Зачем изучать функциональные языки

В 2009 году автор принимал участие в летней школе Microsoft Research, которая проходила в МГУ на факультете ВМК. На эту школу организаторы пригласили выступать с докладами Саймона-Пейтона Джонса, одного из авторов функционального языка Haskell. Он воодушевленно рассказывал про свой язык, периодически обращая внимание слушателей на интересные конструкции языка и приговаривая "Syntax Sugar" (синтаксический сахар). В то время мне немного было понятно в его презентациях, поскольку такие вещи для неподготовленного слушателя, знакомого прежде всего с традиционной, *императивной* структурой программы, кажутся диковинными.

Я обратил внимание, что Саймон использовал среды и компиляторы с открытым исходным кодом, работая при этом в Microsoft Research. Далее стало понятно, что в те годы эта корпорация (даже несмотря на кризисные явления в экономике) задумалась и стала привлекать ведущих ученых к сотрудничеству в сфере функциональной парадигмы программирования, как результат - их собственный функциональный язык F# отныне включен в стандартную поставку Visual Studio, кроме того, императивные языки уже начинают включать в свой синтаксис некоторые решения из функциональных языков (пример - лямбда выражения, Linq).

Зачем же мировые корпорации добавляют поддержку функциональных языков и функционального стиля в императивные языки? Все дело в том, что в функциональной парадигме вся логика программы выражается в рамках вычислений функций, при этом если представить во времени, такое вычисление может быть не последовательным, а параллельным, построенным так, как это будет казаться компилятору с точки зрения эффективности. Традиционные же программы имеют последовательно выполняющиеся инструкции, которые, даже будучи распараллелены с помощью потоков, будут содержать зависимости по данным и, следовательно, по закону Амдала, будут плохо масштабироваться на сотни и тысячи узлов (что актуально при решении действительно сложных задач).

Функциональные языки также славятся "горячей" заменой кода системы, т.е. динамически можно подменять код функций в рабочей системе без остановки ее работы (запущенные функции отработают, вернут значения и будут заменены на новые). Так, например, была замечена и устранена ошибка в программного обеспечении станции Deep space 1, ведь летающая на расстоянии 100 миллионов миль от Земли станция использовала диалект языка Lisp. Еще пример - компания Ericsson разработала и выпустила распределенный функциональный язык Erlang, который с успехом применяется в телекоммуникационных системах и высоконагружаемых масштабируемых системах.

Как итог, сегодня очень актуально изучать такие языки, в рамках данного курса необходимо показать те задачи, для которых функциональные языки прежде всего подходят, выигрывают в более компактном и понятном коде, объяснить основные парадигмы, общие черты языков, перейти к разработке масштабируемых распределенных приложений.

1 Erlang

1.1 Знакомство с новым языком Erlang

1.1.1 Общие сведения об Erlang

Erlang (Ericcson Language) – функциональный язык, ориентированный на решение различных задач, как специфических, так и общего назначения в том числе по распределенной обработке данных. Разработан телекоммуникационной компанией Ericson для управления телефонными коммуникаторами, далее развивается как продукт с открытым исходным кодом (лицензия Apache Licence 2.0). Язык содержит в себе концепции, близкие языкам Lisp (рекурсивная работа со списками) и Prolog (вывод, сопоставление с образцом), поэтому переходить к обучение Erlang'а проще, имея представление о данных языках.

Что хорошего в языке?

1. Функциональный язык, нет повторного использования переменных, широко распространена рекурсия.
2. Компиляция в байт-код (свой BEAM-формат).
3. Сопоставление по шаблону (удобно для реализации протоколов, разбора данных, экспертных систем).
4. Ориентация на процесс как основную единицу программы.
5. Взаимодействие, основанное на приеме и получении сообщений.
6. OTP (Open telecom platform) – модель рабочих и главных процессов, предназначена для написания отказоустойчивых высоконагруженных приложений.
7. Документо-ориентированная (NoSQL) база данных.
8. Средства горячей замены кода без остановки работающей системы.

Erlang – это один из немногих функциональных языков, программы на котором используются в реальных работающих проектах (Yaws – веб сервер, Erlyvideo – видеосервер, серверный код в мессенджере WhatsApp и др.)

1.1.2 Установка Erlang

Для установки под большинство ОС необходимо либо скачать бинарный инсталлятор с официального сайта http://erlang.org, либо исходный код и собрать его, также можно использовать пакеты, входящие в дистрибутивы ОС (обычно Linux), и установить через средства управления пакетами (однако, версия может быть не самая новая). Существует также сайт со свежими сборками под различные ОС - https://www.erlang-solutions.com/downloads/download-erlang-otp . Под Windows рекомендуется добавить путь к Erlang в переменные окружения path, так чтобы можно было запускать интерпретатор erl из командной строки.

