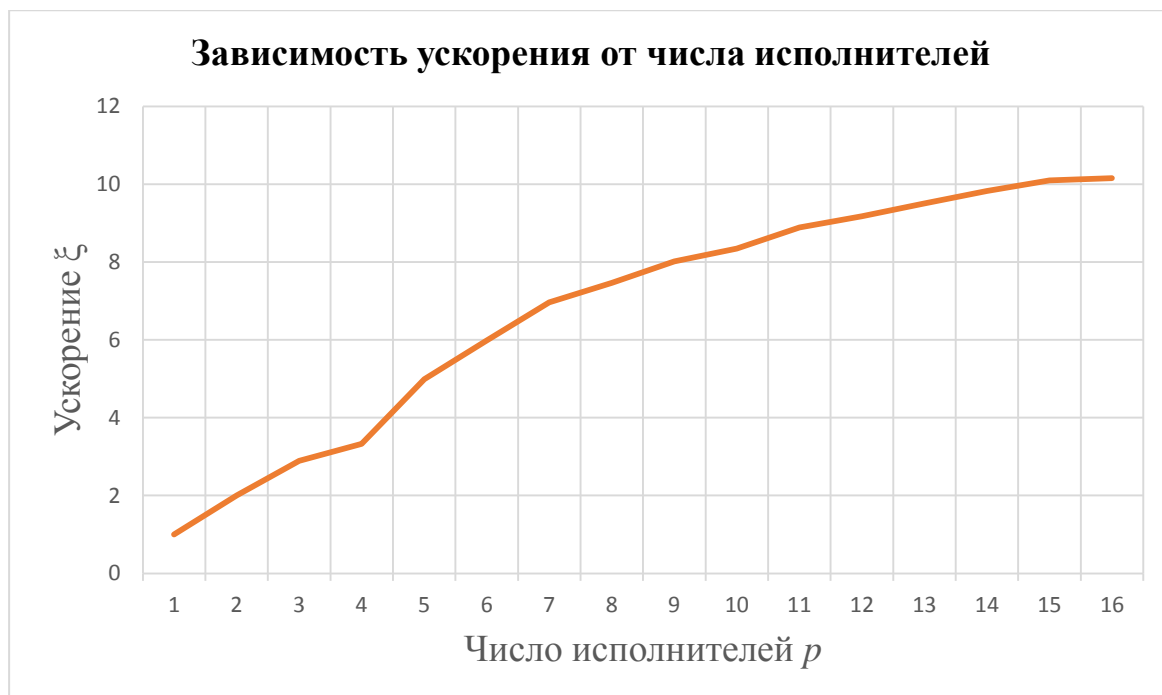
### Зависимость ускорения от числа исполнителей



Умножение матриц, представленных списками списков, размерности  $n = 100$ :

Число исполнителей $p$	Время выполнения $T$	Ускорение $\xi$
1	253,632	1,000
2	126,831	2,000
3	87,710	2,892
4	76,285	3,325
5	50,842	4,989
6	42,398	5,982
7	36,432	6,962
8	33,971	7,466
9	31,631	8,018
10	30,394	8,345
11	28,538	8,888
12	27,637	9,177
13	26,691	9,503
14	25,814	9,825
15	25,112	10,100
16	24,971	10,157



