Control Flow

Julia provides a variety of control flow constructs:

- Compound Expressions: `begin` and `;`.
- Conditional Evaluation: `if-elseif-else` and `?:` (ternary operator).
- Short-Circuit Evaluation: `&&`, `||` and chained comparisons.
- Repeated Evaluation: Loops: `while` and `for`.
- Exception Handling: `try-catch`, `error` and `throw`.
- Tasks (aka Coroutines): `yieldto`.

The first five control flow mechanisms are standard to high-level programming languages. `Task`s are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: `begin` blocks and `;` chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `begin` block:

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the `;` chain syntax comes in handy:

```
julia> z = (x = 1; y = 2; x + y)
3
```

This syntax is particularly useful with the terse single-line function definition form introduced in Functions. Although it is typical, there is no requirement that `begin` blocks be multiline or that `;` chains be single-line:

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;
        y = 2;
        x + y)
3
```

Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if-elseif-else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is `true`, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is `true`, the corresponding block is evaluated; if neither expression is true, the `else` block is evaluated. Here it is in action:

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
           elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end
       end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y
```

```
julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used. The condition expressions in the `if-elseif-else` construct are evaluated until the first one evaluates to `true`, after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

`if` blocks are "leaky", i.e. they do not introduce a local scope. This means that new variables defined inside the `if` clauses can be used after the `if` block, even if they weren't defined before. So, we could have defined the `test` function above as

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           else
               relation = "greater than"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

The variable `relation` is declared inside the `if` block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           end
           println("x is ", relation, " y.")
       end
```

```
test (generic function with 1 method)

julia> test(1,2)
x is less than y.

julia> test(2,1)
ERROR: UndefVarError: relation not defined
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

`if` blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so

```
julia> x = 3
3

julia> if x > 0
           "positive!"
       else
           "negative..."
       end
"positive!"
```

Note that very short conditional statements (one-liners) are frequently expressed using Short-Circuit Evaluation in Julia, as outlined in the next section.

Unlike C, MATLAB, Perl, Python, and Ruby – but like Java, and a few other stricter, typed languages – it is an error if the value of a conditional expression is anything but `true` or `false`:

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

This error indicates that the conditional was of the wrong type: `Int64` rather than the required `Bool`.

The so-called "ternary operator", `?:`, is closely related to the `if-elseif-else` syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

```
a ? b : c
```

The expression `a`, before the `?`, is a condition expression, and the ternary operation evaluates the expression `b`, before the `:`, if the condition `a` is `true` or the expression `c`, after the `:`, if it is `false`. Note that the spaces around `?` and `:` are mandatory: an expression like `a?b:c` is not a valid ternary expression (but a newline is acceptable after both the `?` and the `:`).

The easiest way to understand this behavior is to see an example. In the previous example, the `println` call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

If the expression `x < y` is true, the entire ternary operator expression evaluates to the string `"less than"` and otherwise it evaluates to the string `"not less than"`. The original three-way example requires chaining multiple uses of the ternary operator together:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

To facilitate chaining, the operator associates from right to left.

It is significant that like `if-elseif-else`, the expressions before and after the `:` are only evaluated if the condition expression evaluates to `true` or `false`, respectively:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"

julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the `&&` and `||` boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`.
- In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`.

The reasoning is that `a && b` must be `false` if `a` is `false`, regardless of the value of `b`, and likewise, the value of `a || b` must be true if `a` is `true`, regardless of the value of `b`. Both `&&` and `||` associate to the right, but `&&` has higher precedence than `||` does. It's easy to experiment with this behavior:

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)

julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
```

```
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true

julia> f(1) || f(2)
1
2
false
```

You can easily experiment in the same way with the associativity and precedence of various combinations of `&&` and `||` operators.

This behavior is frequently used in Julia to form an alternative to very short `if` statements. Instead of `if <cond> <statement> end`, one can write `<cond> && <statement>` (which could be read as: <cond> *and then* <statement>). Similarly, instead of `if ! <cond> <statement> end`, one can write `<cond> || <statement>` (which could be read as: <cond> *or else* <statement>).