Пример работы с интерпретатором после установки:

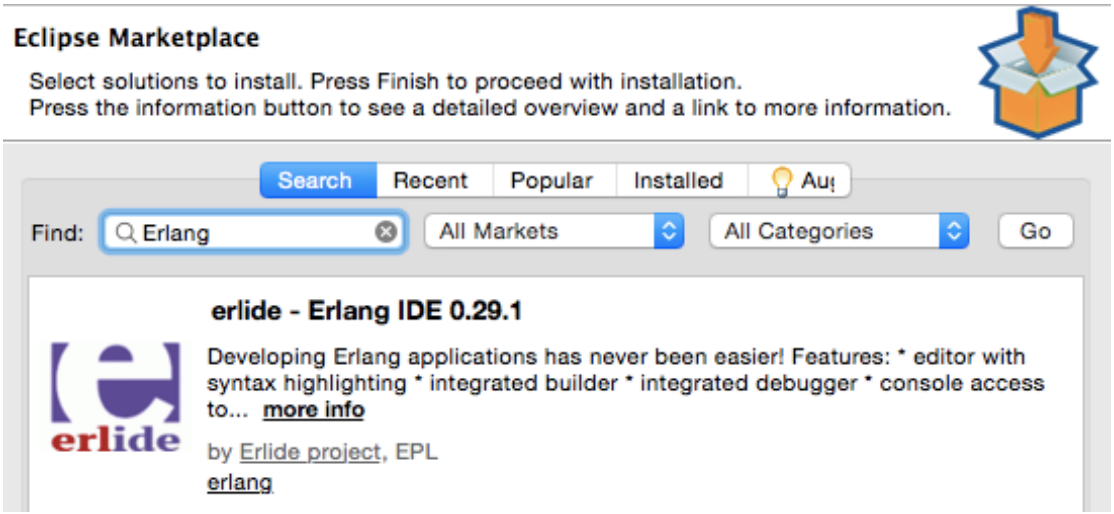
```
bash-3.2$ erl
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]


Eshell V7.0  (abort with ^G)
1> A=3.
3
2> B=4.
4
3> C=A+B.
7
4>
```

Далее в главе 1.1.4 мы рассмотрим работу с интерпретатором подробнее.

1.1.3 IDE для Erlang

Рекомендуется использовать интегрированную среду разработки для написания программ на Erlang. Простые программы можно написать как прямо в интерпретаторе, так и в любом текстовом редакторе, однако работа над проектом большей сложности требует переключение между файлами с исходным кодом, подсветку синтаксиса, запуск проекта, отладку и тд., что как раз и предполагает использование IDE.

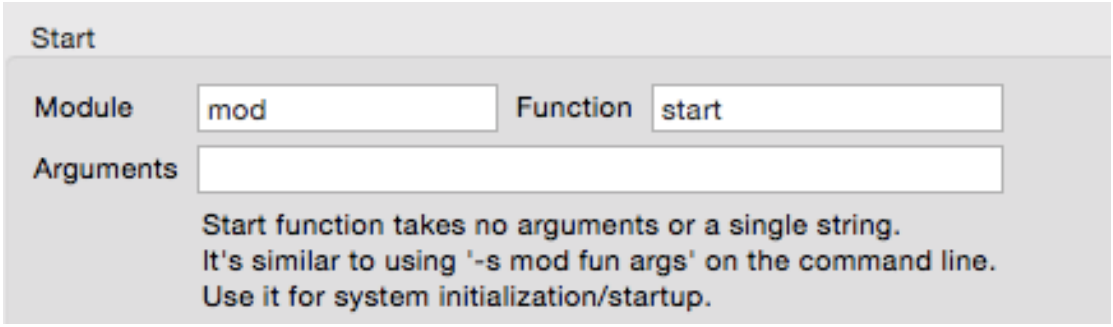
Для Erlang существует среда разработки Erlide, разработанная для среды Eclipse. Для установки ее можно скачать любой дистрибутив Eclipse и в меню Help выбрать Eclipse MarketPlace и в меню поиска ввести Erlang, после этого установить с помощью Install.



После установки доступны новые типы файлов и проектов: для создания проекта существует File->New Erlang Project, для создания модуля (файла с исходным кодом) в уже созданном проекте File->New->Erlang Module.

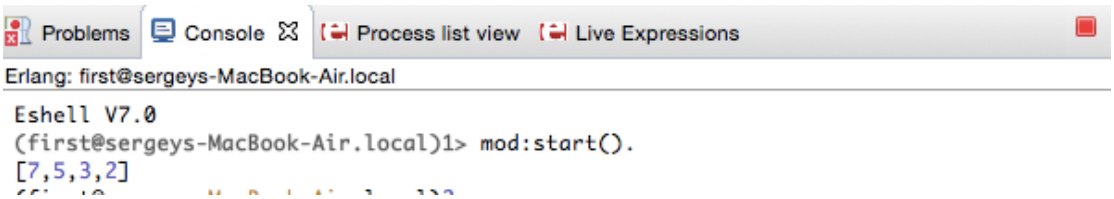
Erlide поддерживает автоматическую компиляцию файлов проекта при сохранении. Поддерживается автозаполнение при вводе (ctrl+пробел), краткая справка по типам функций, а также отладка.

