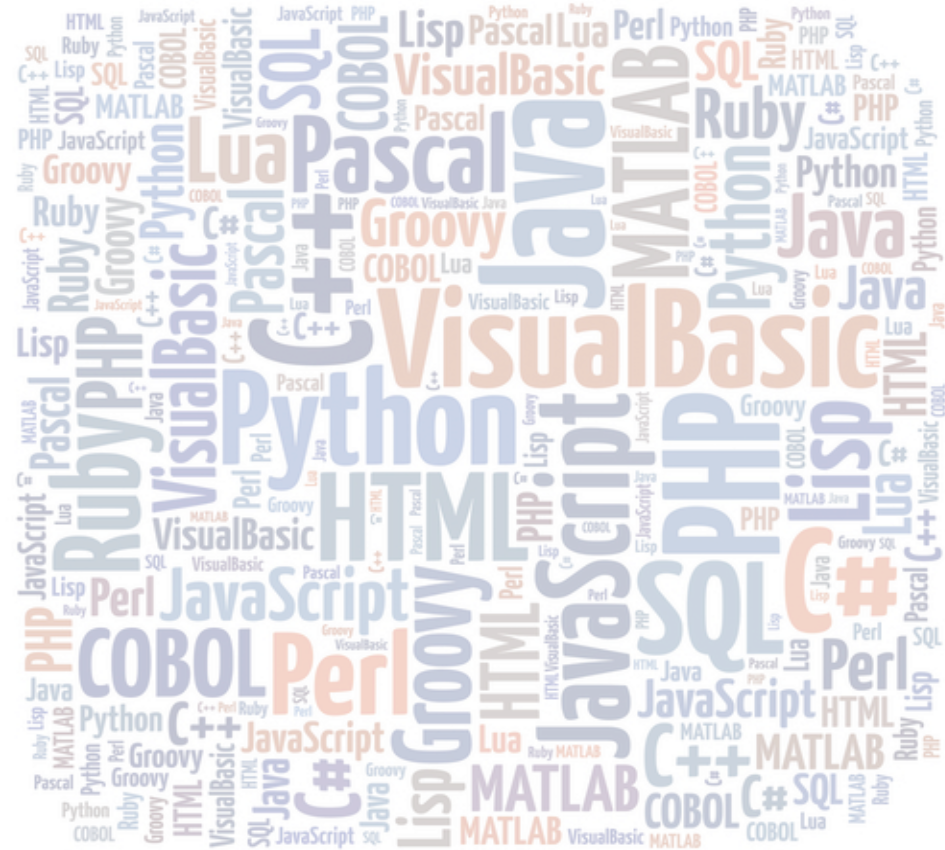


Intro to Erlang



Introduction

- So what is Erlang
- History
- Virtual machine model
- Erlang syntax
 - Types
 - Expression
 - Whitespace
- Data structures
 - Tuples
 - Lists
 - Maps
 - Records

Erlang

- Erlang is functional language with an emphasis on
 - Massive concurrency
 - Fault tolerance/high availability
- It was developed at Ericsson in the 1980s and 1990s by Joe Armstrong (and others) with a particular focus on the needs of the telecommunication industry
 - However, it is a general purpose language used by many companies/products, including
 - Amazon
 - Facebook
 - WhatsApp
- Erlang was open sourced in 1998
- Its strength is its ability to scale safely to enormous volumes of users/connections.

Erlang

- As noted, Erlang is a complete general purpose language.
 - As such, it is relatively large and has many features that one can explore.
- However, our focus in this course will be on concurrency.
- Erlang is particularly well known for this and in many ways is the “king/queen” of concurrent languages.
 - No language does this better.
- If you are a Java programmer, it is worth noting that the Akka concurrency framework is based directly upon the Erlang language.

Basics

- Before we can look at concurrency, of course, we have to look quickly at the basic syntax of the language.
- Erlang, like Clojure, is a functional language.
- Unlike Clojure, it is not a direct mapping of the LISP language.
- So the syntax may seem either odd (in places) or a little more like C/Java in other places.
- For the most part, though, Erlang will utilize the general programming model of any functional language, including Clojure

Basics...cont'd

- Like Java, Erlang runs on top of a virtual machine.
 - In this case, it's called BEAM
 - When you compile erl files, you will note that the compiled output uses the suffix “.beam”
- Again, like Java, Erlang can support other languages that run on its VM
 - Elixir is one such language
 - It is like an updated Erlang, with arguably nicer syntax.
- Erlang code can be run in two ways:
 - An interactive command line shell called Eshell
 - By compiling the code and running it directly with erl.
- Erlang source files use the .erl suffix

Syntax

- Let's start simple
- Erlang uses a combination of simple data types and complex data structures
- Like Clojure, there are no *type declarations*
 - Types are inferred from context
- Simple types include
 - Integers
 - Arbitrary length: can be as many digits as you like
 - Floating point values
 - Stored internally as doubles
 - Strings
 - Surrounded by double quotes
 - Do NOT use single quotes.
 - There is no boolean type (that could be assigned to anything) but there are true/false “constants” that are returned when conditional checks are carried out

Syntax...cont'd

- Unlike Clojure, Erlang does not use LISP-style prefix notation operations
 - So Erlang uses $(2 * 4)$, not $(* 2 4)$.
- It is possible to bind a value to a name/label
 - `Total = 43`
- Note: there are a couple of things to keep in mind here
 - These names can only be assigned ONCE.
 - Attempting to re-assign them will generate an error
 - Such names **MUST** begin with an uppercase letter.
 - Again, an error will be generated otherwise.