For example, a recursive factorial routine could be defined like this:

```
julia> function fact(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * fact(n-1)
       end
fact (generic function with 1 method)

julia> fact(5)
120
```

```
julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fact(::Int64) at ./none:2
 [3] top-level scope
```

Boolean operations *without* short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: `&` and `|`. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

Just like condition expressions used in `if`, `elseif` or the ternary operator, the operands of `&&` or `||` must be boolean values (`true` or `false`). Using a non-boolean value anywhere except for the last entry in a conditional chain is an error:

```
julia> 1 && true
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the `while` loop and the `for` loop. Here is an example of a `while` loop:

```
julia> i = 1;

julia> while i <= 5
           println(i)
           global i += 1
       end
1
2
3
4
5
```

The `while` loop evaluates the condition expression (`i <= 5` in this case), and as long it remains `true`, keeps also evaluating the body of the `while` loop. If the condition expression is `false` when the `while` loop is first reached, the body is never evaluated.

The `for` loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above `while` loop does is so common, it can be expressed more concisely with a `for` loop:

```
julia> for i = 1:5
           println(i)
       end
1
2
3
4
5
```

Here the `1:5` is a range object, representing the sequence of numbers 1, 2, 3, 4, 5. The `for` loop iterates through these values, assigning each one in turn to the variable `i`. One rather important distinction between the previous `while` loop form and the `for` loop form is the scope during which the variable is visible. If the variable `i` has not been introduced in another scope, in the `for` loop form, it is visible only inside of the `for` loop, and not outside/afterwards. You'll either need a new interactive session instance or a different variable name to test this:

```
julia> for j = 1:5
           println(j)
```

```
        end
1
2
3
4
5

julia> j
ERROR: UndefVarError: j not defined
```

See Scope of Variables for a detailed explanation of variable scope and how it works in Julia.

In general, the `for` loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword `in` or `∈` is typically used instead of `=`, since it makes the code read more clearly:

```
julia> for i in [1,4,0]
           println(i)
       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]
           println(s)
       end
foo
bar
baz
```

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., Multi-dimensional Arrays).

It is sometimes convenient to terminate the repetition of a `while` before the test condition is falsified or stop iterating in a `for` loop before the end of the iterable object is reached. This can be accomplished with the `break` keyword:

```
julia> i = 1;

julia> while true
           println(i)
           if i >= 5
               break
```

```
            end
            global i += 1
        end
1
2
3
4
5

julia> for j = 1:1000
            println(j)
            if j >= 5
                break
            end
        end
1
2
3
4
5
```

Without the `break` keyword, the above `while` loop would never terminate on its own, and the `for` loop would iterate up to 1000. These loops are both exited early by using `break`.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The `continue` keyword accomplishes this:

```
julia> for i = 1:10
            if i % 3 != 0
                continue
            end
            println(i)
        end
3
6
9
```

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the `println` call inside the `if` block. In realistic usage there is more code to be evaluated after the `continue`, and often there are multiple points from which one calls `continue`.

Multiple nested `for` loops can be combined into a single outer loop, forming the cartesian product of its

iterables:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

With this syntax, iterables may still refer to outer loop variables; e.g. `for i = 1:n, j = 1:i` is valid. However a `break` statement inside such a loop exits the entire nest of loops, not just the inner one. Both variables (`i` and `j`) are set to their current iteration values each time the inner loop runs. Therefore, assignments to `i` will not be visible to subsequent iterations:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
           i = 0
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

If this example were rewritten to use a `for` keyword for each variable, then the output would be different: the second and fourth values would contain `0`.

Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program while printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances then allow that code to take the appropriate action.

Built-in `Exceptions`

`Exceptions` are thrown when an unexpected condition has occurred. The built-in `Exceptions` listed below all interrupt the normal flow of control.

`Exception`

- ArgumentError
- BoundsError
- CompositeException
- DimensionMismatch
- DivideError
- DomainError
- EOFError
- ErrorException
- InexactError
- InitError
- InterruptException
- InvalidStateException
- KeyError
- LoadError
- OutOfMemoryError
- ReadOnlyMemoryError
- RemoteException
- MethodError
- OverflowError
- Meta.ParseError
- SystemError
- TypeError
- UndefRefError

`UndefVarError`

`StringIndexError`

For example, the `sqrt` function throws a `DomainError` if applied to a negative real value:

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(C
Stacktrace:
[...]
```