Меню Run->Run configurations позволяет определить конфигурацию запуска, в том числе задать выполняемую стартовую функцию (функция без аргументов или принимающая строку), служащая для инициализации.



Однако, вывод на экран работы данной функции недоступен.

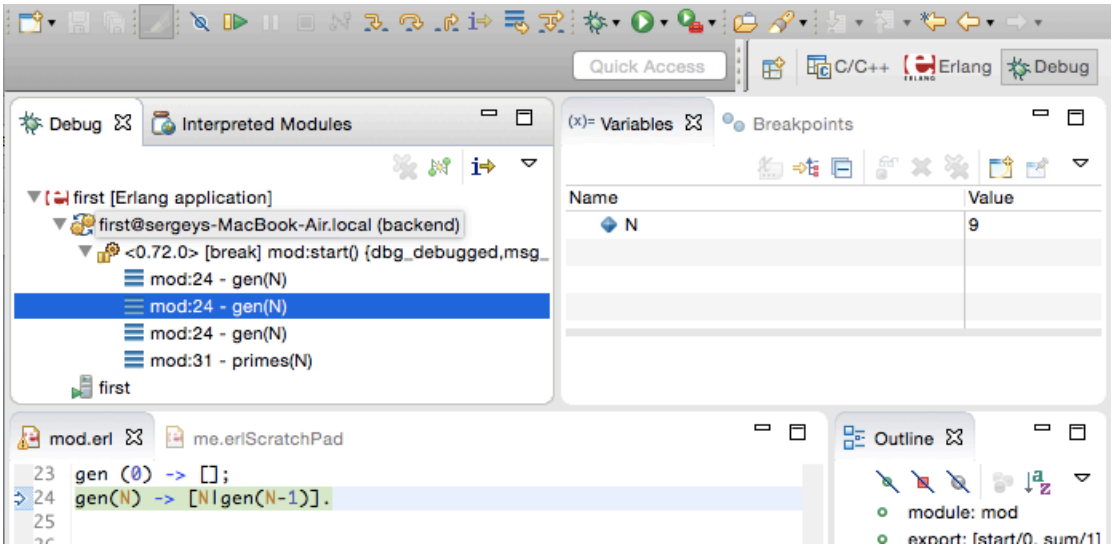
Результаты работы приложения доступны через сеанс интерактивной консоли, которая доступна на ввод с клавиатуры и вывод значений.



Запуск функции производится в формате <модуль>:<функция>(<параметры>).

Для быстрого запуска существует так называемый Scratch pad файл. Он создается при помощи File->New->Erlang scratchpad, в нем записываются выражения на Erlang по строкам, после запуска проекта нажатием Crtl+Enter на строке выражение из Scratch pad копируется в консоль и выполняется.

Для отладки необходимо поставить точку останова, переключиться в перспективу “debug” (перспектива в eclipse – это предопределенный набор окон) и запустить проект в режиме отладки, при этом можно использовать перемещение по стеку вызовов рекурсии, шаг с заходом в тело функции (F5) и без захода (F6).



1.1.4 Работа в интерпретаторе

Интерпретатор вызывается командой erl в командной строке (либо запуском файла erl.exe). После установки путь к erl должен быть добавлен в переменную окружения PATH.

Поддерживаются выражения (например 11111*2222, причем встроена длинная арифметика), вызовы функций, объявление переменных (X=123; Z=[1,2,3]). **Любое выражение или ввод завершается “.”**. При наличии откомпилированных модулей в окружении, фунции из них вызываются после “имя модуля:”.

Если имеется файл с исходным кодом (модуль), перед вызовом функций из него он должен быть откомпилирован командой:

```
c(имя модуля без расширения).
```

В интерпретаторе работает клавиша TAB после ввода первых букв модуля и функции для показа вариантов и автозаполнения.

Запуск интерпретатора как узла кластера распределенной вычислительной системы будет рассмотрен далее.

1.2 Основы языка Erlang

1.2.1 Переменные

Переменные в Erlang обязательно **начинаются с большой буквы** (пример - X, Variable, но не: x, myVar).

