A Julia

Julia is a scientific programming language that is free and open source. It is a relatively new language that borrows inspiration from languages like Python, MATLAB, and R. It was selected for use in this book because it is sufficiently high level so that the algorithms can be compactly expressed and readable while also being fast. This book is compatible with Julia version 1.0. This appendix introduces the necessary concepts for understanding the code included in the text.

A.1 Types

Julia has a variety of basic types that can represent data such as truth values, numbers, strings, arrays, tuples, and dictionaries. Users can also define their own types. This section explains how to use some of the basic types and define new types.

A.1.1 Booleans

The Boolean type in Julia, written Bool, includes the values true and false. We can assign these values to variables. Variable names can be any string of characters, including Unicode, with a few restrictions.

```
done = false
α = false
```

The left-hand side of the equal sign is the variable name, and the right hand side is the value.

We can make assignments in the Julia console. The console will return a response to the expression being evaluated.

1 Julia may be obtained from http://julialang.org.

2 In contrast with languages like C++, Julia does not require programmers to worry about memory management and other lower-level details.
julia> x = true
true
julia> y = false
false
julia> typeof(x)
Bool

The standard Boolean operators are supported.

julia> !x  # not
false
julia> x && y # and
false
julia> x || y # or
true

The # symbol indicates that the rest of the line is a comment and should not be evaluated.

A.1.2 Numbers

Julia supports integer and floating point numbers as shown here

julia> typeof(42)
Int64
julia> typeof(42.0)
Float64

Here, Int64 denotes a 64-bit integer, and Float64 denotes a 64-bit floating point value. We can also perform the standard mathematical operations:

julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
8
julia> x / y
2.0
julia> x ^ y
16

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
Note that the result of \( x \div y \) is a \texttt{Float64}, even when \( x \) and \( y \) are integers. We can also perform these operations at the same time as an assignment. For example, \( x += 1 \) is shorthand for \( x = x + 1 \).

We can also make comparisons:

\begin{verbatim}
 julia> 3 > 4
 false
 julia> 3 >= 4
 false
 julia> 3 >= 4  # unicode also works
 false
 julia> 3 < 4
 true
 julia> 3 <= 4
 true
 julia> 3 <= 4  # unicode also works
 true
 julia> 3 == 4
 false
 julia> 3 < 4 < 5
 true
\end{verbatim}

### A.1.3 Strings

A \textit{string} is an array of characters. Strings are not used very much in this textbook except for reporting certain errors. An object of type \texttt{String} can be constructed using \" characters. For example:

\begin{verbatim}
 julia> x = "optimal"
 "optimal"
 julia> typeof(x)
 String
\end{verbatim}

### A.1.4 Vectors

A \textit{vector} is a one-dimensional array that stores a sequence of values. We can construct a vector using square brackets, separating elements by commas. Semicolons in these examples suppress the output.
appendix a. julia

julia> x = []; # empty vector
julia> x = trues(3); # Boolean vector containing three trues
julia> x = ones(3); # vector of three ones
julia> x = zeros(3); # vector of three zeros
julia> x = rand(3); # vector of three random numbers between 0 and 1
julia> x = [3, 1, 4]; # vector of integers
julia> x = [3.1415, 1.618, 2.7182]; # vector of floats

An array comprehension can be used to create vectors. Below, we use the print function so that the output is printed horizontally.⁴

julia> print([sin(x) for x = 1:5])

[0.841471, 0.909297, 0.14112, -0.756802, -0.958924]

We can inspect the type of vectors:

julia> typeof([3, 1, 4]) # 1-dimensional array of Int64s
Array{Int64,1}

julia> typeof([3.1415, 1.618, 2.7182]) # 1-dimensional array of Float64s
Array{Float64,1}

We index into vectors using square brackets.

julia> x[1] # first element is indexed by 1
3.1415
julia> x[3] # third element
2.7182
julia> x[end] # use end to reference the end of the array
2.7182
julia> x[end - 1] # this returns the second to last element
1.618

We can pull out a range of elements from an array. Ranges are specified using a colon notation.

julia> x = [1, 1, 2, 3, 5, 8, 13];
julia> print(x[1:3]) # pull out the first three elements
[1, 1, 2]

julia> print(x[1:2:end]) # pull out every other element
[1, 2, 5, 13]

julia> print(x[end:-1:1]) # pull out all the elements in reverse order
[13, 8, 5, 3, 2, 1, 1]

We can perform a variety of different operations on arrays. The exclamation mark at the end of function names is often used to indicate that the function mutates (i.e., changes) the input.