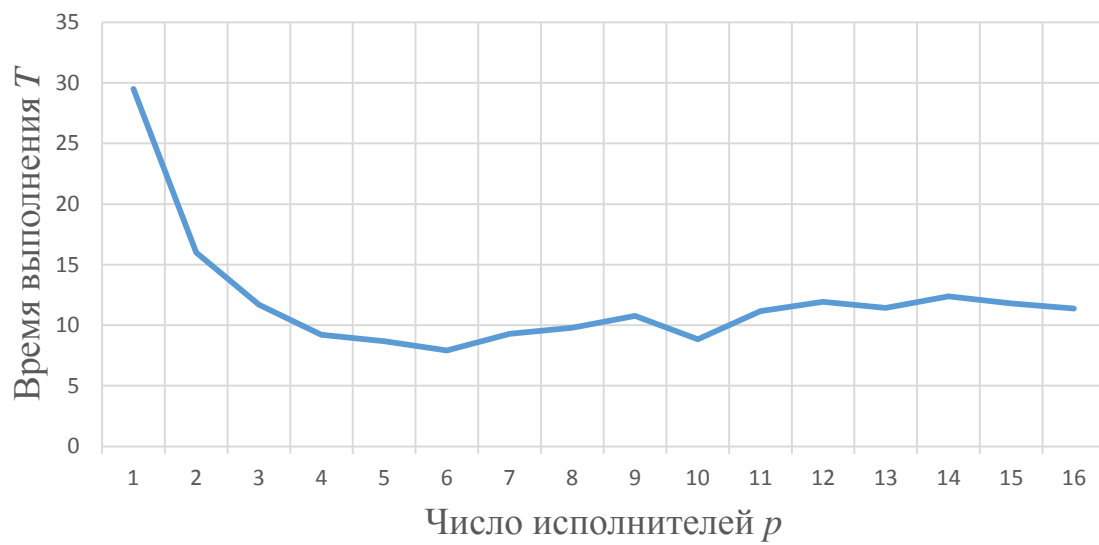


## Быстрая сортировка

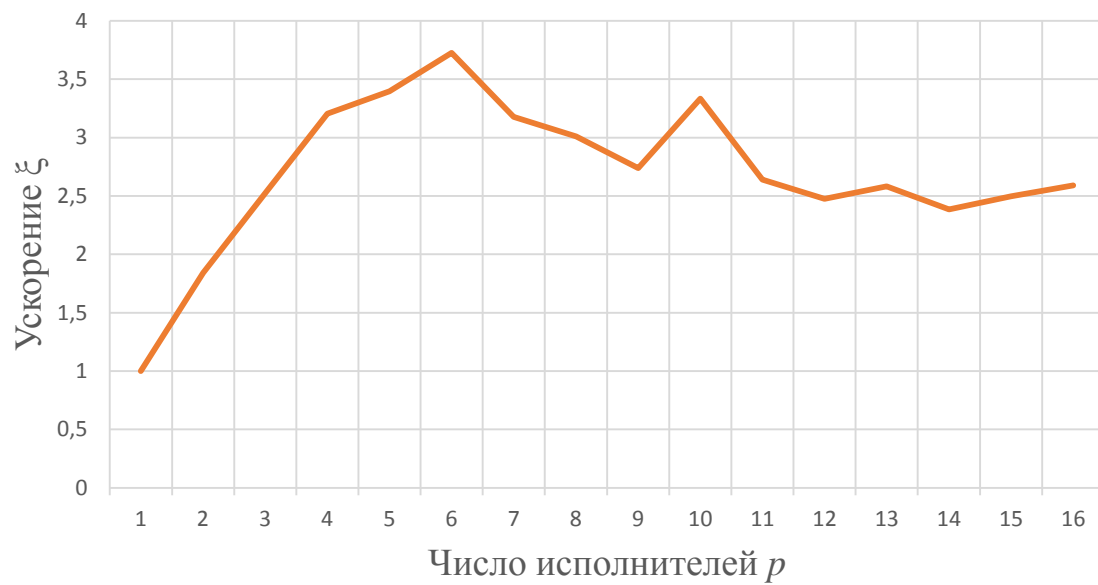
Быстрая сортировка массива размерности  $n = 500000$ :

Число исполнителей $p$	Время выполнения $T$	Ускорение $\xi$
1	29,505	1,000
2	16,014	1,842
3	11,695	2,523
4	9,205	3,205
5	8,691	3,395
6	7,921	3,725
7	9,289	3,176
8	9,804	3,009
9	10,771	2,739
10	8,853	3,333
11	11,179	2,639
12	11,925	2,474
13	11,428	2,582
14	12,370	2,385
15	11,811	2,498
16	11,389	2,591