Поскольку мы имеем дело с функциональным языком, то мы **не можем переприсвоить** значение переменной второй раз. В Erlang оператор присваивания “=” имеет значения сопоставления (bounding). Это действует следующим образом – если значение переменной на момент выполнения “=” не определено, то осуществляется инициализация переменной значением. Иначе производится попытка сопоставления переменной с противоположной после “=” частью. Пример:

```
1> X.
* 1: variable 'X' is unbound
2> X=10.
10
3> X=X+1.
** exception error: no match of right hand side value 11
4> Y=11.
11
5> X=Y.
** exception error: no match of right hand side value 11
6> Z=10.
10
7> X=Z.
10
```

В примере переменная не определена, далее она устанавливается равной 10, далее осуществляется попытка увеличить ее на 1, но это работает как сопоставление, 11 не равно 10 и выдается ошибка, такая же ошибка выдается при сопоставление с другой переменной, однако, сопоставление работает, когда обе переменные означают одно и тоже.

В Erlang тип данных переменной определяется первым сопоставлением, поэтому он явно не указывается.

Тип может быть определен семейством функций erlang:is_*(erlang тут – имя стандартного системного модуля из библиотеки Erlang), например:

```
X=10.
10
erlang:is_integer(X).
true
```

При инициализации переменной можно использовать выражения с числами (целыми и дробными – 1, 1.1, 1.0e-1), операциями +, -,/ , *, div, rem (остаток от деления), band, bnot, bor, bxor(побитовые логические операции), and, or, not, xor (логические) и другие. Поддерживается встроенная длинная арифметика, кроме того, дроби n/m хранятся как дроби, т.е. например:

```
1/3+1/3+1/3.
1.0
```

1.2.2 Тип переменных (грамматика)

Согласно стандарту языка, переменные могут быть следующих типов:

```
Type :: any()              %% Любой тип
      | none()             %% Неопределенный тип
      | pid()
      | port()
      | reference()
      | []                 %% nil
      | Atom
      | Bitstring
      | float()
      | Fun
      | Integer
      | List
      | Map
      | Tuple
      | Union
      | UserDefined        %% определенный пользователем тип

 Atom :: atom()            %% Атом
      | Erlang_Atom        %% 'foo', 'bar', ...

 Bitstring :: <<>>         %% Битовая строка
           | <<_:M>>       %% M положительное целое
```

```
        | <<_:_*N>>       %% N положительное целое
        | <<_:M, _:_*N>>

 Fun :: fun()                    %% Любая функция
     | fun((...) -> Type)       %% любое число аргументов arity, возвращает тип
Type
     | fun(() -> Type)
     | fun((TList) -> Type)

 Integer :: integer()
        | Erlang_Integer                %% ..., -1, 0, 1, ... 42 ...
        | Erlang_Integer..Erlang_Integer    %% Целый промежуток

 List :: list(Type)                    %% Список ([]-пустой)
     |   maybe_improper_list(Type1,   Type2)          %%   Type1=содержание,
Type2=пустой
     | nonempty_improper_list(Type1, Type2) %% Type1 и Type2 как ранее
     | nonempty_list(Type)              %% непустой список

 Map :: map()                     %% словарь (map – имя-значение)
    | #{}                         %%
    | #{PairList}

 Tuple :: tuple()                 %% Кортеж
      | {}
      | {TList}

 PairList :: Type => Type
        | Type => Type, PairList

 TList :: Type
      | Type, TList

 Union :: Type1 | Type2
```

Далее сложные типы будут рассмотрены более подробно.

1.2.3 Списки

Список в Erlang – это аналог массива данных, однако каждым элементом списка может быть также список или другая сложная структура. Список

обозначается с использованием [], элементы списка перечисляются через запятую. Пример:

```
L=[1,2,3,4,5].
[1,2,3,4,5]
L1=[1,2,[1,2,3],4,5].
[1,2,[1,2,3],4,5]
```

Аналогично спискам в Lisp, в Erlang работа со списками основана на выделении головы и хвоста (head и tail), причем: голова – это первый элемент, а хвост – список из остальных элементов.

Существует оператор | , который а) делит список на голову и хвост б) создает список из заданной головы и хвоста. Поскольку = в Erlang – это сопоставление, сопоставив заданный список и сформированный список из новых переменных, мы получим в них отдельно голову и хвост:

