A Short and Incomplete Introduction to Julia

Part 1: Values and types

Riccardo Murri <riccardo.murri@uzh.ch>
S3IT: Services and Support for Science IT,
University of Zurich

September 12, 2019

Expressions

A Julia program consists of *expressions*.

An *expression* is a combination of constants, variables, operators, and functions that are interpreted to produce another value.

Examples of Julia language expressions are: '`2+2`', '`E = ones(3,3)`', '`print(42)`'.

Reference: https://en.wikipedia.org/wiki/Expression_(computer_science)

Lines of code

Julia expressions can be spread over multiple lines.

At the REPL, if you press the *Enter* key when an expression is not yet complete, Julia just continues reading. (You get no "`julia>`" prompt.)

You can combine multiple expressions in a single line with '`;`':

```
julia>   1+1; 2+2
4
```

(Note that only the last value is printed. *Why?*)

Strings and characters, I

Strings must be included in *double* quotes (`"`):

```
julia>   "This is a string"
"This is a string"
```

To include double quotes in a string,
escape them with '\':

```
julia>   print("\"Yes\", he said.")
He said: "42"
```

Strings and characters, II

Multi-line strings are delimited by three quote characters.

```
julia>   a = """This is a string,
     that extends over more
     than one line.
     """
```

Strings and characters, III

Single quotes (') delimit *characters*:

```
julia>   'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia>   'ε'
'ε': Unicode U+03b5 (category Ll: Letter, lowercase)
```

Note that it's an error to include more than one character between single quotes:

```
julia>   'abc'
ERROR: syntax: invalid character literal
```

Operators, I

All the usual unary and binary arithmetic operators are defined in Julia: `+`, `-`, `*`, `/`, `^` (exponentiation), `<<`, `>>`, etc.

Some operators are available in Julia which are not typically found in other languages: $\div$ for integer division, $\approx$ for approximate equality (`isapprox`), ...

Reference:

- https://docs.julialang.org/en/v1/manual/mathematical-operations/

Operators, II

Logical operators are expressed using symbols: `&&` for *and*, `||` for *or*, `!` for *not*.

Numerical and string comparison also follows the usual notation: `<`, `>`, `<=`, `==`, `!=`, . . . (Note that again "pretty" alternates are available: ⩽ for `<=`, ⩾ for `=>`, etc.)

Reference:

- https://docs.julialang.org/en/v1/base/base/#All-Objects-1

Your first exercise

How much is 2^{36} ?

And how much is 2^{72} ?

Getting the type of a value

The type of a Julia value can be gotten via the `typeof()` function:

```
julia>   typeof("a")
String

julia>   typeof(42)
Int64

julia>   typeof(NaN)
Float64
```

Common types (integer)

Types of common literal and variable values:

- **Int64** 64-bit Integer numbers: `1`, `-2`, ... up to `9223372036854775807`
- **Int32** 32-bit integer number: from `-2147483648` to `2147483647`.
- **UInt64** 64-bit non-negative *(unsigned)* integer: from `0` up to `18446744073709551614`.
- **UInt32** 32-bit non-negative *(unsigned)* integer.
- **BigInt** Unbounded integer.

Reference: https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/index.html

Common types (floating-point)

Types of common literal and variable values:

- **Float64** Double precision floating-point numbers, e.g.: `3.1415`, `-1e-3`.
- **Float32** Single precision floating-point numbers.
- **Float16** Half-precision floating-point numbers.
- **BigFloat** Arbitrary-precision floating-point number.

Reference: https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/index.html

Common types (misc)

Types of common literal and variable values:

Bool The type of the two boolean constants `true`, `false`.

Complex{...} A complex number; note there are different complex number types as there are (real) number types!

Rational{...} A rational number (quotient of two integers); note there are different `Rational` types as there are integer types!

String Text (string of UNICODE characters).

Literal values (generic)

When you enter values into the Julia REPL or a text file, the type is inferred using some simple criteria:

`1234`
`Int64`: all digits, no decimal dot ('`.`').

`0x1234`
`UInt8/16/32/64`: hexadecimal digits are read as unsigned integer.

`1 + 2im`
`Complex`: sum notation, the imaginary part has the suffix '`im`'.

