Table of Contents

1
The Three Ms

This chapter is devoted to a discussion of Julia's 3-M's.

1. **Multiple Dispatch** - The feature in Julia which gives it with the ability to generate functional code specific to the parametric types of the arguments passed, which provides the speed equivalent to more conventionally compiled languages.
2. **Metaprogramming** - Dynamic features to extend Julia through the use of of "genuine" macros created at runtime; adding LISP-like functionally to the language .
3. **Modularity** - A relook at the packaging of type structures, functions and macros into modules which are used to compartmentalise individual areas.

We have met all of these before in the preceding chapters and will take the time here to discuss these in more detail.

Multiple Dispatch

Multiple dispatch is a feature of some programming languages in which a function or method can be dynamically dispatched based on the run-time (dynamic) type or, in the more general case some other attribute, of more than one of its arguments.

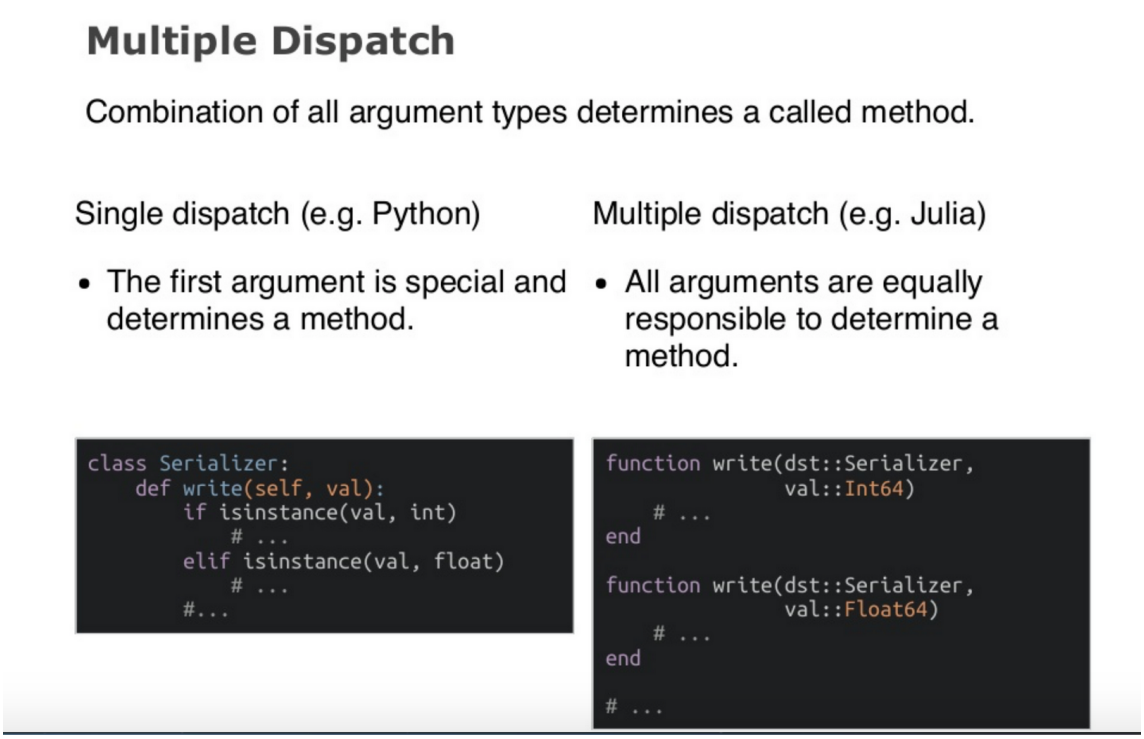
This is a generalisation of single-dispatch polymorphism where a function or method call is dynamically dispatched based on the actual derived type of the object on which the method has been called. Multiple dispatch routes the dynamic dispatch to the implementing function or method using the combined characteristics of one or more arguments.

In more conventional, i.e. single-dispatch object-oriented programming languages, when invoking a method - sending a message in Smalltalk, calling a member function in C++etc., - one of its arguments is treated specially and used to determine which of the (potentially many) methods of that name is to be applied.

In many languages, the special argument is indicated syntactically (for example, a number of programming languages put the special argument before a dot in making a method call). By contrast, in languages with multiple dispatch, the selected method is simply the one whose arguments match the number and type of the function call. There is no special argument that owns the function/method carried out in a particular call.

Function names are usually selected so as to be descriptive of the function's purpose. It is sometimes desirable to give several functions the same name, often because they perform conceptually similar tasks, but operate on different types of input data. In such cases, the name reference at the function call site is not sufficient for identifying the block of code to be executed. Instead, the number and type of the arguments to the function call are also used to select among several function implementations.

Because multiple dispatch occurs at runtime languages of the Lisp-style family represent some of the major examples: viz CLOS (Common Lisp Object System), Dylan, Clojure and Nice. In general these execute quite slowly but Julia, because of the LLVM compilation differs here producing something akin to compiled code (C/C++/Fortran) speeds



We saw in the previous chapter that the Julia type system consists of an hierarchy of abstract types, each terminated by concrete types which can have methods (functions) to operate on them. It is NOT possible to create a subtype of a concert one.

This allows Julia to implement a form of object orientation termed as delegation as opposed to the more familiar inheritance/polymorphic approach.

Delegation is simply passing a duty off to something else.

- Delegation can be an alternative to inheritance.
- Delegation means that you use an object of another class as an instance variable, and forward messages to the instance.
- It is better than inheritance for many cases because it makes you to think about each message you forward, because the instance is of a known class, rather than a new class, and because it doesn't force you to accept all the methods of the super class: you can provide only the methods that really make sense.
- Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate.
- The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.
  But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate `dynamic polymorphism`.

By the means of delegation Julia can create code for functions appropriate all the parameters and will generate different versions for individual instantiations without having too define operations on arrays (say), passing it off to the routines in file: array.jl

Let us define a trival function which returns the reciprocal of a numeric type as follows:

```
julia> recip(x::Number) =
  (x == zero(typeof(x))) ?
     error("Invalid reciprocal") :
        one(typeof(x)) / x;
julia> recip(2)
0.5
julia> recip(recip(2))  # recip does not ALWAYS return the same type
2.0
julia> recip(11//17)
17//11
julia> recip(11 + 17im)
0.02682926829268293 - 0.04146341463414634im
```

This simple definition can be applied to any type of argument for which the division operator applies without having to do anything more than a single line definition of

the **recip()** function.

The function is restricted to numeric types by use of the **::Number** annotation on the parameter and since there is the possibility of division by zero, the function checks for this using the **zero(typeof())** construct to allow for this.