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com

⁴ Print statements were used for compactness and are not needed.
A.1. Types

A.1.5 Matrices

A matrix is a two-dimensional array. Like a vector, it is constructed using square brackets. We use spaces to delimit elements in the same row and semicolons to delimit rows. We can also index into the matrix and output submatrices using ranges.
```julia
julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
julia> typeof(X)  # a 2-dimensional array of Int64s
Array{Int64,2}
julia> X[2]       # second element using column-major ordering
4
julia> X[3,2]    # element in third row and second column
8
julia> print(X[1,:])    # extract the first row
[1, 2, 3]
julia> print(X[:,2])    # extract the second column
[2, 5, 8, 11]
julia> print(X[:,1:2])  # extract the first two columns
[1 2; 4 5; 7 8; 10 11]
julia> print(X[1:2,1:2]) # extract a 2x2 matrix from the top left of X
[1 2; 4 5]

We can also construct a variety of special matrices and use array comprehensions:

julia> print(Matrix(1.0I, 3, 3))  # 3x3 identity matrix
[1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]
julia> print(Matrix(Diagonal([3, 2, 1]))) # 3x3 diagonal matrix with 3, 2, 1 on diagonal
[3 0 0; 0 2 0; 0 0 1]
julia> print(rand(3,2))          # 3x2 random matrix
[0.763487 0.560622; 0.544312 0.907828; 0.510258 0.207356]
julia> print(zeros(3,2))        # 3x2 matrix of zeros
[0.0 0.0; 0.0 0.0; 0.0 0.0]
julia> print([sin(x + y) for x = 1:3, y = 1:2]) # array comprehension
[0.909297 0.14112; 0.14112 -0.756802; -0.756802 -0.958924]

Matrix operations include the following:

julia> print(X')                     # complex conjugate transpose
[1 4 7 10; 2 5 8 11; 3 6 9 12]
julia> print(3X .+ 2) # multiplying by scalar and adding scalar
[5 8 11; 14 17 20; 23 26 29; 32 35 38]
julia> X = [1 3; 3 1];  # create an invertible matrix
julia> print(inv(X))  # inversion
[-0.125 0.375; 0.375 -0.125]
julia> det(X)            # determinant
-8.0
julia> print([X X])  # horizontal concatenation
[1 3 1 3; 3 1 3 1]
julia> print([X; X])  # vertical concatenation
[1 3; 3 1; 1 3; 3 1]
```

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
A.1.6 Tuples

A tuple is an ordered list of values, potentially of different types. They are constructed with parentheses. They are similar to arrays, but they cannot be mutated.

```julia
julia> x = (1,)  # a single element tuple indicated by the trailing comma
(1,)
julia> x = (1, 0, [1, 2], 2.5029, 4.6692)  # third element is a vector
(1, 0, [1, 2], 2.5029, 4.6692)
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
```

A.1.7 Dictionaries

A dictionary is a collection of key-value pairs. Key-value pairs are indicated with a double arrow operator. We can index into a dictionary using square brackets as with arrays and tuples.

```julia
julia> x = Dict();  # empty dictionary
julia> x[3] = 4  # associate value 4 with key 3
4
julia> x = Dict(3=>4, 5=>1)  # create a dictionary with two key-value pairs
Dict{Int64,Int64} with 2 entries:
  3 => 4
  5 => 1
julia> x[5]  # return value associated with key 5
1
julia> haskey(x, 3)  # check whether dictionary has key 3
true
julia> haskey(x, 4)  # check whether dictionary has key 4
false
```
A.1.8 Composite Types

A composite type is a collection of named fields. By default, an instance of a composite type is immutable (i.e., it cannot change). We use the struct keyword and then give the new type a name and list the names of the fields.

```julia
struct A
    a
    b
end
```

Adding the keyword mutable makes it so that an instance can change.

```julia
mutable struct B
    a
    b
end
```

Composite types are constructed using parentheses, between which we pass in values for the different fields. For example,

```julia
x = A(1.414, 1.732)
```

The double-colon operator can be used to annotate the types for the fields.

```julia
struct A
    a::Int64
    b::Float64
end
```

This annotation requires that we pass in an Int64 for the first field and a Float64 for the second field. For compactness, this text does not use type annotations, but it is at the expense of performance. Type annotations allow Julia to improve runtime performance because the compiler can optimize the underlying code for specific types.

A.1.9 Abstract Types

So far we have discussed concrete types, which are types that we can construct. However, concrete types are only part of the type hierarchy. There are also abstract types, which are supertypes of concrete types and other abstract types.

We can explore the type hierarchy of the Float64 type shown in figure A.1 using the supertype and subtypes functions.

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
Julia> supertype(Float64)
AbstractFloat
Julia> supertype(AbstractFloat)
Real
Julia> supertype(Real)
Number
Julia> supertype(Number)
Any
Julia> supertype(Any) # Any is at the top of the hierarchy
Any
Julia> subtypes(AbstractFloat) # different types of AbstractFloats
4-element Array{Any,1}:
    BigFloat
    Float16
    Float32
    Float64
Julia> subtypes(Float64) # Float64 does not have any subtypes
Type[]

We can define our own abstract types.

abstract type C end
abstract type D <: C end # D is an abstract subtype of C
struct E <: D # E is composite type that is a subtype of D
    a
end

A.1.10 Parametric Types

Julia supports parametric types, which are types that take parameters. We have already seen a parametric type with our dictionary example.