**Зависимость времени выполнения от числа исполнителей**

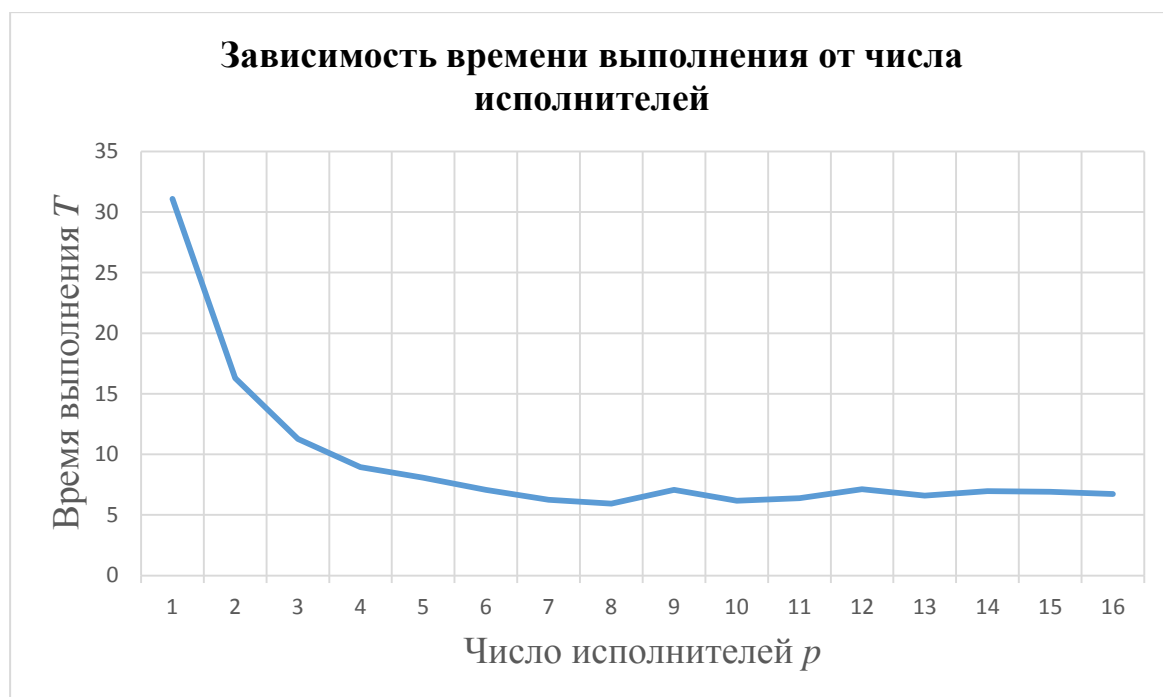


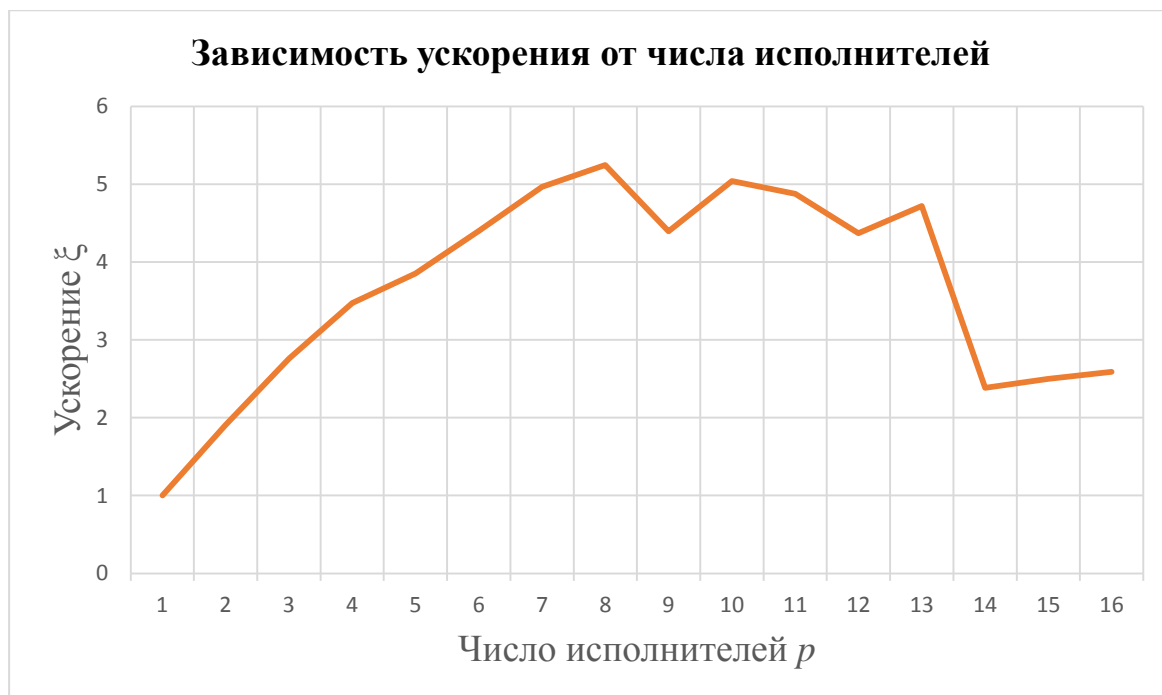
**Зависимость ускорения от числа исполнителей**



# Быстрая сортировка списка размерности $n = 150000$

Число исполнителей $p$	Время выполнения $T$	Ускорение $\xi$
1	31,084	1,000
2	16,302	1,907
3	11,271	2,758
4	8,951	3,473
5	8,069	3,852
6	7,064	4,400
7	6,257	4,968
8	5,926	5,245
9	7,070	4,397
10	6,169	5,039
11	6,372	4,878
12	7,111	4,371
13	6,587	4,719
14	6,969	2,385
15	6,922	2,498
16	6,736	2,591