Although we can pass simple numeric types the above definition will not work with arrays.

```
julia> aa = rand(3)
3-element Array{Float64,1}:
 0.576912912826387
 0.3440632929223615
 0.9025587904208758

julia> recip(aa)
MethodError: no method matching recip(::Array{Float64,1})
Closest candidates are:
  recip(!Matched::Number) at In[1]:1
Stacktrace:
 [1] top-level scope at In[7]:1
```

To extend **recip()** we need to define an additional form using a list comprehension or equivalently (in version 1.0) use of the **map()** function:

```
julia> recip(a::Array) = map(recip,a)
recip (generic function with 2 methods)

julia> recip(aa)
3-element Array{Float64,1}:
 1.7333638713351776
 2.906441984863674
 1.1079610664848614

# We can also map functions to an array
julia> map(sin,recip(aa))
3-element Array{Float64,1}:
 0.9868149731935016
 0.2329895103659116
 0.894790189217874

# Although the definition of recip() was in terms of a 1-D array ...
# ... we can still do this
julia> bb = [2.1 3.2 4.3; 9.8 8.7 7.6]
2×3 Array{Float64,2}:
 2.1 3.2 4.3
 9.8 8.7 7.6
```

```
julia> recip(bb)
2×3 Array{Float64,2}:
 0.47619  0.3125 0.232558
 0.102041 0.114943 0.131579

#... and this
julia> cc = recip(aa)'.*recip(bb)
2×3 Array{Float64,2}:
 0.825411 0.908263 0.257665
 0.176874 0.334074 0.145784
```

Code Generation

To investigate what is happening in Julia when running a function in the REPL, IDE or command it - it is easier to start with a very simple function, **`incr()`**, defined as :

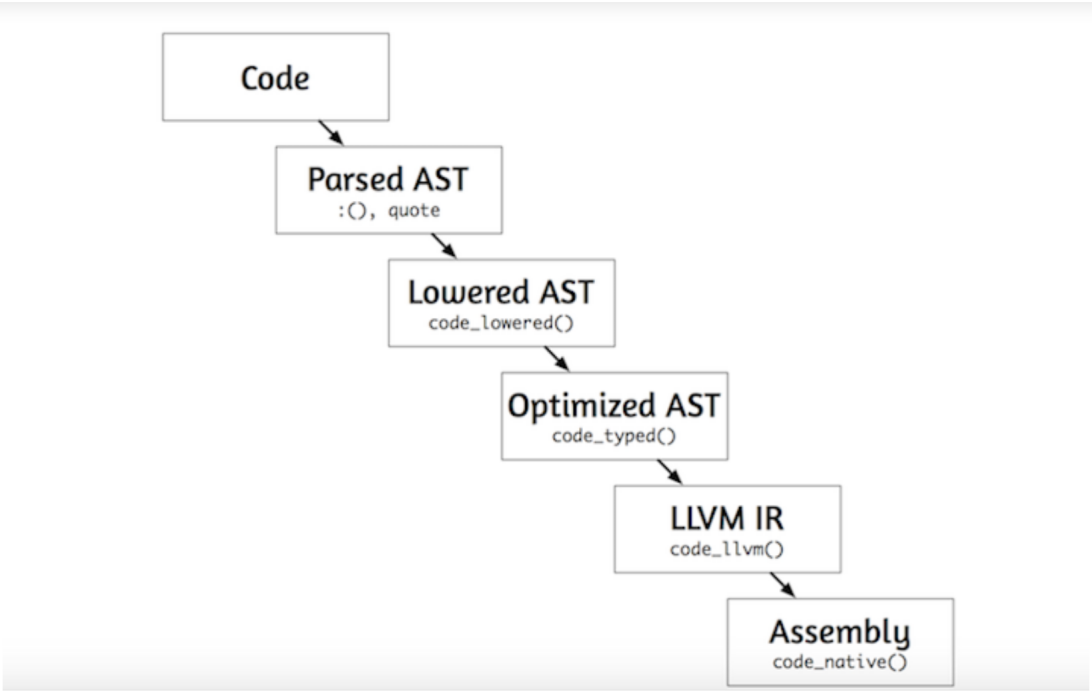
```
# A simple function to increment its argument
julia> incr(x) = x + 1

# Test it to see it works
julia> incr(2)
3
```

Julia parses the code to produce an abstract syntax tree (AST), which is akin to a functional representation such a one may get in more familiar languages such as LISP.

Julia then produces 3 intermediate representations

- **Lowered Code**: converts the AST into a representation which Julia can interpret, this regardless of the actual types of parameters, variables etc.
- **Typed Code**: now presents an implementation for a particular set of argument types after type inference and inlining
- **LLVM Code**: This transforms the typed code into standard LLVM internal representation (IR)

All these are the same regardless of the operating system of the platform on which Julia is running. At the LLVM (IR) stage it is possible for the specific LLVM compiler to kick-in and produce native code. The machine I am using is a MacPro with OSX and an Intel X86 processor, so naturally any native code I display will be in Intel x86 assembly languages, other processors will generate different code according to the LLVM backend.



Julia provides a set of routines to show the intermediate stages from the AST to native code, which are usually invoked by a set of macros.

First let us look at the native code for various types of parameters

```
# Increment an integer argument
julia> @code_native incr(2)
.section __TEXT,__text,regular,pure_instructions
; Function incr {
; Location: In[53]:1
; Function +; {
; Location: In[53]:1
        decl    %eax
        leal    1(%edi), %eax
;}
        retl
        nopw    %cs:(%eax,%eax)
;}
```

The **`decl / retl / nopw`** instructions are part of the general housekeeping of the x86 and all calls will have these.

The only specific instruction is a single call to adding 1 to an integer register **`leal`**

1(%edi), %eax

```
# Now look at the code for a real (float) number
julia> @code_native incr(2.7)
.section __TEXT,__text,regular,pure_instructions
; Function incr {
; Location: In[53]:1
        decl    %eax
        movl    $649920832, %eax        ## imm = 0x26BD0140
        addl    %eax, (%eax)
        addb    %al, (%eax)
; Function +; {
; Location: promotion.jl:313
; Function +; {
; Location: float.jl:395
        vaddsd  (%eax), %xmm0, %xmm0
;}}
        retl
        nop
```

This is a little longer, using float-point (extended) registers and a different addition instruction **`vaddsd (%eax), %xmm0, %xmm0`**

Note that the value *$649920832* is an internal representation of the real number 2.7 This is still quite short, however not all generated code is this compact. If we increment a rational (via the rational.jl module), we get:

```
julia> @code_native incr(2//7)
.section      __TEXT,__text,regular,pure_instructions
; Function incr {
; Location: In[53]:1
        pushl   %ebx
        decl    %eax
        subl    $16, %esp
        decl    %eax
        movl    %edi, %ebx
        decl    %eax
        movl    $451539248, %eax   ## imm = 0x1AE9F130
        addl    %eax, (%eax)
        addb    %al, (%eax)
        decl    %eax
        movl    %esp, %edi
        movl    $1, %edx
        calll   *%eax
        vmovups (%esp), %xmm0
        vmovups %xmm0, (%ebx)
        decl    %eax
        movl    %ebx, %eax
```

```
        decl    %eax
        addl    $16, %esp
        popl    %ebx
        retl
        nop
;}
```

And also for a complex argument; it is possible to spot the increment code for the real part embedded her:

```
julia> @code_native incr(2.0 + 7.0im)
      .section        __TEXT,__text,regular,pure_instructions
; Function incr {
; Location: In[53]:1
; Function +; {
; Location: complex.jl:304
; Function +; {
; Location: In[53]:1
        vmovsd  (%esi), %xmm0  ## xmm0 = mem[0],zero
        decl    %eat
        movl    $649924992, %eax ## imm = 0x26BD1180
        addl    %eax, (%eax)
        addb    %al, (%eax)
        vaddsd  (%eax), %xmm0, %xmm0
;}}
; Location: complex.jl:12
        decl    %eax
        movl    8(%esi), %eax
;}
        vmovsd  %xmm0, (%edi)
        decl    %eax
        movl    %eax, 8(%edi)
        decl    %eax
        movl    %edi, %eax
        retl
        nopw    %cs:(%eax,%eax)
;}
```

Recall that native code is the end of the (virtual) processing chain. First the AST has to be computed, then the code lowered, the data types inserted and finally the LLVM representation created.