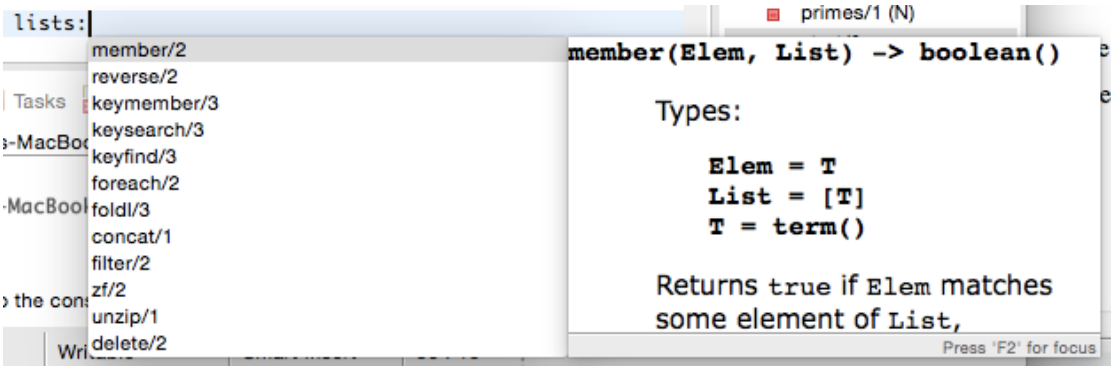
```
1>L=[1,2,3,4,5].
[1,2,3,4,5]
2>[H|T]=L.
[1,2,3,4,5]
3> H.
1
4> T.
[2,3,4,5]
```

обратное сопоставление после того, как H и T определены, получит из них опять список:

```
L2=[H|T].
[1,2,3,4,5]
```

Обратите внимание, обратное выражение [T|H] создаст “испорченный список”, поскольку первый параметр операции | - это элемент, а второй – список, а не наоборот.

В Erlang существует стандартная библиотека для работы со списками, доступная в модуле lists:



Впрочем, большинство из этих функций можно легко написать самому, с использованием рекурсии, как будет показано далее.

Также имеются встроенные операции:

- добавить элемент (или список) в список ++ (добавление в конец).

```
L=[1,2,3,4,5].
[1,2,3,4,5]
LPlus = L++[6].
[1,2,3,4,5,6]
LPlusPlus = L++[7,8,9].
[1,2,3,4,5,7,8,9]
```

- убрать элемент (или список элементов) из списка --:

```
Lminus = L--[1].
[2,3,4,5]
LminusMinus = L--[1,3,5].
[2,4]
```

Следует учесть также, что строки (элементы вида “string”) – это списки из символов.

1.2.4 Функции

Функции в Erlang – это отображение множества значений аргументов в множество значений результата, при этом множество значений характеризуется своим типом. Пример: функция, вычисляющая N+1 для целых N есть Integer->Integer (или, в общем случае, number()->number()), функция, считающая число элементов в списке есть List->Integer (см 2.2).

Имя функции всегда пишется с маленькой буквы.

Функция может иметь параметры. Количество параметров называется арностью функции. В описании функции в некоторых местах арность ставится через / (пример f/2 – функция с двумя параметрами).

Один модуль с исходным кодом может содержать любое количество функций. Функции, которые необходимо сделать доступными другим модулям, описываются в разделе –export модуля. Если функция является промежуточной, то ее экспортировать не обязательно.

Пример объявления модуля с двумя функциями (start без параметров и sum с одним параметром) в файле mod.erl:

```
-module(mod).
-export([start/0, sum/1]).
<описание функций>
```

Для экспорта всех функций без указания их имени можно использовать директиву:

```
-compile(export_all).
```

Функции определяются как последовательность строк вычисления значения для всех возможных значений параметров, отделяемых -> от имени функции.

Пример - функция, вычисляющая A+B:

```
plus (A,B) -> A+B.
```

Функция, делящая два элемента, причем при элементе 0 выводящая предупреждение:

```
divEx (A,0) ->io:format("ops");
divEx (A,B) ->A/B.
```

(в данном случае io:format – это функция из модуля ввода-вывода, она печатает данные на экран и возвращает пустое значение ok).

Если требуется записать последовательно несколько операторов в вычислении функции, то используется оператор “,”, причем значение последнего выражения есть значение функции, пример – вывести предупреждение и вернуть 0 при 0 делимом:

```
divEx(A,0)-> io:format("ops~n"), 0;
divEx(A,B)-> A/B.
```

Если переменная не нужна, в списке аргументов можно писать «_», например, в прошлом примере A в первом случае не используется(и дает предупреждение при компиляции):

```
divEx(_,0)-> io:format("ops~n"), 0;
divEx(A,B)-> A/B.
```

Порядок описания функций с разными вариантами параметров важен. Erlang осуществляет поиск варианта вызова функции, с начала их объявлений в коде, пока не будет найден первый подходящий из списка вариантов, который можно сопоставить, после этого он применяется. Для примера с divEx, если записать

```
divEx (A,0) ->io:format("ops");
divEx (A,B) ->A/B.
```

или

```
divEx (A,B) ->A/B;
divEx (A,0) ->io:format("ops").
```

результат будет различным при вызове divEx(10, 0), т.к. и вариант с B (переменная без значения) и 0 подходят.