## Список литературы

1. Бажанов С.Е., Кутепов В.П., Шестаков Д.А., “Структурный анализ и планирование процессов параллельного выполнения функциональных программ”, Известия РАН. Теория и системы управления, № 6, с. 131-146, 2005г.
2. Кутепов В.П., “О параллелизме с разных сторон”, Пленарные доклады пятой международной конференции «Параллельные вычисления и задачи управления», Москва, 2010г.
3. Кутепов В.П., Шамаль П.Н., “Реализация языка функционального параллельного программирования FPTL на многоядерных вычислительных системах”, 2012г.
4. “Архитектура компьютеров и ВС, принципы параллельного программирования.”, Материалы лекций кафедры ПМ, НИУ МЭИ.
5. “Функциональное программирование”, Материалы лекций кафедры ПО ВИА, СБГТУ им. В.Г.Шухова.
6. “Язык программирования FPTL и средства работы с ним”, Воронцов М., <http://files.rsdn.ru/14680/FPTL%20language.html>
7. Кортеж (Информатика), Википедия, [http://ru.wikipedia.org/wiki/Кортеж\\_\(информатика\)](http://ru.wikipedia.org/wiki/Кортеж_(информатика))
8. Тип-произведение, Википедия, <http://ru.wikipedia.org/wiki/Тип-произведение>
9. Tuple, Wikipedia, <http://en.wikipedia.org/wiki/Tuple>

## Приложение

### А. Встроенные функции языка

Функция	Описание	Допустимые типы данных
add	$x + y$	$\text{int} * \text{int} \Rightarrow \text{int}$ $\text{real} * \text{real} \Rightarrow \text{int}$ $\text{real} * \text{int} \Rightarrow \text{real}$ $\text{int} * \text{real} \Rightarrow \text{real}$
sub	$x - y$	
mul	$x * y$	
div	$x / y$	
mod	остаток от целочисленного деления	$\text{int} * \text{int} \Rightarrow \text{int}$
round	целая часть $x$	$\text{real} \Rightarrow \text{int}$
abs	модуль $x$	$\text{int} * \text{int} \Rightarrow \text{int}$ $\text{real} * \text{real} \Rightarrow \text{int}$
equal	$x = y$	$\text{bool} * \text{bool} \Rightarrow \text{bool}$ $\text{int} * \text{int} \Rightarrow \text{bool}$ $\text{real} * \text{real} \Rightarrow \text{bool}$ $\text{real} * \text{int} \Rightarrow \text{bool}$ $\text{int} * \text{real} \Rightarrow \text{bool}$ $\text{string} * \text{string} \Rightarrow \text{bool}$
nequal	$x \neq y$	
less	$x < y$	
greater	$x > y$	
cat	конкатенация строк	$\text{string} * \text{string} \Rightarrow \text{string}$
[i]	выбор $i$ -го элемента кортежа	$'t_1 * \dots * 't_i * \dots * 't_n \Rightarrow 't_i$
<c>	функция-константа: для любого $x$ $\langle c \rangle(x) = x$ , где $c$ – некоторая константа любого типа	$'tc \Rightarrow 'tc$
id	тождественная функция: для любого $x$ $id(x) = x$	$'t \Rightarrow 't$