To reiterate, this will be common on all machines, only the native code will be different.
The following is for **`incr()`** function for an real argument, the rest I'll leave to the reader.

```
# Using the dump function to view the parsed AST
julia> dump(:(incr(2.7)))
Expr
  head: Symbol call
  args: Array{Any}((2,))
```

```
    1: Symbol incr
    2: Float64 2.7
```

The expression has a head which refers to a call operation on the symbol incr (which is the function call) and the argument of 2.7

After this the three intermediate stages of compilation are shown below:

```
julia> @code_lowered incr(2.7)CodeInfo(
│1 1 ── %1 = x + 1
│   └────      return %1
)

julia> @code_typed incr(2.7)
CodeInfo(
│ │ │ +1 1 ── %1 = (Base.add_float)(x, 1.0)::Float64
│   └──── return %1
) => Float64

julia> @code_llvm(incr(2.7))
; Function incr
; Location: In[53]:1
define double @julia_incr_36746(double) {
top:
; Function +; {
; Location: promotion.jl:313
; Function +; {
; Location: float.jl:395
  %1 = fadd double %0, 1.000000e+00
;}}
  ret double %1
}
```

Viewing the intermediate code is principally for interest but we will see later an example where this is quite illustrating.

How intermediate code representation is presented has changed over different versions of Julia.

Whereas we are told there will be no breaking changes in version 1, clearly how code is compiled can change and even the LLVM generated.

At the time of writing what is generated by v1.0.1 is accurate

Metaprogramming

Julia is homoiconic which means that a program can be written symbolically in a way that it can be manipulated as data using the language. This adds great power in as much as the program can create genuine code as it is executing, modifying due to data types, circumstances occurring at runtime etc.

The ability of a programming language to be its own metalanguage is termed reflection and this is a valuable language feature to facilitate metaprogramming, popular in list processing languages such as LISP.

As remarked earlier is that, in compiling its code, problems of long execution times, which arise in most scripting languages, are not present in Julia.

Symbols and Expressions

Before discussing macros we need to understand the role of symbols and expressions in Julia.

The symbol is a type in Julia identified by a colon (:) prefix. The symbol for a variable

```
julia> :(x)
:x
```

For complex expressions, combining variable, constants and functions can also be converted to a symbol with the colon notation. An alternative representation is to enclose the code in a **`quote . . . end`** block.

```
julia> ex1 = :((x^2 + y^2 - 2*x*y)^0.5)
:(((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.5

julia> ex2 = quote
 (x^2 + y^2 - 2*x*y)^0.5
end
quote
    #= In[45]:2 =#
    ((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.5
end
```

If we instantiate the variables x and y then we can evaluate the expression.

This is *NOT* a function call, so values need to be known before hand

```
julia> x = 1.1; y = 2.5;
julia> eval(ex1)
```

```
1.4
```

Note that the **`@eval`** macro does ***NOT*** behave as the function call unless the expression is prefixed with '$'

```
julia> @eval ex1
:(((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.5)

julia> eval $ex1
# i.e. @eval $(ex) === eval(ex)
```

Although ex1 and ex2 are equivalent, the AST of the two expressions are slightly different; here it is for *ex1*, (we leave *ex2* to the reader)

We see that the first term in the AST is a **`call`** , followed an array of arguments, which may be other symbols corresponding to variables or functions and/or constants

```
julia> dump(ex1)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol ^
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol -
        2: Expr
          head: Symbol call
          args: Array{Any}((3,))
            1: Symbol +
            2: Expr
              head: Symbol call
              args: Array{Any}((3,))
                1: Symbol ^
                2: Symbol x
                3: Int64 2
            3: Expr
              head: Symbol call
              args: Array{Any}((3,))
                1: Symbol ^
                2: Symbol y
                3: Int64 2
        3: Expr
          head: Symbol call
          args: Array{Any}((4,))
            1: Symbol *
            2: Int64 2
            3: Symbol x
```

```
            4: Symbol y
      3: Float64 0.5
```

Dump(ing) the expression can get very verbose even for quite short , so using the Meta function **`show_sexpr`** outputs a *LISP* style S-expression version.

```
julia> Meta.show_sexpr(ex1)
(:call, :^, (:call, :-, (:call, :+, (:call, :^, :x, 2), (:call, :^, :y,
2)), (:call, :*, 2, :x, :y)), 0.5
```

We can more clearly how expressions are parsed into a series of symbols such as :call, :^, :x and constant values, i.e. 2, 0.5

In processing an expression (in a macro) we often need to identify the various components.

The following function(s) traverse an expression tree:

```
# Traversing a tree
# Catchall version, other than symbols or expressions
function traverse!(ex, symbols) end

# If ex is a symbol push it onto the tree
function traverse!(ex::Symbol, symbols)
  push!(symbols, ex)
end

# Main processing function.
# We need to distinguish between a :call and
# other arguments (recursively).
#
function traverse!(ex::Expr, symbols)
  if ex.head == :call # function call
    for arg in ex.args[2:end]
      traverse!(arg, symbols) # recursive
    end
  else
    for arg in ex.args
      traverse!(arg, symbols) # recursive
    end
  end
end

# Define a wrapper function around traverse! function(s)
# Define an empty symbols array and push any found.
# Notice the use of unique to prune the symbols array
$
julia> function traverse(ex::Expr)
  symbols = Symbol[]
```

```
  traverse!(ex, symbols)
  return unique(symbols)  # Don't output duplicates
end

# And apply it to one of the expressions above.
julia> traverse(ex1)
2-element Array{Symbol,1}:
 :x
 :y
```

Macros

Armed with an understanding of symbols and expressions we can now discuss creating macros.

The first macro is very simple - if an expression is passed it prints out its argument list, otherwise just returns.

```
# Check 'ex' is an expression, else just return it.
julia> macro pout(ex)
   if typeof(ex) == Expr
     println(ex.args)
   end
   return ex
 end

julia> x = 1.1; @pout x
1.1

# For an expression return its arguments and then evaluate it.
julia> @pout (x^2 + y^2 - 2*x*y)^0.5
Any[:^, :((x ^ 2 + y ^ 2) - 2 * x * y), 0.5]
1.4
```

The following is a slightly more complex macro, which executes a body of code, a number of times. Notice the construct **`$(esc(n))`** which is used to stop the macro creating a local copy of the value of 'n'. This is termed macro hygiene and I'll discuss it in a little more detail later.

```
julia> macro dotimes(n, body)
  quote
    for i = 1:$(esc(n))
      $(esc(body))
    end
  end
```

```
end

julia> @dotimes 3 print("Hi")
HiHiHi

julia> i = 0;
julia> @dotimes 3 [global i += 1; println(i*i)]
1
4
9
```

Recall the closure **`addOne()`** which we defined in the previous chapter; that coupled with this macro can be used for a more general incrementing function

```
julia> reset();
julia> @dotimes(3, global k = addOne())
julia> k
3
```

In the second of the examples above we need to specify that the variable 'i' is in global scope otherwise we will get three 1's printed

A more useful version of **`@dotimes`** macro is **`@until`** .

This creates a loop and breaks out when a condition fails.

```
macro until(condition, block)
   quote
      while true
          $(esc(block))
          if $(esc(condition))
             break
         end
     end
  end
end

julia> i = 0;
julia> @until (i >= 3) [global i += 1; println(i*i)]
1
4
9
```

Again we need to specify that 'i' is in global scope and because of the testing of the value of 'i', need to initialise it prior to calling the macro.

The **`@until`** macro can be used to implement a simple immediate if-then-

else construct **`iif`** - equivalent to the Julia build-in **`cond?body1:body2`** statement

```
macro iif(cond, body1, body2)
  :(if !$cond
     $(esc(body1))
   else
     $(esc(body2))
   end)
end
```

Lets test this by printing out a factorial; we will use the version in the stdlib SpecialFunctions, although the versions we have written previously could be use here too.

```
julia> using SpecialFunctions
julia> n = 10;
julia> @iff (n < 1) factorial(n)
             ArgumentError("$n not positive")
3628800

julia> n = -1;
julia> @iff (n < 1) factorial(n)
           ArgumentError("$n not positive")
ArgumentError("-1 not positive")
```

A Timing Macro

We have made use of macros to time the execution of code: these were **`@time`**, **`@elapsed`** and (from the BenchmarkTools package) **`@benchmark`**. To see how these operate we will formulate our own version

Before coding this we will define a function which executes very slowly but looking at the sum of Kempner series.