`1//2`
`Rational`: a pair of integer numbers sparated by '`//`'.

The same format is used for printing back values.

Literal values (floating-point)

Floating-point types allow also some special non-numeric values:

`123.4` *or* `123.4e5`
`Float64`: all digits with a dot, or using the scientific notation with 'e' (`123.4e5` meaning 123.4×10^5)

`123.4f5`
`Float32`: scientific notation with 'f' to separate mantissa from exponent.

`Inf`
Infinity (with '+' or '−' sign); used to represent some limits (e.g., '`1. / 0.`')

`NaN`
"Not a number"; used to represent indeterminate mathematical operations (e.g., `0. / 0.`)

Note: **Letter case matters!** '`nan`' and '`inf`' are names of (probably unbound) variables, not the floating-point constants.

Type conversion, I

You can force Julia to convert a value into a compatible type by using the `convert` function.

```
julia>   convert(Float64, 1)
1.0

julia>   convert(Int64, 42.0)
42

julia>   convert(BigInt, 42)
42
julia>   typeof(ans)
BigInt
```

Type conversion, II

Alternatively, type names double up as conversion functions themselves:

```
julia>   Float64(1)
1.0

julia>   Int64(42.0)
42

julia>   BigInt(42)
42
julia>   typeof(ans)
BigInt
```

Assignment of variables, I

Assignment is done via the '=' operator:

```
julia>    x = 42
42
```

Note: **assignment is an expression** in Julia: it evaluates to the value being assigned to the variable. For example:

```
julia>    (x = 42) / 2
21.0
```

Assignment of variables, II

There are a few shortcut notations for updating an assigned variable:

`a += b` is short for `a = a + b`,

`a -= b` is short for `a = a - b`,

`a *= b` is short for `a = a * b`,

etc. — one for every legal operator.

For example:

```
julia>    x = 21;

julia>    x *= 2
42
```

(Note that '==' is the *equality comparison* operator, not an assignment operator!)

Exercise 1.B: How do you compute 2^{72} with Julia? (The result has to be an integer.)

Converting to and from String

You cannot cast a numeric value into a string or vice-versa using `convert`:

```
julia>   convert(String, 42)
ERROR: MethodError: Cannot `convert` an object of type Int64
to an object of type String

julia>   convert(Int64, "42")
ERROR: MethodError: Cannot `convert` an object of type String
to an object of type Int64
```

Converting *from* String

Converting a String into a value of another type is done by function `parse`:

```
julia> parse(Int64, "42")
42

julia> parse(Float32, "42")
42.0f0
```

Note that `parse` isn't for numbers only!

```
julia>   parse(Color, "black")
RGBN0f8(0.0,0.0,0.0)
```

Converting *to* String: interpolation

String interpolation means that character representation of values are substituted into a template string.

Julia allows substitution of values into strings using the '`$(...)`' syntax:

```
julia>   x = 42;


julia>   "$(x) is the ultimate answer!"
"42 is the ultimate answer!"
```

Note that *any Julia expression* be used inside '`$(...)`':

```
julia>   "$((x-21)*2) is the ultimate answer!"
"42 is the ultimate answer!"
```

Formatting output

Plain string interpolation does not allow you to control the printed format of numeric expressions (e.g., how many decimal digits).

Julia's standard library comes with the `@sprintf` macro (in the "Printf" module) which allows using C-style format specifiers:

```
julia>   using Printf  # load 'Printf' module
julia>   @sprintf("%03.5e", 12.3)
"1.23000e+01"
julia>   @sprintf("%03.5g", 12.3)
"12.3"
julia>   @sprintf("%03.5f", 12.3)
"12.30000"
julia>   @sprintf("%+8s %s %-8s", "welcome", "to", "julia")
" welcome to julia   "
```

See also: https://alvinalexander.com/programming/printf-format-cheat-sheet

The 'const' keyword

Outside of functions, variables can be marked as being *constant*: any redefinition or assignment of a different value to a 'const' variable is rejected with an error:

```
julia>    const FORTYTWO = 42;

julia>    FORTYTWO = 42.0
ERROR: invalid redefinition of constant FORTYTWO
```