## В. Грамматика FRTL в БНФ

```
<FunctionalProgram> ::= <DataTypeDefinitionsBlocks> <Scheme>
<Application> | <Scheme> <Application>

<DataTypeDefinitionsBlocks> ::= <DataTypeDefinitionsBlock> |
<DataTypeDefinitionsBlock> <DataTypeDefinitionsBlocks>

<DataTypeDefinitionsBlock> ::= Data <TypeName> { <TypesDefinition-
List> } | Data <TypeName> [ <TypeParametersList> ] { <TypesDefini-
tionList> } | Data <TypeName> { <ConstructorName> { <Construc-
torsDefinitionList> } <TypesDefinitionList> } | Data <TypeName> [
<TypeParametersList> ] { <ConstructorName> { <ConstructorsDefini-
tionList> } <TypesDefinitionList> }

<TypeName> ::= <Name>

<TypeParametersList> ::= <TypeParameterDef> | <TypeParameterDef> ,
<TypeParametersList>

<TypeParameterDef> ::= <TypeParameter>

<ConstructorsDefinitionList> ::= <ConstructorDef> | <ConstructorDef>
<ConstructorsDefinitionList>

<ConstructorDef> ::= => <TypeName> : <ConstructorName> ; | <Con-
structorParametersList> => <TypeName> : <ConstructorName> ;

<ConstructorParametersList> ::= <AtomType> | <AtomType> * <Construc-
torParametersList>

<TypesDefinitionList> ::= <TypeDefinition> | <TypeDefinition>
<TypesDefinitionList>

<TypeDefinition> ::= <TypeName> = <TypeExpression> ; |
<DataTypeDefinitionsBlock>

<TypeDefConstructor> ::= <ConstructorParametersList> .
<ConstructorName> | <ConstructorName>

<TypeExpression> ::= <TypeDefConstructor> | <TypeDefConstructor> ++
<TypeExpression>

<AtomType> ::= <BaseType> | <TypeName> | <TypeName> [ <TypeExpres-
sionsList> ] | <TypeParameter>

<BaseType> ::= string | int | uint | float | double | bool

<TypeExpressionsList> ::= <AtomType> | <AtomType> , <TypeExpression-
sList>
```

```

<TypeParameter> ::= ' <Name>

<ConstructorName> ::= c_ <Name>

<Scheme> ::= <SchemeBegin> { <DefinitionsList> } | <SchemeBegin> [
<FormalParametersList> ] { <DefinitionsList> }

<SchemeBegin> ::= Scheme <Name>

<FormalParametersList> ::= <FormalParameter> | <FormalParameter> ,
<FormalParametersList>

<FormalParameter> ::= <Name>

<DefinitionsList> ::= <Definition> | <Definition> <DefinitionsList>

<Definition> ::= @ = <Term> ; | <FuncVarName> = <Term> ; | <Con-
structionFun>

<Term> ::= <VariantTerm>

<AtomTerm> ::= ( <Term> ) | <ElementaryFunctionName> | <FuncObject-
Name> | @

<FuncObjectName> ::= <Name> | <FuncObjectWithParameters>

<FuncObjectWithParameters> ::= <Name> ( <FuncArgumentList> )

<FuncArgumentList> ::= <FuncParameterName> | <FuncParameterName> ,
<FuncArgumentList>

<FuncParameterName> ::= <Name> | <ElementaryFunctionName> | <FuncOb-
jectWithParameters>

<SequentialTerm> ::= <AtomTerm> | <SequentialTerm> . <AtomTerm>

<CompositionTerm> ::= <SequentialTerm> | <CompositionTerm> * <Se-
quentialTerm>

<ConditionTerm> ::= <CompositionTerm> | <CompositionTerm> -> <Compo-
sitionTerm> , <ConditionTerm> | <CompositionTerm> -> <Composition-
Term>

<VariantTerm> ::= <ConditionTerm>

<ConstructionFun> ::= Fun <ConstructionFunName> { <DefinitionsList>
} | Fun <ConstructionFunName> [ <FormalParametersList> ] { <Defini-
tionsList> }

<ConstructionFunName> ::= <Name>

```

```

<FuncVarName> ::= <Name>

<ElementaryFunctionName> ::= <BuiltInFunction> | <Constructor> |
<Destructor>

<Constructor> ::= <ConstructorName>

<Destructor> ::= ~<ConstructorName>

<BuiltInFunction> ::= <BuiltInFunctionName> | <TupleElement> | <Con-
stant>

<TupleElement> ::= [ <Number> ]

<Constant> ::= <Number> | <RealNumber> | <String> | true | false

<BuiltInFunctionName> ::= <Name>

<Application> ::= Application <InterpFunProgramName> | Application
<InterpFunProgramName> ( <Data> ) | Application <DataInit> <In-
terpFunProgramName> | Application <DataInit> <InterpFunProgramName>
( <Data> )

<DataInit> ::= <OneDataInit> | <OneDataInit> <DataInit>

<OneDataInit> ::= <DataName> = <Value> ;

<DataName> ::= <Name>

<InterpFunProgramName> ::= % <Name>

<Data> ::= <OneData> | <OneData> , <Data>

<OneData> ::= <DataName> | <Value>

<ValueAtom> ::= <Constant> | <Constructor> | ( <Value> )

<ValueConstructor> ::= <ValueAtom> | <ValueAtom> . <Constructor>

<ValueComposition> ::= <ValueConstructor> | <ValueConstructor> *
<ValueComposition>

<Value> ::= <ValueComposition>

<Name> ::= <Letter> | <Name> <Letter> | <Name><Digit> | <Name> _
<Name>

<Letter> ::= A | ... | Z | a | ... | z

<Digit> ::= 0 | ... | 9

```

$\langle \text{RealNumber} \rangle ::= \langle \text{Number} \rangle . \langle \text{Number} \rangle$   
 $\langle \text{Number} \rangle ::= \langle \text{Digit} \rangle \mid - \langle \text{Digit} \rangle \mid \langle \text{Number} \rangle \langle \text{Digit} \rangle$   
 $\langle \text{String} \rangle ::= " \langle \text{StringExpr} \rangle "$   
 $\langle \text{StringExpr} \rangle ::= \langle \text{Symbol} \rangle \mid \langle \text{StringExpr} \rangle \langle \text{Symbol} \rangle$   
 $\langle \text{Symbol} \rangle ::= 0x20 \mid 0x21 \mid 0x23 \mid \dots \mid 0x7E$