We know that the sum(1/n) diverges, but in a `Kempner sequence` all values of n containing a 9 are ignored, and this **does** converge to a the value of the sum is 22.92067 ..., but needs around 10^28 to even achieve a value over 22!

In the following function we use regular expressions to detect the terms to be ignored. If there is no match the function returns 'nothing' and it is these terms which we wish to include in the sum.

The Regex matchs values containing one or more 9's i.e 9, 19, ..., 91, .. 99, .. 109,

```
function kempner(n::Integer)
  @assert n > 0
```

```
  s = 0.0
  r9 = r"9"     # Match a string containing a 9
  for i in 1:n
    if (match(r9,string(i)) == nothing)
      s += 1.0/float(i)
    end
  end
  return s
end

# Now exercise the function
[kempner(10^i) for i in 1:7]
7-element Array{Float64,1}:
  2.8178571428571426
  4.78184876508206
  6.590720190283038
  8.223184402866208
  9.692877792106202
 11.015651849872553
 12.206153722565858
```

Even for 10^7 terms this is barely 50% for the known converged total.

Now we want to calculate the elapsed time for executing the **`kempner()`** function and will write our own macro **`@bmk`** to be it.

We pass a function name the first parameter and an integer, which is a number of times to execute the function. The different execution times are summed and the total averaged. Note that before the summing loop there is an initial call to the function, done so the any compilation time is not included in the sum.

```
# To see the basis for our timer we will write a
# modified version of @elapsed, called @bmk

macro bmk(fex, n::Integer)
  quote
    let s = 0.0
      if $(esc(n)) > 0
        val = $(esc(fex))
        for i = 1:$(esc(n))
          local t0 = Base.time_ns()
          local val = $(esc(fex))
          s += Base.time_ns() - t0
        end
        return s/($(esc(n)) * 10e9)
      else
        Base.error("Number of trials must be positive")
```

```
        end
      end
    end
  end
```

Notes:

1. We need to wrap the code in a let/end block so that the variable s is visible inside the loop even if called from the top-level in the REPL due to the crazy, new scoping rules.
2. The Base routine **`time_ns()`** returns the current clock time in nanoseconds.
3. We call the function **`$(esc(fex))`** escaping it for hygiene purposes *(see below)*
4. The function is executed (n+1) times, the first is ignored to ensure that compilation times are not taken into account.
5. The code returns the mean time, so is equivalent to the **`@elapsed`** macro but averaged over a number of trials.

```
# So now run it against the kemper function
@bmk kempner(10^7) 10
0.73997413854
```

For a discussion of the Kempner series see the arXiv.org paper:
ref: `https://arxiv.org/pdf/0806.4410.pdf`

The paper also discusses the Irwin series. This is where only terms the containing ONE 9 are selected i.e. 9, 19, 29, 90, 91, etc. , but ignoring but those 99, 909, and so on. This also converges to the sum: 23.04428 ... but even slower as there are many less terms to include in the summation.

The reader might like to formulate a function to compute the Irwin sequence, my version is included in the Jupyter notebook accompanying this chapter.

At first sight it would seem that these are the missing terms in the harmonic series and so it should diverge but it is not so as denominators such as 99, 909, 990, 991 etc are not in either series and these make the difference as they become more common later, i.e. for very large integers.

An exhausting approach needing in excess of 10^28 terms is not practicable even for Julia.

The paper above discusses ways that the sums of these series can be computed.

Macro Hygiene

Macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. This is normally done by creating local variables beginning with a '#' character, which would be illegal in standard Julia code since the hash character marks the start of a comment but is quite valid in intermediate representations

However expressions that are passed to a macro as arguments may be expected to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. So such variable should NOT be replaced by local copies.

We saw in our @bmk macro that values of the passed arguments can be copied into (local) variables but used of the '$' symbol, similar to the usage in strings an, as we will see in the next chapter, in tasks.

In fact we used the expression of the form $(esc(n)). This is because a second problem may occur with name clashes, consider the following code in the REPL as follows

```
import Base.@time
time() = ... # define a routine called time
@time time()
```

Clearly there is a difference between the **`time()`** routine and the **`@time`** macro as the former refers to **`Main.time()`**

So we need to ensure that code in any argument in **`@time`** is resolved in the macro call environment and this is the purpose of the escaping expression with esc() is defined in **`@time`** (as well as in **`@bmk`**) by :

`local val = $(esc(ex))` rather than just **`$ex`** ; the latter will usually work but it does not hurt to escape ex.

Macro Expansions

We have now seen how to write relatively simple macros but will be convenient, especially when debugging more complex macros, to see how the code is being generated.

Julia provides a function **`macroexpand()`** to do just this.

Let's start by looking at the simple case of **`@assert`**

```
julia> macroexpand(Main,:(@assert n > 0))
:(if n > 0
      nothing
  else
      (Base.throw)((Base.AssertionError)("n > 0"))
  end)
```

The code is pretty straight-forward. It checks a condition, specified by a combination of ALL the arguments and does nothing if the condition is met, otherwise it throws an assertion error, with the text constructed from the condition.

```
julia> n = -1; @assert n > 0
ERROR: AssertionError: n > 0
```

Expanding our **`@dotimes`** macro is also equally clear.

```
julia> macroexpand(Main,:(@dotimes 3
              [global i += 1; println(i*i)]))
 quote
    #= In[77]:3 =#
    for #10#i = 1:3
        #= In[77]:4 =#
        [global i += 1; println(i * i)]
    end
 end
```

We can see the 'local' values for the loop variable *#10#i* which is quite different from the global variable *i*. The macroexpand() function also provides block comments such as *#= In [77]:3 =#* to indicate the position of the code in the macro

Looking at expanding our **`@bmk`** macro, which is a little more complex with many more local variable but the function *kempner()* and loop count *10* are passed as is.

```
macroexpand(Main,:(@bmk kempner(10^7) 10))
quote
  #= REPL[1]:3 =#
  let #12#s = 0.0
    #= REPL[1]:4 =#
    if 10 > 0
      #= REPL[1]:5 =#
      #15#val = kempner(10 ^ 7)
      #= REPL[1]:6 =#
      for #13#i = 1:10
        #= REPL[1]:7 =#
        local #14#t0 = (Main.Base).time_ns()
        #= REPL[1]:8 =#
        local #15#val = kempner(10 ^ 7)
        #= REPL[1]:9 =#
```

```
          #12#s += (Main.Base).time_ns() - #14#t0
        end
        #= REPL[1]:11 =#
        return #12#s / (10 * 1.0e10)
      else
        #= REPL[1]:13 =#
        (Main.Base).error("Number of trials must be positive")
      end
    end
end
```

Not all macro expansions produce short boiler plate

Try expanding the following and examine the code produced.

```
julia> using Printf, Statistics
julia> aa = [rand() for i = 1:100000];
julia> @printf "The average value is %f over %d trials" mean(aa) length(aa)
The average value is 0.498691 over 100000 trials
```

Horner's algorithm for polynomial evaluation

Horner's method is used to reduce the evaluation of polynomial to the nth power to a series of (n-1) multiplications and n additions

$$
\begin{aligned}
P(x) &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \\
&= (a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1) x + a_0 \\
&= ((a_n x^{n-2} + a_{n-1} x^{n-3} + \cdots) x + a_1) x + a_0 \\
&= \underbrace{(\ldots((}_{n-1} \underbrace{a_n x + a_{n-1}}_{b_{n-1}}) x + \cdots) x + a_1) x + a_0
\end{aligned}
$$

The algorithm is shown in figure 4.3 (above) which comprises by continually nesting the terms from the nth term in the variable x, adding the next lower coefficient, multiplying all by x and so on until reaching the final coefficient.