При разрешении условий рекомендуется использовать сопоставление. Но если требуется написать условный оператор в теле функции, то можно поступить двумя способами:

- Ограничение на параметры (охранное условие) – с оператором when:

```
divEx(A,B) when B/=0 -> A/B.
```

- If внутри функции:

```
divEx(A,B) ->
    if
        B/=0 -> A/B;
        B==0 -> io:format("ops~n"), 0
    end.
```

1.2.5 Атомы

Атом – это элемент, значение которого равно самому себе. Атом начинается всегда с маленькой буквы и может участвовать в сопоставлении. Пример:

```
1> StudentType=smart.
smart
2> StudentType=dumb.
```

```
** exception error: no match of right hand side value dumb
```

Атом можно преобразовать в строку (список) или обратно при помощи atom_to_list() и list_to_atom() из модуля erlang.

Основное использование атома – ключ в сопоставлениях, пример – требуется функция, которая возвращает температуру в цельсиях как есть, а при подачи на вход температуры в фаренгейтах – преобразует ее в цельсии:

```
temp(cel,  T) -> T;
temp(fahr, T) -> 5/9 * (T - 32).
```

Использование:

```
> mod:temp(fahr, 0).
-17.77777777777778
> mod:temp(cel, 0).
0
```

1.2.6 Кортежи

Кортежи (tuples) – это элементы, содержащие связанную информацию. Кортежи характеризуются структурой, а не порядком, как списки. Кортежи хранят логически однородные данные. Например, дата рождения может быть задана как один кортеж {18, august, 1984}. Такие данные передаются как параметры функциям и участвуют в сопоставлении. Пример – как извлечь день, месяц, год из даты рождения:

```
Date  = {18, august, 1984}.
{18,august,1984}
> {D,M,Y}=Date.
{18,august,1984}
> D.
18
> M.
august
> Y.
1984
```

Также можно использовать библиотечные функции erlang:element(номер, кортеж) и erlang:setelement(номер,кортеж, значение).

```
> erlang:element(1,Date).
18
> NewDate=erlang:setelement(3,Date,2015).
{18,august,2015}
```

В кортеже может содержаться другой кортеж или список, как и в списке-кортеж. Соответственно, чтобы добраться до его содержимого, нужно построить образец такого-же уровня вложенности. Пример – пусть есть список из 2х человек с их датой рождения [{ivanov, {1, september, 2000}}, {petrov, {1, may, 1990}}], для извлечения месяца рождения Иванова (все другие параметры нам не интересны), можно составить следующий образец: `[{_,{_,X,_}},_]`

```
>L=[{ivanov, {1, september, 2000}}, {petrov, {1, may, 1990}}].
[{ivanov,{1,september,2000}},{petrov,{1,may,1990}}]
1> [{_,{_,X,_}},_]=L.
[{ivanov,{1,september,2000}},{petrov,{1,may,1990}}]
> X.
september
```

Используя кортежи и атомы, мы можем перенести аналог специального параметризма в программу на языке Erlang. Рассмотрим класс Фигура и его подклассы Треугольник, Прямоугольник и Круг. В подклассах Фигуры имеется метод для вычисления площади (площадь треугольника – 1/2*h*a, прямоугольника – a*b, круга – п*r*r). В терминах Erlang можно передать в функцию кортеж с атрибутами фигуры и ее идентификатором.

```
sq ({treug, A, H})    -> 1/2 * A *H;
sq ({rectangle, A, B})-> A * B;
sq ({round, R})       -> math:pi() * R * R.
```

Далее, для вычисления площади, например, круга радиусом 10 достаточно написать sq({round, 10}). Функциональный подход: та же функция над разными данными.

1.2.7 Функторы

По языку С известно, что можно передать указатель на функцию в другую функцию как параметр и применить в нужном месте алгоритма. В С++ можно перегрузить оператор () и в этом случае класс становится так называемым функтором. В Erlang в качестве параметра можно задать в том числе и функцию.

Функтор (функциональный объект) – это функция, которая может быть использована как параметр другой функции либо в другом месте исходного кода.

Функтор объявляется в любом месте, где разрешено вычисление выражения, как переменная:

```
Fun1= fun(A,B) -> (A > B) end.
```

Далее эта переменная может быть передана как параметр и применена (например, как Fun1(1,2) вернет false).

Функторы могут быть использованы в качестве операций над списками. Стандартная функция lists:map (функция, список) применяет заданную функцию ко всем элементам списка и возвращает список из результатов этих применений, например вот код, который возвращает квадраты значений:

```
>FunSq = fun(X)->X*X end.
#Fun<erl_eval.6.54118792>
> L=[1,2,3,4,5].
[1,2,3,4,5]
> lists:map(FunSq,L).
[1,4,9,16,25]
```

Для работы со списками удобна запись в форме предикатов, встроенная в язык:

```
[X*X || X <- [1,2,3,4,5]]
```

Такая запись получает в X по очереди каждый элемент из указанного списка и применяет к нему X*X, которые становятся элементами нового списка

```
[X*X || X <- [1,2,3,4,5]].
[1,4,9,16,25]
```

1.2.8 Рекурсия

В Erlang нет цикла “for”. Для перебора значений используется “хвостовая” рекурсия, которая транслируется компилятором в эффективный нерекурсивный код (по возможности).