You may define your own exceptions in the following way:

```
julia> struct MyCustomException <: Exception end
```

The `throw` function

Exceptions can be created explicitly with `throw`. For example, a function defined only for nonnegative numbers could be written to `throw` a `DomainError` if the argument is negative:

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must be nonnegative"))
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError with -1:
argument must be nonnegative
Stacktrace:
 [1] f(::Int64) at ./none:1
```

Note that `DomainError` without parentheses is not an exception, but a type of exception. It needs to be called to obtain an `Exception` object:

```
julia> typeof(DomainError(nothing)) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

Additionally, some exception types take one or more arguments that are used for error reporting:

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: x not defined
```

This mechanism can be implemented easily by custom exception types following the way `UndefVarError` is written:

```
julia> struct MyUndefVarError <: Exception
           var::Symbol
       end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

Note

When writing an error message, it is preferred to make the first word lowercase. For example,

`size(A) == size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`

is preferred over

`size(A) == size(B) || throw(DimensionMismatch("Size of A not equal to size of B"))`.

However, sometimes it makes sense to keep the uppercase first letter, for instance if an argument to a function is a capital letter:

`size(A,1) == size(B,2) || throw(DimensionMismatch("A has first dimension..."))`.

Errors

The `error` function is used to produce an `ErrorException` that interrupts the normal flow of control.

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the `sqrt` function that raises an error if its argument is negative:

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt(::Int64) at ./none:1
 [3] top-level scope
```

If `fussy_sqrt` is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

```
julia> function verbose_fussy_sqrt(x)
           println("before fussy_sqrt")
           r = fussy_sqrt(x)
           println("after fussy_sqrt")
           return r
       end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt at ./none:1 [inlined]
 [3] verbose_fussy_sqrt(::Int64) at ./none:3
 [4] top-level scope
```

The `try/catch` statement

The `try/catch` statement allows for `Exception`s to be tested for, and for the graceful handling of things that may ordinarily break your application. For example, in the below code the function for square root would normally throw an exception. By placing a `try/catch` block around it we can mitigate that here. You may choose how you wish to handle this exception, whether logging it, return a placeholder value or as in the case below where we just printed out a statement. One thing to think about when deciding how to handle unexpected situations is that using a `try/catch` block is much slower than using conditional branching to handle those situations. Below there are more examples of handling exceptions with a `try/catch` block:

```
julia> try
           sqrt("ten")
       catch e
           println("You should have entered a numeric value")
       end
You should have entered a numeric value
```

`try/catch` statements also allow the `Exception` to be saved in a variable. The following contrived example calculates the square root of the second element of `x` if `x` is indexable, otherwise assumes `x` is a real number and returns its square root:

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           end
       end
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])
2.0

julia> sqrt_second([1 -4])
0.0 + 2.0im

julia> sqrt_second(9)
3.0

julia> sqrt_second(-9)
```

```
ERROR: DomainError with -9.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(C
Stacktrace:
[...]
```

Note that the symbol following `catch` will always be interpreted as a name for the exception, so care is needed when writing `try/catch` expressions on a single line. The following code will *not* work to return the value of `x` in case of an error:

```
try bad() catch x end
```

Instead, use a semicolon or insert a line break after `catch`:

```
try bad() catch; x end

try bad()
catch
    x
end
```

The power of the `try/catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. Julia provides the `rethrow`, `backtrace`, `catch_backtrace` and `Base.catch_stack` functions for more advanced error handling.

`finally` Clauses

In code that performs state changes or uses resources like files, there is typically clean-up work (such as closing files) that needs to be done when the code is finished. Exceptions potentially complicate this task, since they can cause a block of code to exit before reaching its normal end. The `finally` keyword provides a way to run some code when a given block of code exits, regardless of how it exits.

For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

When control leaves the `try` block (for example due to a `return`, or just finishing normally), `close(f)` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `catch` block may be combined with `try` and `finally` as well. In this case the `finally` block will run after `catch` has handled the error.

Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. We mention them here only for completeness; for a full discussion see Asynchronous Programming.

« Functions Scope of Variables »

Powered by Documenter.jl and the Julia Programming Language.