The two snippets of code are for the 'conventional' power expansion **`poly_native()`** and

Horner method **`poly_horner()`**

```
# This is NOT the macro version
poly_native(x, a...)
  p=zero(x)
  for i = 1:length(a)
    p = p + a[i] * x^(i-1)
  end
  return p
 end

# Define a specific instance of poly_native
julia> f_native(x) = poly_native(x,1,2,3,4,5)
julia> f_native(2.1)
152.71450000000002

# The actual value is 152.7145, i.e. use
# round(152.71450000000002, digits=4)

# Neither IS this!
function poly_horner(x, a...)
  b = zero(x)
  for i = length(a):-1:1
    b = a[i] + b * x
  end
  return b
 end

# (x) -> (((5*x + 4)*x + 3)*x + 2)*x + 1
julia> f_horner(x) = poly_horner(x,1,2,3,4,5)
julia> round(f_horner(2.1),digits=4)
152.7145
```

Note in most languages, such as Python, and earlier versions of Julia, the second version (Horner's) executed faster than the former.

But in version 1 of Julia the code optimisation is so good that there is little difference in the elapsed times even for large polynomials.

Now lets consider a version to generate an Horner expansion by a macro. For this we will use a 'helper' function to multiply two numbers and add a third

```
# Define the 'helper function: mad(x,a,b)
# [In fact Julia has this function too as - muladd(x,a,b)]
julia> mad(x,a,b) = a*x + b;
julia> mad(2.1,5,4)
```

```
14.5
```

And NOW we can use **`mad()`** in a macro as:

```
# p is a variable list of arguments, passed in an array
macro horner(x, p...)
  ex = esc(p[end])
  for i = length(p)-1:-1:1
    ex = :(mad(t, $ex, $(esc(p[i]))))
  end
  Expr(:block, :(t = $(esc(x))), ex)
end

# Check this behaves as expected
# Notice that this works but using a different
# calling method for the macro.
julia> round(@horner(2.1,1,2,3,4,5), digits=4)
152.7145
```

We can look at the expansion of the @horner macro which is a series of nested mad function calls mimicking the form we saw above:

*(x) -> (((5*x + 4)*x + 3)*x + 2)*x + 1*

So our macro has eliminated the looping variable relegating it to just a series of function calls.

```
macroexpand(Main, :(@horner 2.1 1 2 3 4 5))
quote
    #36#t = 2.1
    (Main.mad)(#36#t, (Main.mad)(#36#t, (Main.mad)(#36#t, (Main.mad)(#36#t,
5, 4), 3), 2), 1)
end
```

Macrotools

Mike Innes' has authored a number of packages notable the Juno IDE and the machine learning package Flex (which we will be looking at in chapter 10). He is a great practitioner of the use of macros and, his package Macrotools has a useful set of macros and utility functions. I'll look at a couple of examples from the package here, but the code on Github will pay dividends for anyone wishing to engage in complex metacoding.

The postwalk function splits an expression into symbols and then reconstructs it, so we can apply different operations to each symbol

```
using MacroTools:postwalk
```

```
ex = :(1 + (2 + 3) + 4)
p = postwalk(ex) do x
   x isa Integer ? fac(x) : x
end
:(1 + (2 + 6) + 24)

# Evaluate the expression
eval(p)
33

map(x -> @show(x), [1,2,3,4]);
x = 1
x = 2
x = 3
x = 4

postwalk(ex) do x
 @show x
end
x = :+
x = 1
x = :+
x = 2x = 3
x = :(2 + 3)
x = 4
x = :(1 + (2 + 3) + 4)

@capture(ex, a_ + b_ + c_)
true

b
:(2+3)

a*eval(b) + c #=> 1*5 + 4
9

reduce(+, 1:10)
55
plus(a, b) = :($a + $b)
p = reduce(plus, 1:10)
:((((((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10)

eval(p)
55
```

Let us employ **`reduce()`** to do something useful. Below is the series expansion for the SINE function and again I'll use the factorial function from stdlib rather than our own versions.

The function is now in the SpecialFunctions module, which must then be included.

It is a good idea to include *using Printf, SpecialFunctions* etc., in a startup configuration file when working in the REPL.

We will see how to do this later in the book.

In a Jupyter notebook, earlier cells will contain the corresponding *'using'* statements

```
julia> using SpecialFunctions
julia> k = 2;
julia> pp = [:($((-1)^k) * x^$(1+2k) /
            $(factorial(1+2k))) for k = 0:5]
6-element Array{Expr,1}:
 :((1 * x ^ 1) / 1)
 :((-1 * x ^ 3) / 6)
 :((1 * x ^ 5) / 120)
 :((-1 * x ^ 7) / 5040)
 :((1 * x ^ 9) / 362880)
 :((-1 * x ^ 11) / 39916800)

# We can reduce this to a single expression
 reduce(plus,pp
:((((((1 * x ^ 1) / 1 + (-1 * x ^ 3) / 6) + (1 * x ^ 5) / 120) + (-1 * x ^
7) / 5040) + (1 * x ^ 9) / 362880) + (-1 * x ^ 11) / 39916800)

# ... and evaluate it for a specific value of x
julia> x = 2.1; eval(reduce(plus,pp)
0.8632069372306019
```

Lazy

Lazy is a more specialist module by Mike Innes, which uses of MacroTools to provides Julia with the cornerstones of functional programming - lazily-evaluated lists and a large library of functions for working with them. As with most the the packages Mike writes it is well worth a look.

For the unfamiliar, laziness just means that the elements of the list aren't actally calculated until you use them. This allows you to perform all sorts of magic, like working with infinite lists or lists of items from the future.

The following code scratches the surface: by using the **`@lazy`** macro we create a list of Fibonacci numbers and pick off first 15; because of lazy evaluation these are only evaluated

at the *take* time.

```
using Lazy
import Lazy: cycle, range, drop, take

julia> fibs = @lazy 0:1:(fibs + drop(1, fibs));
julia> take(15, fibs)
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
```

Lazy defines a set of macros which permit a functional style of writing:

```
# Pass the argument π/6 to a function sin and then onto exp.
@> π/6 sin exp  # ==> exp(sin(π/6))
1.6487212707001282
```

The **@>** macro can also have functional arguments.

In functional programming terminology this is terming currying, i.e. creating an intermediate function with some of the parameters defined and the remainder filled it a later stage

```
julia> f(x,μ) = -(x - μ)^2
julia> @> π/6 f(1.6) ex
0.3139129389863363
```

The **@>>** macro reverse the order of the arguments; let us use this to output the first 15 even squares

```
julia> esquares = @>> range() map(x -> x^2) filter(iseven);
julia> take(15, esquares)
(4 16 36 64 100 144 196 256 324 400 484 576 676 784 900)
```

We can use this macro to create a list of primes and then check if a number is itself a prime.

A helper function **takewhile()** defined in Lazy is required here:

```
isprime(n) =
  @>> primes begin
     takewhile(x -> x<=sqrt(n))
     map(x -> n % x == 0)
   any; !
end;

# We need to initialise the primes list
julia> primes = filter(isprime, range(2));

julia> isprime(113)
true
```

Generated Functions

Generated functions are defined by use of the macro @generated. They were introduced by Julia developers but the paradym is are now being adopted in a number of other language disciplines

GFs create specialized code depending on the types of their arguments with more flexibility and/or less code than what can be achieved with multiple dispatch, viz macros work with expressions at parse time are not able access the types of their inputs, a generated function gets expanded at a laate stage, at a time when the types of the arguments are known, but the function is not yet compiled.