Пример: сложение чисел от 1 до N:

```
sum(0) ->0;                    % в 0 – завершение рекурсии
sum(N) -> N + sum (N-1).       % N + предыдущая накопленная сумма
```

Данную сумму можно расписать и получим: sum(N)=N+ sum(N-1)=N+ (N-1) + sum(N-1) = …= N+ (N-1)+ (N-2)+ …+ 1 + 0. При N=0 производится останов рекурсии.

Другой пример рекурсии – рекурсивная работа со списками, для примера напишем свою функцию, вычисляющуу сумму элементов списка:

```
sumList([]) -> 0;                % сумма пустого списка – 0
sumList([H|T]) = H + sumList(T). % делим список на голову и хвост и
вычисляем сумму как элемент + сумма для оставшейся части списка
```

В примере для вычисления суммы элементов в списке производится первоначальное сопоставление переданного списка с пустым, если он сопоставляется с пустым, то срабатывает первый вариант функции, иначе он сопоставляется со списком [H|T] и переменные после сопоставления будут содержать в H – первый элемент списка, а в T – оставшуюся часть списка (H – всегда элемент, а T-список). Далее сумма считается как первый элемент + сумма от оставшейся части списка. Например: sumList([1,2,3,4]) = 1+sumList([2,3,4]) = 1+(2+sumList([3,4]))=1+(2+(3+sumList[4])) = 1+(2+(3+(4+sumList([])))) = 1+(2+(3+(4+0))) = 1+2+3+4=10.

Пример рекурсивного деления элементов заданного списка на 2:

```
div2 ([])->[];
div2 ([H|T])->[H/2 | div2(T)].
```

Пример генерации списка N…1 (аналог - стандартная функция lists:seq(N, 1, -1)):

```
gen (0) -> [];
gen (N) -> [N | gen(N-1)].
```

Пример проверки числа на простоту (по определению – у простого числа 2 делителя - само число и 1) и поиск простых чисел до N с использованием этой функции:

```
isPrime (N) -> sum ([1 || X<- gen(N), (N rem X)=:=0 ]) =:= 2.
primes (N) -> [X || X<-gen(N), isPrime(X)].
```

Рассмотрим более сложный пример – быстрая сортировка массива (применительно к Erlang – списка). Данная процедура сортировки известна тем, что она изначально рекурсивная.

```
qsort([]) -> [];
qsort([H | T]) ->  qsort([Front || Front <- T, Front < H])
   ++ [H] ++
```

```
    qsort([Back || Back <- T, Back >= H]).
```

Пример использует оператор ++, которая сцепляет списки, в данном случае на каждом шаге рекурсии формируется отсортированный список Front, состоящий из элементов списка, меньших заданного (первого элемента), который сцепляется со списком из самого элемента и с отсортированным списком Back из элементов больше заданного. Таким образом, получаем на каждом шаге отсортированные списки, которые формируются путем правильного расположения (до, элемент, больше) и, согласно принципу математической индукции, все элементы будут в этом случае отсортированы.

Пример рекурсивного обхода дерева каталогов с использованием функций Erlang по работе с файлами и каталогами из модуля file:

```
recurser(Dir)->
        {ok, Filenames}=file:list_dir(Dir),
        processOne(Dir,Filenames).
```

Эта функция для заданного каталога возвращает список файлов и каталогов в нем и вызывает рекурсивную функцию его обработки:

```
processOne(Parent, [First|Next])->
        FullName=Parent++"/"++First,
        IsDir=filelib:is_dir(FullName),
        if
                IsDir==true -> recurser(FullName);
                IsDir==false -> io:format(FullName++"~n")
        end,
        processOne(Parent,Next);
processOne(_Parent,[])->ok.
```

Данная функция получает на вход каталог и список с файлами в нем, обрабатывает каждый элемент списка хвостовой рекурсией, при обработке каждого файла формируется полное имя файла, и проверяется, является ли он файлом или каталогом, если каталогом, то для него поиск запускается рекурсивно, если файлом - то его имя выводится.

1.2.9 Функциональный стиль

В последнем примере используются операторы ввода-вывода, и вообще, сама программа написана в императивном стиле. В обычной реализации на

императивном языке мы получим такой-же код с рекурсивным обходом, различие будет только в цикле прохода по списку файлов (в Erlang приходится делать хвостовую рекурсию).