## С. Листинг тестируемых программ

### Численное интегрирование методом трапеций

Scheme Integration

```
{
  @ = (0.0 * 10.0 * 0.001).Integrate(Func);

  Func = (((((id*id).mul*id).mul * 0.5).mul.sin * ((id*id).mul *
0.25).mul.sin).mul * (id * 0.125).mul.sin).mul);

  // [1] - левая граница отрезка интегрирования
  // [2] - правая граница отрезка интегрирования
  // [3] - точность интегрирования
  Fun Integrate[Fn]
  {
    @ = (((([1] * Mid).Trp * (Mid * [2]).Trp).add * ([1] *
[2]).Trp).sub.abs * [3]).less -> ([1] * [2]).Trp, ((([1] *
Mid * ([3] * 2.0).div).@ * (Mid * [2] * ([3] *
2.0).div).@).add);

    Mid = (([1] * [2]).add * 2.0).div;

    Trp = ((([1].Fn * [2].Fn).add * ([2] * [1]).sub).mul *
2.0).div;
  }
}
```

Application

% Integration

### Нахождение числа Фибоначчи

Scheme Fibonacci

```
{
  @ = Fib.print;

  Fib = ([1] * 0).equal -> 1, ([1] * 1).equal -> 1, ((([1] *
2).sub.Fib * ([1] * 1).sub.Fib).add);
}
```

Application

% Fibonacci(37)

## Умножение матриц-массивов

Scheme MatrixMultiply

```
{
  N = [1];

  @ = MulMat;

  FillFn = ([2] * (([1]*[3]).mul*[4]).add * rand).arraySet;
  PrintFn = print(" ".print);

  CreateMat = ((N * N).mul * 0.0).arrayCreate;

  MatA = (N * CreateMat * 0 * 0).FillMatrixIJ(FillA);
  MatB = (N * CreateMat * 0 * 0).FillMatrixIJ(FillB);

  // Генерация единичной матрицы
  FillA = ([2] * (([1]*[3]).mul*[4]).add * (([3]*[4]).equal ->
  1.0, 0.0)).arraySet;

  // Генерация константной матрицы
  FillB = ([2] * (([1]*[3]).mul*[4]).add * 2.0).arraySet;

  MulMat = (N * CreateMat * MatA * MatB).FillMatrixIJ(MulRowCol);

  MulRowCol = ([2] * (([1]*[3]).mul*[4]).add *
  ([1]*[5]*[6]*[3]*[4]).DotProduct(GetRowElem, GetColE-
  lem)).arraySet;

  Fun DotProduct[GetRowElem, GetColElem]
  {
    N = [1];
    MatA = [2];
    MatB = [3];
    I = [4];
    J = [5];

    @ = (N * MatA * MatB * I * J * 0).Recurse;

    Recurse = ([6] * N).equal -> 0.0, ((([6] * I * MatA *
    N).GetRowElem * ([6] * J * MatB * N).GetColElem).mul * (N
    * MatA * MatB * I * J * ([6]*1).add).Recurse).add;
  }

  // Возвращает элемент k строки i матрицы M размера n
  Fun GetRowElem
  {
    K = [1];
    I = [2];
    Mat = [3];
```

```

        N = [4];
        @ = (Mat * (K * (I * N).mul).add).arrayGet;
    }

    // Возвращает элемент k столбца j матрицы M размера n.
    Fun GetColElem
    {
        K = [1];
        J = [2];
        Mat = [3];
        N = [4];
        @ = (Mat * (J * (K * N).mul).add).arrayGet;
    }

    // N - размерность.
    // Mat - матрица размера N * N
    // X - доп. параметр 1.
    // Y - доп. параметр 2.
    // Выполнение операции над каждым элементом матрицы с помощью
    // функции F.
    // F: N * Mat * i * j * X * Y (i=0..N-1, j=0..N-1) => Mat
    Fun FillMatrixIJ[F]
    {
        N = [1];
        Mat = [2];

        FillMatrixIJ = (N * Mat * 0 * [3] * [4]).Recurse;

        Recurse = ([3] * N).equal -> Mat, ((N * Mat * [3] * 0 *
        [4] * [5]).FillRow * (N * Mat * ([3] * 1).add * [4] *
        [5]).Recurse).[1];

        // N
        // Mat
        // [3] - строка.
        // [4] - начальный столбец.
        // [5] - доп. параметр 1.
        // [6] - доп. параметр 2.
        FillRow = ([4] * N).equal -> Mat,
        ((N*Mat*[3]*[4]*[5]*[6]).F *
        (N*Mat*[3]*([4]*1).add*[5]*[6]).FillRow).[1];
    }
}

Application
% MatrixMultiply(500)