Instead of performing some calculation or action, a generated function declaration returns a quoted expression which then forms the body for the method corresponding to the types of the arguments. When a generated function is called, the expression it returns is compiled and then run.

There are four points to note when using generated functions:

1. The function declaration is annotated with the @generated macro which adds information to the AST to inform the compiler that this is a generated function.
2. The body of the generated function has access to the types of the arguments but not their values but also any function that was defined before the definition of the GF.
3. The generated function returns a quoted expression rather than the result of some calculation; when evaluated, the expression performs the required computation.
4. Generated functions must not mutate or observe any non-constant global state, which means they can only read global constants, and cannot have any side effects. In functional parlance the must be completely pure. and currently, at least cannot define a closure.

Here is a simple generated function to execute a multiply + add operation which we met in the Horner algorithm; notice how the result returned is a symbol.

```
@generated function mad(a,b,c)
   Core.println("Calculating: a*b + c")
   return :(a * b + c)
 end

# Call the function
 julia> mad(2.3,1.7,1.1)
Calculating: a*b + c
5.01
```

And again with the same types of arguments; this time the function is not re-evaluated

```
julia> mad(2.3,1.7,2.1)
6.01
```

But with different arguments it IS evaluated again

```
julia> mad(2.3,1.7,1)
Calculating: a*b + c
4.91
```

Clearly this is not very useful over our previous **`mad()`** function.

To illustrate a more realistic use consider the following function which multiplies the size of dimensions of a n-D array

Here is a version using a *conventional* function:

```
function pdims(x::Array{T,N}) where {T,N}
  s = 1
  for i = 1:N
    s = s * size(x, i)
  end
  return s
end
pdims (generic function with 1 method)
```

... and then the generated function version

```
@generated function gpdims(x::Array{T,N}) where {T,N}
  ex = :(1)
  for i = 1:N
    ex = :(size(x, $i) * $ex)
  end
  return ex
end
gpdims (generic function with 1 method)
```

We need an array to test the two versions of the function and unsurprisingly they both produce *exactly* the same result

```
# We need an array to test the function
julia> aa  = [rand() for i = 1:1000];
julia> aax = reshape(aax,10,5,5,4); size(aax)
(10, 5, 5, 4)

julia> pdims(aax) == gpdims(aax)
true
```

And the difference? :- look at the lowered code:

```
@code_lowered pdims(aax)
CodeInfo(
│7  1 ──       s = 1
│8  │   %2  = 1:$(Expr(:static_parameter, 2))
│   │         #temp# = (Base.iterate)(%2)
│   │   %4  = #temp# === nothing
│   │   %5  = (Base.not_int)(%4)
│   └──       goto #4 if not %5
│   2 ?  %7  = #temp#
│   │         i = (Core.getfield)(%7, 1)
│   │   %9  = (Core.getfield)(%7, 2)
│9  │   %10 = s
│   │   %11 = (Main.size)(x, i)
│   │         s = %10 * %11
│   │         #temp# = (Base.iterate)(%2, %9)
│   │   %14 = #temp# === nothing
│   │   %15 = (Base.not_int)(%14)
│   └──       goto #4 if not %15
│   3 ──      goto #2
│11 4 ──      return s
)

@code_lowered gpdims(tax)
CodeInfo(
│┌ macro expansion16 1 ── %1 = (Main.size)(x, 4)
│╎    │   %2 = (Main.size)(x, 3)
│╎    │   %3 = (Main.size)(x, 2)
│╎    │   %4 = (Main.size)(x, 1)
│╎    │   %5 = %4 * 1
│╎    │   %6 = %3 * %5
│╎    │   %7 = %2 * %6
│╎    │   %8 = %1 * %7
│╎    └──      return %8
)
```

The latter is much more compact AND does not have the if/goto statements which naturally results in very different generated native code

```
@code_native dims(aax)
.section  _TEXT,__text,regular,pure_instructions
; Function pdims {
; Location: In[43]:7
  pushl       %eax
  decl        %eax
  movl        $4294967293, %ecx    ## imm = 0xFFFFFFFD
  movl        $1, %eax
```

```
  nopl        (%eax)
; Location: In[43]:9
; Function size; {
; Location: array.jl:154
L16:
  decl        %eax
  leal        4(%ecx), %edx
  decl        %eax
  cmpl        $4, %edx
  ja  L37
;}
; Function *; {
; Location: int.jl:54
  decl        %eax
  imull       48(%edi,%ecx,8), %eax
;}
; Function iterate; {
; Location: range.jl:575
; Function ==; {
; Location: promotion.jl:425
  decl        %eax
  testl       %ecx, %ecx
;}}
  je  L75
; Function size; {
; Location: array.jl:154
L37:
  decl        %eax
  leal        1(%ecx), %edx
  decl        %eax
  addl        $5, %ecx
  decl        %eax
  testl       %ecx, %ecx
  decl        %eax
  movl        %edx, %ecx
  jg  L16
  decl        %eax
  movl        $3802080, %eax          ## imm = 0x3A03E0
  addl        %eax, (%eax)
  addb        %al, (%eax)
  decl        %eax
  movl        $3794271536, %edi       ## imm = 0xE227FD30
  xchgl       %esp, %eax
  jg  L72
L72:
  addb        %bh, %bh
  rcrb        -61(%ecx)
  nopl        (%eax)
```

```
;}

@code_native gpdims(aax)
.section __TEXT,__text,regular,pure_instructions
; Function gpdims {
; Location: In[43]:16
; Function macro expansion; {
; Location: In[43]
; Function size; {
; Location: In[43]:16
  decl        %eax
  movl        40(%edi), %eax
;}}
; Function macro expansion; {
; Location: int.jl:54
  decl        %eax
  imull       48(%edi), %eax
  decl        %eax
  imull       32(%edi), %eax
  decl        %eax
  imull       24(%edi), %eax
;}
  retl
  nopw        %cs:(%eax,%eax)
;}
```

So generated functions have found to be especially useful in dealing with multidimensional arrays. You are advised to inspect the module **`Base.multidimensional.jl`** to see how generated functions are employed in practice.

Modularity

Julia code is organised into files, modules, and packages.

One or more modules can be stored in a package, and these are managed using the git version control system. Most Julia packages, including the official ones distributed by Julia, are stored on GitHub, where each package, conventionally, has with a ".jl" or ".jl.git" extension.

I'll be discussing what is involved in producing enterprise standard packages at the end of this book.

We saw in the first chapter that packages are managed by use of the *new* Julia package

manager (aka Pkg3) which was introduced in version 1.0 and via the use of the interactive shell mode how to add, update and remove them .

Note that there is a separate programmable API mode which can be used for similar operations

```
# Add the Gadfly visualisation package
using Pkg
Pkg.add("Gadfly")
```

The reader referred to the `online documentation` for a full discussion of the API and all other aspects of Pkg3

We can seen a few examples of Julia modules in preceding chapters but it is instructive to take a little time to focus on some general aspects of Julia modules.

- Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope which are are delimited syntactically, inside module name and the matching. end statement
- They allow you to create top-level definitions (i.e. global variables) without worrying about name conflicts when your code is used together with somebody else's.
- Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).
- There are three important standard modules: Main, Core, and Base.

Main is the top-level module, and Julia starts with Main set as the current module. Variables defined at the REPL prompt go in Main, and the function **`varinfo()`** lists variables in Main.

```
julia> mad(a,b,c) = a*b + c
julia> mad(2.3,1.7,2.1)
6.01

julia> varinfo()
name             size         summary
———————————————— ———————————— ————————
Base                          Module
Core                          Module
InteractiveUtils 157.769 KiB Module
Main                          Module
ans              8 bytes      Float64
mad              0 bytes      typeof(mad)
```

Core contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies a **`using Core`**, since nothing can be done without those definitions.