На летней школе MS Research по функциональным языкам был продемонстрирован слайд:

Он означает, что при программировании в функциональном стиле следует **избегать последовательного кода** (в Erlang оператор “,”). Функциональный код должен быть вида: вызов функции (вызов функции (… (параметры))…).
Для избавления от императивного кода можно сделать следующее:

- Вводить дополнительные маленькие функции для промежуточных вычислений.
- Передавать в них вычисления на предыдущем шаге через параметры.
- Отказываться от условных операторов: использовать объявление разного тела функции для разных параметров, условный оператор будет работать в виде сопоставления.
- Избавляться от дублирующего кода сопоставлением по одинаковым переменным.

Модифицируем пример выше.
Видим код:

```
FullName=Parent++"/"++First,
        IsDir=filelib:is_dir(FullName),
```

Для избавления от первого “,” делаем функцию, которая сцепляет два параметра:

```
construct(Parent, First)-> Parent++"/"++First.
```

Теперь можно уже заменить код на ее вызов и избавится от одного последовательного кода:

```
        IsDir=filelib:is_dir(construct(Parent, First)),
```

И так делаем последовательно далее.
Для избавления от if

```
        if
                IsDir==true -> recurser(FullName);
                IsDir==false -> io:format(FullName++"~n")
        end,
```

Делаем функцию “с перегруженным телом”

```
processByDir (true, FullName) -> recurser(FullName);
processByDir (_, FullName) ->  io:format(FullName++"~n").
```

Сопоставление идет последовательно, поэтому во втором случае пустой первый параметр будет означать «иначе».
В итоге после последовательных преобразований последовательного кода получаем функциональный код с промежуточными функциями:

```
getFileNames({_, Names}) -> Names.
construct(Parent, First)-> Parent++"/"++First.
processByDir (true, FullName) -> recurser(FullName);
processByDir (_, FullName) ->  io:format(FullName++"~n").

recurser(Dir)->
 processOne(Dir, getFileNames(file:list_dir(Dir))).

processOne(Parent, [First|Next])-> processByDir(filelib:is_dir(construct(Parent, First)), construct(Parent, First)), processOne(Parent,Next);
processOne(_Parent,[])->ok.
```

Видим дублирование кода. construct(Parent, First) вызывается два раза. Делаем так, чтобы был один вызов и он передавался. Готовый вариант:

```
getFileNames({_, Names}) -> Names.
construct(Parent, First)-> Parent++"/"++First.
```

```
processByDir (true, FullName) -> recurser(FullName);
processByDir (_, FullName) ->  io:format(FullName++"~n").


recurser(Dir)->
 processOne(Dir, getFileNames(file:list_dir(Dir))).


processConstruct (D) ->   processByDir(filelib:is_dir(D), D).
processOne(Parent, [First|Next])-> processConstruct(construct(Parent, First)),
                                                    processOne(Parent,Next);
processOne(_Parent,[])->ok.
```

Запуск mod:recurser(“.”) работает ожидаемо. Мы “офункционалили” код, сделали его менее читаемым, зато отделили логику от вывода (для изменения способа вывода на экран не нужно менять логику основной функции), сделали код функциональным.

Задачи по рекурсии и спискам для самостоятельного решения

1. Дано число N. Сгенерировать список из цифр числа.
2. Найти все совершенные числа (число равно сумме своих делителей) , не превосходящие заданного N.
3. Дан список чисел. Отсортировать его методом пузырька.
4. Дан список чисел. Отсортировать его сортировкой слиянием.
5. Сгенерировать список списков значений функции Аккермана A(m,n) для заданных аргументов 0…m, 0…n
6. Бинарное дерево задано в виде списка, в каждом узле хранит либо код операции (+-*/), либо число. Посчитать значение выражения, заданного деревом.
7. Два числа заданы по цифрам в списках. Сложить эти два числа поразрядно.
8. Дан список. Определить, содержится ли он в виде последовательного подсписка в другом списке.
9. Дано N списков с элементами. Считая каждый список множеством, создать пересечение этих множеств.
10. Дано N – число дисков. Решить задачу Ханойских башен для этого числа дисков и вывести решение.
11. Дан матрица как список из списков. Посчитать обратную матрицу.
12. Даны 2 матрицы как списки из списков. Посчитать произведение матриц по определению.
13. Даны 2 матрицы как списки из списков. Посчитать произведение матриц алгоритмом Штрассена.
14. Дан список чисел. Найти НОД.
15. АВЛ-дерево задано списком. Написать процедуру добавления в дерево с балансировкой.
16. Даны два одноразмерных списка чисел и степеней. Возвести числа в заданные степени алгоритмом быстрого возведения в степень.
17. Социальная сеть. Даны списки из пользователей, а также их друзей. Найти тех людей, которые не являются друзьями своих друзей.
18. Дан список из студентов с их оценками по разным предметам. Определить, какой из предметов наиболее простой для всех студентов.