```

## Умножение матриц-списков

```
Data List['t]
{
    List = c_nil ++ 't * List['t].c_cons;
}

Scheme MatrixMult
{
    GraphSize = 50;
    Lst = (GraphSize * GraphSize).RandomMatrix(10, 0);
    Lst1 = (GraphSize * GraphSize).RandomMatrix(10, 0);
    @=(Lst * GraphSize * GraphSize * Lst1 * GraphSize * GraphSize *
    0 * 0).matrixMult;

    Fun RandomMatrix[Max, Min]
    {
        Fun RandomList[Max, Min]
        {
            @ = (id * 0).equal -> c_nil, ((Min * (rand * (Max *
            Min).sub).mul).add * (id *
            1).sub.RandomList).c_cons;
        }

        TmpMax = Max;
        TmpMin = Min;
        @ = ([1] * 0).equal -> c_nil, ([2].RandomList(TmpMax,
        TmpMin)* ([1] * 1).sub * [2]).RandomMatrix).c_cons;
    }

    /// matrixMult: возвращает результат умножения двух матриц
    // [1]: матрица
    // [2]: количество строчек матрицы [1]
    // [3]: количество столбцов матрицы [1]
    // [4]: матрица
    // [5]: количество строчек матрицы [2]
    // [6]: количество столбцов матрицы [2]
    // [7]: номер текущей строки
    // [8]: номер текущего столбца
    // -> [1]: выходная матрица
    matrixMult = ([7] * [2]).equal -> c_nil, (rowMult * ([1] * [2]
    * [3] * [4] * [5] * [6] * ([7] * 1).add * 0).matrixMult).c_cons;

    /// rowMult: возвращает строку (список) результирующей матрицы
    // [1]..[8]:
    // -> [1]: строка результирующей матрицы
    rowMult = ([8] * [6]).equal -> c_nil, ((id * 0).elemSum * ([1]
    * [2] * [3] * [4] * [5] * [6] * [7] * ([8] *
    1).add).rowMult.[1]).c_cons;
```



```

// elemSum: возвращает элемент результирующей матрицы
// [1]..[8]:
// [9]: номер текущего столбца
// -> [1]: элемент матрицы
elemSum = ([9] * [3]).equal -> 0, ((([1] * [7] * [9]).nnth *
([4] * [9] * [8]).nnth).mul * ([1] * [2] * [3] * [4] * [5] * [6]
*[7] * [8] * ([9] * 1).add).elemSum).add;

/// readMatrix - читает матрицу из файла
// [1]: файловая строка
// -> [1]: список строчек матрицы (строчка - список элементов
// строки)
// -> [2]: количество строчек матрицы
// -> [3]: количество столбцов матрицы
readMatrix = ([1].readFile.readSize).(readElements * [2] *
[3]);

/// readSize - читает размер матрицы в файле
// [1]: файловая строка
// -> [1]: файловая строчка с элементами матрицы
// -> [2]: количество строчек
// -> [3]: количество столбцов
readSize = ([1] * "[\\d]+").getToken.([2]
* "[\\d]+").getToken.([2] * [1].toInt) * [1].toInt).([1] * [3] *
[2]);

/// readNumber - читает число в файле
// [1]: файловая строка
// -> [1]: остаток файловой строки
// -> [2]: прочитанное число
readNumber = ([1] * "[\\d]+").getToken.([2] * [1].toInt);

/// readElements - читает элементы матрицы в файле
// [1]: файловая строка
// [2]: количество строк
// [3]: количество столбцов
// -> [1]: список строчек (строчка - список элементов строки)
readElements = ([1] * "").equal -> c_nil , ((([1] * [2] * [3] *
0).readLine * [2] * [3]).([1] * ([2] * [3] *
[4])).readElements).c_cons;

/// readLine - читает элементы строки матрицы в файле
// [1]: файловая строка
// [2]: количество строк
// [3]: количество столбцов
// [4]: номер текущего элемента
// -> [1]: список прочитанной строки
// -> [2]: остаток файловой строки
readLine = ([3] * [4]).equal -> c_nil * [1],
((([1].readNumber * id).([2] * ([1] * [4] * [5] * ([6] *
1).add).readLine)).([1] * [2])).c_cons * [3]);