Base is a module that contains basic functionality (the contents of base/). All modules implicitly contain using Base, since this is required in the most situations.

In addition to using **Base**, modules also automatically contain definitions of the eval and include functions, which evaluate expressions/files within the global scope of that module.

If these default definitions are not wanted, modules can be defined using the keyword **baremodule** instead, although as mentioned above, **Core** is necessary and still imported.

The global variable *LOAD_PATH* contains the directories Julia searches for modules when calling require.

It can be extended using push!:

push!(LOAD_PATH, "/Users/malcoln/Julia/MyMods/")

Putting this statement in the file *~/.julia/config/startup.jl* will extend *LOAD_PATH* on every Julia startup.

Alternatively, the module load path can be extended by defining the environment variable *JULIA_LOAD_PATH.*

Modular Integers

In mathematics, modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" upon reaching a certain value—the modulus (plural moduli). The modern approach to modular arithmetic was developed by Carl Friedrich Gauss in his book Disquisitiones Arithmeticae, published in 1801.

If this seems unusual think of how we ass time. Hours are mod(60) as are minutes where as days are mod(24) or perhaps mod(12) depending on which clock configuration we use 24hr or 12hr.

Since v0.1 Julia distributed an examples folder which include a ModInt implementation; currently in v1.0 this has been dropped. It needs a little revision an so it s include below

Modular arithmetic (https://en.wikipedia.org/wiki/Modular_arithmetic) can be handled

mathematically by introducing a congruence relation on the integers that is compatible with the operations on integers: addition, subtraction, and multiplication. For a positive integer n, two numbers a and b are said to be congruent modulo n, if their difference a – b is an integer multiple of n (that is, if there is an integer k such that a – b = kn). This congruence relation is typically considered when a and b are integers, and is denoted

a ≡ b (mod n)

Operations suchs a s addition, subtraction and multiplication are possible between values having the same modulus. It is also possible to define an inverse function, i.e the value which when multiplied will give 1 mod(n) and using this it is possible to formulate a kind of division operator

Below is a Julia representation of modular integers

```
# Conventionally in a module code is not indented, ...
# ... otherwise ALL of it would be so.

module ModInts
export ModInt
import Base: +, -, *, /, inv
struct ModInt{n} <: Integer
  k::Int
  ModInt{n}(k) where n = new(mod(k,n))
end
Base.show(io::IO, k::ModInt{n}) where n =
 print(io, get(io, :compact, false) ? k.k : "$(k.k) mod $n")

(a::ModInt{n} + b::ModInt{n}) where n = ModInt{n}(a.k+b.k)
(a::ModInt{n} - b::ModInt{n}) where n = ModInt{n}(a.k-b.k)
(a::ModInt{n} * b::ModInt{n}) where n = ModInt{n}(a.k*b.k)
-(a::ModInt{n}) where n = ModInt{n}(-a.k)

inv(a::ModInt{n}) where n = ModInt{n}(invmod(a.k, n))
 (a::ModInt{n} / b::ModInt{n}) where n = a*inv(b)

Base.promote_rule(::Type{ModInt{n}}, ::Type{Int}) where n = ModInt{n}
 Base.convert(::Type{ModInt{n}}, i::Int) where n = ModInt{n}(i)

end
```

Notes:

1. The ModInt is defined as a struct and this includes a specific constructor
2. Basic operations such as +,-,*,/ as well as the inv() function are imported from Base so that they can be overloaded

3. Also a special show() routine is included which is used by Julia to display ModInt values
4. Promote and convert rules are included to deal with bare integers

```
# Test the module

julia> using Main.ModInts
julia> m1 = ModInt{11}(2)
julia> m2 = ModInt{11}(7)
julia> m3 = 3*m1 + m2
2 mod 11  ; # => 13 mod 11, i.e 2 mod 11
```

Because of multiple dispatch we can operate on arrays of modular integers and do the following :

```
julia> mm = reshape([ModInt{11}(rand(0:10))
                     for i = 1:100],10,10);
julia> ma = [ModInt{11}(rand(0:10))
                     for i = 1:10];
julia> mm.*ma'
10×10 Array{ModInt{11},2}:
  3  1  9   3   3  7   7   1  0   0
  2  6  1   2  10  9   2   0  0   2
  8  5  1   5   2  4   8   3  0   1
  1  1  4   5   7  8   1   8  0   0
  9  1  3   3   4  1   9   9  0   9
  8  1  3  10   8  7  10   4  0   5
 10  4  5   3   7  6   2   8  0   1
  3  9  3   7   8  4   9  10  0   4
  2  5  3   0   8  0   6   9  0  10
  9  9  0   5   0  4   7  10  0   7
```

Testing

A cornerstone of creating any production quality software is to have the ability to add some degree of test harness; this may be useful when:

- the software is finally completed
- it is modified, to implement further changes, address bugs etc.
- there are changes to the environment, e.g. new versions of the compiler or the operating system

Julia has a noodle in Base called Test, which furnish a set of macros to aid in the testing process

```
#=
In the simplest case the @test macro behaves in a similar fashion to
@assert, except it outputs Test Passed or Failed
=#
julia> using Test
julia> x = 1;
julia> @test x == 1
Test Passed

# When a test fails it is more verbose
# Indicating where the test fails and why
julia> @test x == 2
Test Failed at REPL[7]:1
 Expression: x == 2
 Evaluated: 1 == 2
ERROR: There was an error during testing
```

Other macros can be used, for example to test for argument, domain, bounds errors etc. In the following example we are testing that a specific error IS trapped.

```
# Generated an array of 10 elements ...
# ... and try to set the 11th
julia> a = rand(10);
julia> @test_throws BoundsError a[11] = 0.1
Test Passed
 Thrown: BoundsError

#=
The above is a bounds error, so if we check for a different error type we
still get the error report but this time the test did not succeed
=#
julia> @test_throws DomainError a[11] = 0.1
Test Failed at REPL[12]:1
 Expression: a[11] = 0.1
 Expected: DomainError
 Thrown: BoundsError
ERROR: There was an error during testing
```

The Test suite can also define a series of tests which can be executed as a whole. Below is an example to exercise some well known trigonometric formulae, in which I have deliberately got one wrong!

It is run by using a @testset macro, followed by a title (as a string) and a begin/end block containing normal Julia code and a set of @test macros.

```
julia> @testset "Trigonometric identities" begin
 θ = 2/3*π
 @test sin(-θ) ≈ -sin(θ)
```

```
 @test cos(-θ) ≈ -cos(θ)
 @test sin(2θ) ≈ 2*sin(θ)*cos(θ)
 @test cos(2θ) ≈ cos(θ)^2 - sin(θ)^2
 end;
trigonometric identities: Test Failed at REPL[16]:4
 Expression: cos(-θ) ≈ -(cos(θ))
 Evaluated: -0.4999999999999998 ≈ 0.4999999999999998
Test Summary: | Pass Fail Total
Trigonometric identities | 3 1 4
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

The report generated is helpful, identifying the test which failed:
cos(-θ) == +cos(θ), and not -cos(θ) as was specified

Ordered Pairs

For a more complex example, I will close this chapter by defining some operation on an "ordered pair".

This is a structure with two parameters (a, b) where the order does matter; some operations will be different when applied to the pair(b, a) as opposed to (a, b)

We want to ensure that the parameters of the ordered pair are numbers, not (say) string s, dates etc., and will wish to create a set of arithmetic operations, all of which need to be imported from Base.

Note that as well as the 'normal' functions such as +,-,*,/ etc., comparison rules such as ==, <, <= . . ., need to be defined and also a set of functions such as abs, zero, one etc. Not all now reside in Base, some (such as ***norm***) are in the LinearAlgebra module.

We also need to export the OrdPair definition

