Summarization of Julia programming language
Assignment 3

SILPA C

Ph.D.,IIIT Chittoor

1.Julia Features

- Julia is a Dynamically typed with fast user-defined types
- Julia combines Python's convenience with C's performance
- Syntactically it is easy to learn from Python
- Includes some of C and Python standard library functions, ccall(), pycall() is used to call the C and Python library functions.
- Designed for just-in-time compilation
- Julia's implementation of message passing is different from other environments such as MPI, Julia is a high level language whose performance in parallel computation is comparable to Message Passing in C/C++ (used in HPC - High Performance Computing clusters) making it an amazing language for parallel computing.
- Multiple Dispatch, when a function is called in Julia it will lookup in a table at runtime which concrete function it should call based on the types of its arguments.

2. Variables, Types, Dictionary, Numbers, Data Structures

2.1 Variables

- Names of variables are in lower case, word separation can be indicated by underscores
- Some of the special symbols like alpha, beta can also be used as variable names.
- Some of the constants can also be used as variable names

Ex:
pi is a constant which holds its value as 3.14592... ,
if it is used as a variable it can be assigned to new value but shows some warning message
julia> pi
n= 3.1415926535897...
julia> pi=15
WARNING: imported binding for pi overwritten in module Main
15

2.2 Numbers

- In Julia the complex numbers can be represented as 3+5im, imaginary part was represented with im to avoid conflicts in polynomial equations.
- Polynomial equations was represented in Julia as 2x2-3x+1, no need of representing * symbol.
- In 5/2 returns 2.5, if there is no need of fraction part then we can use div(5,2) then it returns 2
- Julia has a // division operator, but it returns a rational number rather than an integer.
- Julia evaluates a prefix arithmatic expression

Ex:
julia> +(4,3)
7

2.3 Type Declarations

- Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types.
- Names of Types and Modules begin with a capital letter and word separation is shown with upper camel case instead of underscores.
- The:: operator can be used to attach type annotations to expressions and variables in programs.

julia> a=10
10
julia> typeof(a)
Int64

2.3 Abstract types

Abstract types are declared using the abstract keyword. The general syntax for declaring an abstract type are:
abstarct << *name* >>
abstarct << *name* >><:<< *supertype* >>
The abstarct keyword introduces a new abstract type, whose name is given by << *name* >>. This name can be optionally followed by <: and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

Ex:
abstract MyAbstractType
By default, the type you create is a direct subtype of Any :
julia> super(MyAbstractType)
Any
we can specify type using the <: *operator*.
abstract MyAbstractType2 <: *Number*

2.4 Concrete Types

In Julia we can create new concrete types. To do this, use the type keyword, which has the same syntax as declaring the supertype. Also, the new type may contain multiple fields, where the object stores values.
Ex:

```
MyAbstractType :
        type MyType <: MyAbstractType
                msg
                val::Int
        end
```

Here we created a type called MyType , a subtype of MyAbstractType , with two fields: msg that can be of any type, and val , that is of type Int .

julia> x = MyType("Hello World!", 10)
MyType("Hello World!", 10)

2.5 Functions

There are various syntaxes for defining functions:

- function containing a single expression
- function containing multiple expressions
- function doesn't need a name

1)Single expression functions

To define a simple function, provide the function name and argument on the left and an expression on the right of an equals sign. These are like mathematical functions:
Ex:
julia> f(x) = x * x
f (generic function with 1 method)
julia> f(2)
4
julia> g(x, y) = sqrt(x^2 + y^2)
g (generic function with 1 method)
julia> g(3,4)
5.0
2)Functions with multiple expressions
The syntax for defining a function with more than one expression, here is a typical function that calls two other functions and then ends.

```
julia> function breakfast()
            maketoast()
            brewcoffee()
        end
breakfast (generic function with 1 method)

To return more than one value from a function, use a tuple.
julia> function doublesix()
(6,6)
end
doublesix (generic function with 1 method)
julia> doublesix()
(6,6)
```

3)Anonymous functions
Anonymous functions are the functions with no name can be used in a number of places in Julia, such as with map().

Ex:
i)x -> x 2 + 2x - 1, which defines a nameless function that takes an argument,

```
calls it x, and produces x 2 + 2x - 1 .
ii)For the map() function,

julia> map(x -> x 2 + 2x - 1, [1,3,-1])
3-element Array{Int64,1}:
2
14
-2
After the map() finishes, the function, and the argument x has disappeared:
julia> x
ERROR: x not defined
```

2.6 Dictionaries

In Julia dictionaries are represented as follows:

```
julia> dict = {"a" => 1, "b" => 2, "c" => 3}
Dict{Any,Any} with 3 entries:
"c" => 3
"b" => 2
"a" => 1
```

2.7 Data structures

Some of the Collections of data structures in Julia are Dequeues, Priority queues and Heap functions. These collections can be directly used in Julia programming.

Ex:

```
i)
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1 2 3 4 5 6
ii)
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6 5 4 3 2 1
iii)
julia> a = [1,3,4,5,2];

julia> Base.Collections.heapify(a)
5-element Array{Int64,1}:
 1 2 4 5 3
Some of the sorting algorithms are directly available in Julia:
Insertion sort, Quick sort and merge sort
Ex:
julia> sort([9,12,6,13,25,4,15,7,1,3,19],alg=InsertionSort)
11-element ArrayInt64,1:
 1 3 4 6 7 9 12 13 15 19 25
```

3. Differences with Julia from other languages

- Julia arrays are assigned by reference. After A=B, changing elements of B will modify A as well. If a function modifies an array, the changes will be visible in the caller. Julia discourages the used of semicolons to end statements.
- Functions in Julia return values from their last expression or the return keyword instead of listing the names of variables to return in the function definition
- Julia's $->$ operator creates an anonymous function, like Python.
- Julia performs matrix transposition using the .' operator and conjugated transposition using the'operator. Julia's A.' is therefore equivalent to R's t(A).

```
julia> a=[1  2;3  4]
2 2  Array{Int64,2}:
 1  2
 3  4

julia> a.'
2 2  Array{Int64,2}:
 1  3
 2  4
```

- Julia does not require parentheses when writing if statements or for/while loops: use for i in [1, 2, 3] instead of for (i in c(1, 2, 3)) and if i == 1 instead of if (i == 1).
- Julia has several functions that can mutate their arguments. For example, it has both sort() and sort!().
- Julia does not support the NULL type.
- In Julia, return does not require parentheses.
- In Julia, indexing of arrays, strings, starts from 1 not from 0.
- Julia's for, if, while etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python.
- #= indicates the start of a multiline comment, and =# ends it.