```

```

/// orderList: создает список вида (1,2,3, ... , N)
// [1]: число элементов в списке
// -> [1]: список с упорядоченными значениями
orderList = ([1] * 0).orderList1;
orderList1 = ([1] * [2]).equal -> c_nil, ([1] * ([2] *
1).add).([2] * orderList1).c_cons;

/// in,notIn: проверяет, есть ли элемент в списке
// [1]: список
// [2]: элемент
// -> [1]: true или false
in = ~c_nil -> false, ([1].~c_cons.[1] * [2]).equal -> true,
([1].~c_cons.[2] * [2]).in;
notIn = in -> false, true;

/// nth: возвращает n-ый элемент списка
// [1]: список
// [2]: номер
// -> [1] : элемент списка
nth = ([1]*[2] * 0).nth1;
nth1 = ([2] * [3]).equal -> [1].~c_cons.[1], ([1].~c_cons.[2] *
[2] * ([3]*1).add).nth1;

// nnth: возвращает элемент в списке списков (типа S[i,j])
// [1]: список
// [2]: номер внутреннего списка
// [3]: номер элемента внутреннего списка
// -> [1] : элемент списка
nnth = (([1] * [2]).nth * [3]).nth;

/// delete : возвращает n:ый элемент списка
// [1]: список
// [2]: элемент
// -> [1] : список
delete = ~c_nil -> c_nil, ([1].~c_cons.[1] * [2]).equal ->
[1].~c_cons.[2], ([1].~c_cons.[1] * ([1].~c_cons.[2] *
[2]).delete).c_cons;
}

```

Application

```
% MatrixMult("MatrixA.txt","MatrixB.txt")
```

## Быстрая сортировка массива

Scheme ArrayQSort

```
{
  Array = ([1]*([1] * 0).arrayCreate).FillArray(FillFn);

  @ = ([1]*Array).QSort;

  FillFn = ([2]*[1]*(rand*100).mul.toInt).arraySet;

  Fun FillArray[F]
  {
    N = [1];
    Arr = [2];

    @ = (N * Arr * 0).Recurse.[1];

    Recurse = ([3]*N).equal -> Arr*Arr,
              (N*([3]*Arr).F*([3]*1).add).Recurse;
  }

  Fun QSort
  {
    N = [1];
    Arr = [2];

    @ = (0 * (N*1).sub * Arr * N).QSortRecurse(Partition);

    Fun QSortRecurse[Partition]
    {
      A = [1];
      B = [2];
      Arr = [3];
      N = [4];

      Mid = ((A * B).add * 2).div;
      Cmp = (Arr * Mid).arrayGet;

      @ = (A * B).gequal -> Arr, (id * (A * B * Cmp * Arr *
      N).Partition).((A * [6] * Arr * N).QSortRecurse * ([5] *
      B * Arr * N).QSortRecurse).[1];
    }

    Fun Partition
    {
      L = [1];
      R = [2];
      Cmp = [3];
      Arr = [4];
      N = [5];
```

```

    ArrL = (Arr * L).arrayGet;
    ArrR = (Arr * R).arrayGet;

    // Возвращает элемент больше опорного
    FindLeft = ((ArrL * Cmp).gequal -> L, ((L * 1).add * R *
    Cmp * Arr * N).FindLeft);

    // Возвращает элемент меньше или равный опорному
    FindRight = ((ArrR * Cmp).lequal -> R, (L * (R * 1).sub
    * Cmp * Arr * N).FindRight);
    Partition = (L * R).lequal -> ((id * FindLeft *
    FindRight).([6]*[7]).greater -> ([6]*[7]),
    ([6]*[7]*Cmp*Arr*N).Swap.Partition)), L * R;

    Partition1 = (id * FindLeft * FindRight).([6]*[7]);

    // Меняет местами два элемента
    Swap = (L*1).add * (R*1).sub * Cmp * ((id * ArrL * ArrR)
    .((Arr * L * [7]).arraySet * (Arr * R *
    [6]).arraySet)).[1] * N;
    }
}

```

```

Application
% ArrayQSort(500000)

```

## Быстрая сортировка списка

```
Data List['t]
{
  List = c_nil ++ 't * List['t].c_cons;
}

Scheme ListQSort
{
  @ = 150000.RandomList(-9999,9999).QSort;

  Fun RandomList[Max, Min]
  {
    @ = (id * 0).equal -> c_nil, ((Min * (rand * (Max *
    Min).sub).mul).add * (id * 1).sub.RandomList).c_cons;
  }

  Fun QSort
  {
    @ = ~c_nil -> c_nil, (((id*Pivot).Filter(less).QSort *
    (id*Pivot).Filter(equal)).Concat *
    (id*Pivot).Filter(greater).QSort).Concat;

    Fun Filter[fPredicate]
    {
      @ = [1].~c_nil -> c_nil, [1].~c_cons ->
      Args.(([1]*[3]).fPredicate -> ([1] *
      ([2]*[3]).Filter).c_cons, ([2]*[3]).Filter);

      Args = [1].~c_cons * [2];
    }

    // Возвращает опорный элемент (первый в списке).
    Pivot = id.~c_cons.[1];

    Concat = [1].~c_nil -> [2], [2].~c_nil -> [1],
    ([1].~c_cons.[1] * ([1].~c_cons.[2]*[2]).Concat).c_cons;
  }
}

Application
% ListQSort
```