Syntax...cont'd

- Erlang uses several “separators” to distinguish elements/expressions from each other
 - Commas are used between function arguments and between expressions in a function body
 - Semi-colons are used between alternate forms or arities of a function
 - A period (plus a whitespace char like a newline) is used to terminate a complete expression.
- If you are using the shell, a common mistake is to forget the period at the end of the expression.

Data structures

- Erlang provides four main composite data structures
 - Tuple
 - List
 - Record
 - Map
- Tuples are the most basic type and are used as a simple way to represent a fixed number of elements
 - This would be roughly analogous to a struct in C
 - Erlang programmers use them A LOT.

Tuple...cont'd

- Tuples are written using curly braces, with elements separated by commas.
 - {“John Smith”, 43}
- Note that, unlike a C struct, there are no labels associated with a tuple (e.g., no name/age labels)
- In practice, Erlang programmers often use an “atom” as the first element of a tuple
 - An atom is essentially a constant label/string.
 - It MUST begin with a lowercase letter
- So we might have something like the code below
- Here, P will be assigned the tuple.
 - P can NOT be changed in the future

```
P = {person, “John Smith”, 43}.
```

Extracting values from Tuples

- Erlang makes extensive use of *pattern matching*.
- For example, with tuples, we can extract the values of the tuple with the code below.
- Note the following:
 - Comments in Erlang use the % character
 - Erlang uses pattern matching to map the fields of P (right hand side or RHS) to the elements on the left hand side (LHS)
 - An error (i.e., no match of right hand side) will be generated if the match is not valid.
 - Finally, we can use an underscore “_” when we do not need to save a particular field.

```
P = {person, "John Smith", 43}.  
  
{person, _ , Age} = P.  
  
% Age now = 43
```

Lists

- Not surprisingly, Erlang has a list structure.
- We use square brackets to define a list, with each element separated by commas
- Lists can contain any other types/structures and can be of arbitrary length.
 - [14, 26, {person, “john”, 43}]
- Erlang considers lists to consist of a Head and a Tail
 - The head is the first element, the tail is everything else.
- We can create and deconstruct lists directly using the syntax below

```
List1 = [14, 12].
```

```
List2 = [“abc” | List1].
```

```
[TheHead | TheRest] = List2.
```

```
% TheHead = “abc”, TheTail = [14, 12]
```

Lists...cont'd

- There are dozens of additional list functions provided by the Erlang framework.
- These include things like:
 - Append
 - Delete
 - Sort
 - Split
 - Reverse
- All of the library documentation is online, so use that as a resource.

Records

- Erlang also provides a mechanism for creating records
- Records are like tuples, except that we can associate labels with values.
- Records are a little different than other types because they do have an actual declaration.
- The general format is listed below
 - Note that the name and keys must be atoms.

```
-record(name, {  
    key1 = val1,      % val1 is a default  
    key2 = val2,      % val2 is a default  
    ...  
    key3,             % no default for key3  
    ...  
}).
```

Records...cont'd

- So we are ready to create and use the record.
- Below, we declare a person record and provide a default for name.
- To create a record, we use the `#{ }` syntax.
 - We can use the default values
 - We can also over-ride the defaults by using the key name
- Finally, we can access the field using the `Var#record.field` syntax

```
-record(person, {name = unknown, age}).  
R1 = #person{}.    % name = unknown, age = undefined  
R2 = #person{age = 99}. % name = unknown, age = 99  
Age = R2#person.age.    % Age = 99
```


Importing record declarations

- Unlike the other data structures, record declarations are generally declared separately and then included in the current file(s).
 - This is a lot like C, with its “header” files.
 - Erlang actually has a separate preprocessor
- This is done simply by adding an include statement at the top of the current file, as shown below
 - Included files are given the .hrl file extension

```
-include("person.hrl").  
...  
R1 = #person{}.    % name = unknown, age = undefined
```

Maps

- Maps behave like the comparable structures in Python and Clojure.
 - Introduced in version 17 of Erlang (so you cannot use an old version of Erlang for this)
- The basic syntax is
 - `#{key1 Op Val1, key2 Op Val2, ...key3 Op Val3}`
 - Op can either be `=>` or `:=`
 - The syntax uses `#{ }`, like a record, but there is no record name in the first position.
- Basic use is in the sample below
- Like lists, there is a map module that provides many functions, including lookups

```
M1 = #{man => "joe", women => "sue"}.  
...  
M2 = M1.    % M2 is a copy of M1
```