```
module OrdPairs

import Base: +,-,*,/,==,!=,>,<,>=,<=
import Base: abs,conj,inv,zero,one,show
import LinearAlgebra: transpose,adjoint,norm

export OrdPair
struct OrdPair{T<:Number}
  a::T
  b::T
end
OrdPair(x::Number) = OrdPair(x, zero(T))
value(u::OrdPair) = u.a
```

```
epsilon(u::OrdPair) = u.b
zero(::Type{OrdPairs.OrdPair}) =
                    OrdPair(zero(T),zero(T))
one(::Type{OrdPairs.OrdPair}) =
                    OrdPair(one(T),zero(T))
abs(u::OrdPair) = abs(value(u))
norm(u::OrdPair) = norm(value(u))

+(u::OrdPair, v::OrdPair) = OrdPair(value(u) + value(v), epsilon(u) +
epsilon(v))
-(u::OrdPair, v::OrdPair) = OrdPair(value(u) - value(v), epsilon(u) -
epsilon(v))
*(u::OrdPair, v::OrdPair) = OrdPair(value(u)*value(v), epsilon(u)*value(v)
+ value(u)*epsilon(v))
/(u::OrdPair, v::OrdPair) = OrdPair(value(u)/value(v),(epsilon(u)*value(v)
- value(u)*epsilon(v))/(value(v)*value(v)))

==(u::OrdPair, v::OrdPair) = norm(u) == norm(v)
!=(u::OrdPair, v::OrdPair) = norm(u) != norm(v)
>(u::OrdPair, v::OrdPair)  = norm(u) > norm(v)
>=(u::OrdPair, v::OrdPair) = norm(u) >= norm(v)
<(u::OrdPair, v::OrdPair) = norm(u) < norm(v)
<=(u::OrdPair, v::OrdPair) = norm(u) <= norm(v)

+(x::Number, u::OrdPair) = OrdPair(value(u) + x, epsilon(u))
+(u::OrdPair, x::Number) = x + u
-(x::Number, u::OrdPair) =
            OrdPair(x - value(u), epsilon(u))
-(u::OrdPair, x::Number) =
            OrdPair(value(u) - x, epsilon(u))
*(x::Number, u::OrdPair) =
            OrdPair(x*value(u), x*epsilon(u))
*(u::OrdPair, x::Number) = x*u
/(u::OrdPair, x::Number) = (1.0/x)*u

conj(u::OrdPair) =
             OrdPair(value(u),-epsilon(u))
inv(u::OrdPair) = one(OrdPair)/u
transpose(u::OrdPair) = transpose(uu::Array{OrdPairs.OrdPair,2}) = [uu[j,i]
for i=1:size(uu)[1],j=1:size(uu)[2]]

adjoint(u::OrdPair) = u
convert(::Type{OrdPair}, x::Number) =
                     OrdPair(x,zero(x))
promote_rule(::Type{OrdPair}, ::Type{<:Number}) = OrdPair

function show(io::IO,u::OrdPair)
  op::String = (epsilon(u) < 0.0)? " - ":" + ";
```

```
      print(io,value(u),op,abs(epsilon(u)),"ϵ")
    end
  end
```

One first sight this seems be a different representation of complex numbers but if we inspect the multiplication rule more closely:

(u1,v1) * (u2,v1) = (u1*u2, u1*v2 + u2*v1), which is quite a different form from that for complex arithmetic, viz. (u1*u2 - v1*v2, u1*v2 + u2*v1)

In fact these form the basis of a numeric type, termed a Dual Number.

Dual Numbers

$D(a,b) = a + b\epsilon$, where ϵ satisfies $\epsilon^2 = 0$

This is analogous to imaginary numbers, except $\epsilon^2 = 0$ instead of $i^2 = -1$

Or, think in terms of dropping higher order ($O(\epsilon^2)$) terms

In linear algebra, the dual numbers extend the real numbers by adjoining with the new element ε with the property that term in ε of order 2 or more are zero. Because of this convention the package above uses the functions **`value()`** and **`epsilon()`** to retrieve the first and second components of the number respectively.

With this property in mind we can list the normal rules of dual numbers which define its arithmetic operations:

Four rules of Dual numbers

$$(a + b\epsilon) \pm (c + d\epsilon) = (a + c) \pm (b + d)\epsilon$$

$$(a + b\epsilon) * (c + d\epsilon) = (ac) + (bc + ad)\epsilon$$

$$(a + b\epsilon)/(c + d\epsilon) = (a/c) + (bc - ad)/d^2\epsilon$$

Dual numbers were introduced in 1873 by William Clifford, and were used at the beginning of the twentieth century by the German mathematician Eduard Study, who used

them to represent the dual angle which measures the relative position of two skew lines in space. Another important application of dual numbers is that of automatic differentiation.

Lets define a couple of OrdPairs and first try to perform some simple arithmetic operations

```
julia> using Main.OrdPairs

julia> p1 = OrdPair(2.3,-1.7);
julia> p2 = OrdPair(4.4,0.9);

julia> p1 * p2
10.12 - 5.41ϵ

julia> p2/p1
1.9130434782608698 + 1.8052930056710779ϵ
```

Note how the **`show()`** function as defined produces much more readable output that the default one: **`OrdPair(10.13,-5.41)`**

Again we can operate on arrays of OrdPairs, without having to implement any additional code:

```
julia> using Statistics
julia> pp = [OrdPair(rand(),rand()) for i in 1:100];
julia> mean(pp)
0.48458715353415577 + 0.47958797125132224ϵ
```

And promote a rational (say) in a mixed operations

```
julia> p3 = OrdPair(2.3, 11/7)
2.3 + 1.5714285714285714ϵ
```

However the module described above is not a full implementation

```
julia> p4 = OrdPair(2.3, 11.0 + 7.2im)
MethodError: no method matching OrdPair(::Float64, ::Complex{Float64})
Closest candidates are:   OrdPair(::Number) at In[71]:16
OrdPair(::T<:Number, !Matched::T<:Number) where T<:Number at In[71]:12
```

Also a number of other arithmetic functions need to be imported from Base such as **`iterate()`**, and so on..

```
julia> std(p1)
MethodError: no method matching iterate(::OrdPair{Float64}) Closest
candidates are:   iterate(!Matched::Core.SimpleVector) at essentials.jl:578
iterate(!Matched::Core.SimpleVector, !Matched::Any) at essentials.jl:578
iterate(!Matched::ExponentialBackOff) at error.jl:171   ...
```

Complete implementations of dual numbers can become quite extensive.

The community group ***JuliaDiff*** as `one such` to which the reader is referred for further study.

Summary

In this chapter we have discussed three aspects of Julia which serve to make it stand out from other scripting languages.

First we looked at the idea of multiple dispatch. This is a relatively new paradigm within object oriented programming, very different from the more common polymorphic/inherence one which uses a single dispatch method. The advantage we saw was that it permitted Julia to compile specific, compact, well optimised code and in addition, when coupled with delegation, meant that there was no need to implement routines for (say) array operations, broadcasting and so on.

Secondly we discussed homoiconic representations of Julia code, in terms of symbols and expressions and how these can be instanted and then evaluated as part of the running process. This leads to a system for creating macros which can inject code into the program, which stands in place of (often) considerable boilerplate.

Finally we saw how Julia 'knits' all of these together in the form of modules, which extends the language by adding datatypes, functions, macros pertinent to particular aspects. A number of such modules are provided by Julia's base and standard library but the majority are from third parties, covering a variety of disciplines: statistics, analytics, visualisation, database etc.. We will be meeting some of the main ones in the later chapters of this book.

Index