Julia> x = Dict(3=>4, 5=>1)
Dict{Int64,Int64} with 2 entries:
    3 => 4
    5 => 1

This constructs a Dict{Int64,Int64}. The parameters to the parametric type are listed within braces and delimited by commas. For the dictionary type, the first parameter specifies the key type, and the second parameter specifies the value type. Julia was able to infer this based on the input, but we could have specified it explicitly.
It is possible to define our own parametric types, but we do not do that in this text.

### A.2 Functions

A function is an object that maps a tuple of argument values to a return value. This section discusses how to define and work with functions.

#### A.2.1 Named Functions

One way to define a named function is to use the `function` keyword, followed by the name of the function and a tuple of names of arguments.

```julia
function f(x, y)
    return x + y
end
```

We can also define functions compactly using assignment form.

```julia
julia> f(x, y) = x + y;
```

```julia
julia> f(3, 0.1415)
3.1415
```

#### A.2.2 Anonymous Functions

An anonymous function is not given a name, though it can be assigned to a named variable. One way to define an anonymous function is to use the arrow operator.

```julia
julia> h = x -> x^2 + 1  # assign anonymous function to a variable
#1 (generic function with 1 method)
```

```julia
julia> g(f, a, b) = [f(a), f(b)];  # applies function f to a and b and returns array
```

```julia
julia> g(h, 5, 10)
2-element Array{Int64,1}:
  26
  101
julia> g(x->sin(x)+1, 10, 20)
2-element Array{Float64,1}:
  0.4559788891106302
  1.9129452507276277
```


A.2.3 Optional Arguments

We can specify optional arguments by setting default values.

```
 julia> f(x = 10) = x^2;
 julia> f()
 1
 julia> f(3)
 9
 julia> f(x, y, z = 1) = x*y + z;
 julia> f(1, 2, 3)
 5
 julia> f(1, 2)
 3
```

A.2.4 Keyword Arguments

Functions with keyword arguments are defined using a semicolon.

```
 julia> f(; x = 0) = x + 1;
 julia> f()
 1
 julia> f(x = 10)
 11
 julia> f(x, y = 10; z = 2) = (x + y)*z;
 julia> f(1)
 22
 julia> f(2, z = 3)
 36
 julia> f(2, 3)
 10
 julia> f(2, 3, z = 1)
 5
```

A.2.5 Function Overloading

The types of the arguments passed to a function can be specified using the double colon operator. If multiple functions of the same name are provided, Julia will execute the appropriate function.
julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.141500000000001
julia> f(1.3)
4.4415000000000004

The implementation of the most specific function will be used.

julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415

A.3 Control Flow

We can control the flow of our programs using conditional evaluation and loops. This section provides some of the syntax used in the book.

A.3.1 Conditional Evaluation

Conditional evaluation will check the value of a Boolean expression and then evaluate the appropriate block of code. One of the most common ways to do this is with an if statement.

```plaintext
define if
    if x < y
        # run this if x < y
    elseif x > y
        # run this if x > y
    else
        # run this if x == y
    end
```

We can also use the ternary operator with its question mark and colon syntax. It checks the Boolean expression before the question mark. If the expression evaluates to true, then it returns what comes before the colon; otherwise it returns what comes after the colon.
\texttt{julia}\> f(x) = x > 0 ? x : 0;
\texttt{julia}\> f(-10)
0
\texttt{julia}\> f(10)
10

\subsection*{A.3.2 Loops}

A loop allows for repeated evaluation of expressions. One type of loop is the while loop. It repeatedly evaluates a block of expressions until the specified condition after the \texttt{while} keyword is met. The following example will sum the values in array \( x \)

\begin{verbatim}
x = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while x != []
    s += pop!(x)
end
\end{verbatim}

Another type of loop is the for loop. It uses the \texttt{for} keyword. The following example will also sum over the values in the array \( x \) but will not modify \( x \).

\begin{verbatim}
x = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(x)
    s += x[i]
end
\end{verbatim}

The \texttt{=} can be substituted with \texttt{in} or \( \in \). The following code block is equivalent.

\begin{verbatim}
x = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in x
    s += y
end
\end{verbatim}

\subsection*{A.4 Packages}

A package is a collection of Julia code and possibly other external libraries that can be imported to provide additional functionality. Julia has a built-in package manager. A list of registered packages can be found at \url{https://pkg.julialang.org}. To add a registered package like \texttt{Distributions.jl}, we can run:
To update packages, we use:

```
Pkg.update()
```

To use a package, we use the keyword **using**:

```
using Distributions
```

Several code blocks in this text specify a package import with **using**. Some code blocks make use of functions that are not explicitly imported. For instance, the `var` function is provided by `Statistics.jl`, and the golden ratio $\varphi$ is defined in `Base.MathConstants.jl`. Other excluded packages are `InteractiveUtils.jl`, `LinearAlgebra.jl`, `QuadGK.jl`, `Random.jl`, and `StatsBase.jl`. 