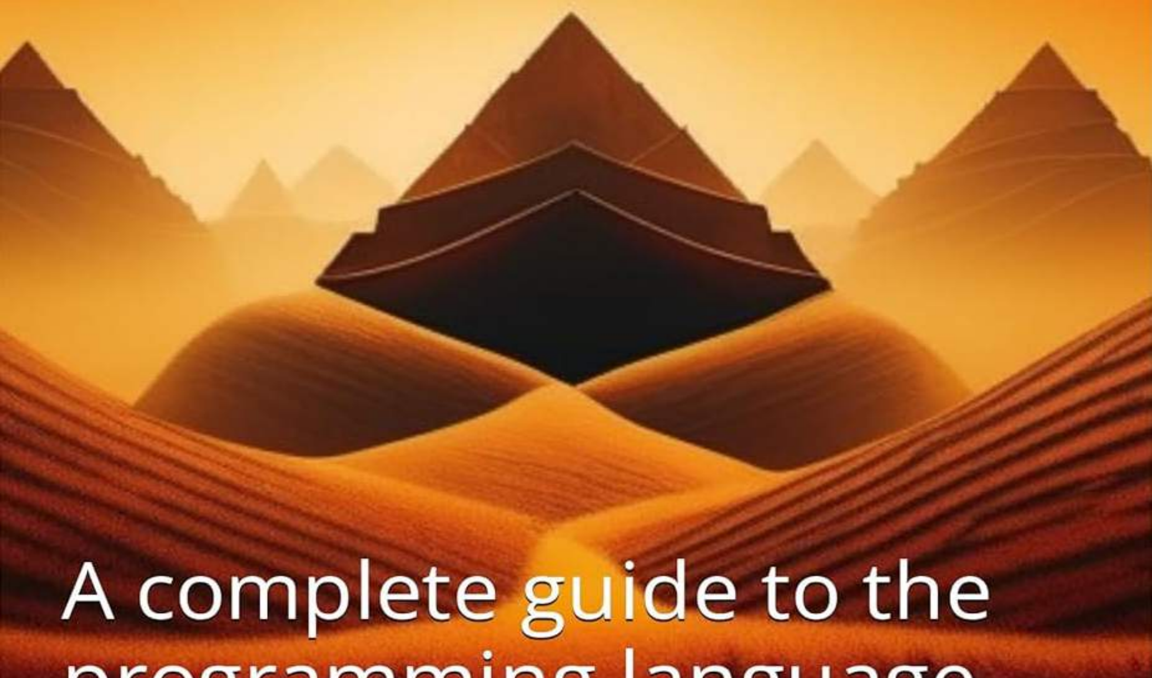
Mastering Nim



A complete guide to the programming language

Andreas Rumpf

Mastering Nim

A complete guide to the programming language

Andreas Rumpf

Mastering Nim 2.0

2nd edition, August 2023

(c) 2023 by Andreas Rumpf

This book is dedicated to Angelika, the love of my life.

Table of Contents

Acknowledgments

Parts of the book are based on Nim's manual [https://nim-lang.org/docs/manual.html] which is the work of many people over many years. I would like to thank everybody who made valuable contributions, such as proof-reading, catching spelling mistakes or pointing out mistakes or omissions in the documentation:

Ivan Bobev, Juan Carlos, Federico Ceratto, Neelesh Chandola, Timothee Cour, Huy Doan, Ico Doornekamp, Andrea Ferretti, Jasper Jenkins, James Johnson, Zahary Karadjov, Avahe Kellenberger, Lorenz Krakau, Andrey Makarov, Chris McIntyre, Kaushal Modi, Peter Munch-Ellingsen, Oscar Nihlgård, Rory O'Kane, Dominik Picheta, Andrey Riabushenko, Jacek Sieka, Mathias Stearn, Deansher Thompson, Miran Tuhtan, Elliot Waite, Brian Wignall, Zeshen Xing, Danil Yarantsev, Yanis Zafirópulos.

Special thanks go to Andre von Houck and Miran Tuhtan for reviewing this book. This book would not be as good as it is without your help.

I apologize to anybody that I have left out by accident.

Preface

Compiled languages are on the rise again. It's clear why, they generally produce programs that have better performance over interpreted languages and those programs are of higher quality due to static type checking. Nim goes a step beyond this new wave of compiled languages, by also giving you a powerful macro system to express complex problems.

Nim does not depend on any virtual machine, instead producing a binary that runs directly on the CPU — either with the help of an operating system or even without one. Every drop of performance can be squeezed out of your code should the need arise.

Nim's easy to use static type system catches mistakes early. The type system helps you ensure that a small typo will not break production and a big refactor is no longer scary. You don't even type that much, type inference is everywhere and generics make it very comfortable to use. Unlike in dynamic languages, every type is known at compile time and is properly optimized for both run time speed and memory space.

Meta programming and macros are really powerful and set Nim apart from other static compiled languages. Nim allows you to design deep and composable domain-specific languages with its powerful templates and macros.

Nim offers a "pluggable" memory management system called ARC giving you the required control for low latency, hard real-time systems without having to write "low level" programs. It is good for game and embedded development. ARC also integrates well with custom memory management.

Nim is a cross-platform language not tied to any operating system. It feels right at home on Linux, Mac and Windows, and can also be used to make mobile iOS and Android apps. Even on the web it can be used either as compiled to JavaScript or through WASM.

For embedded programming, the size of the produced machine code is important — Nim also shines in this aspect. The Nim compiler prunes unused code thanks to its focus on static binding needing no runtime introspection features. Where required, Nim's macro system can be used to provide the required introspection capabilities.

Productivity is also improved thanks to a well designed standard library and a large number of third party packages. The standard library covers topics such as hash tables, common algorithms like sorting or computing the edit distance, powerful collections, wrappers around operating systems, string parsing, unicode and time handling, implementations for the most common internet protocols, math and cryptography algorithms and much more.

Software becomes ever more complex and a modern programming language needs to reflect that to some extent. It cannot be "simple" anymore. Many features are taken for granted these days, and rightly so. They enable the construction of complex software much like sophisticated materials and tools enable the construction of skyscrapers. It follows that learning a language is a huge time investment. Nim rewards you with a single coherent language that can be used for everything and it works well on everything: It runs on web browsers, on virtually every operating system as well as on tiny embedded devices and even on GPUs. Nim's complexity is still very manageable, this book tries to cover Nim completely in about 300 pages.

Some describe Nim as a "better Python with types, macros and C's speed". The code you write is easy to understand by feeling like running pseudocode, thanks to its indentation-based syntax, short type names, and type inference. It makes people fall in love with programming again by presenting fast compile times, fast run times, minimal boilerplate and a comfortable language. But now enough of the praise, please dive in and be the judge!

History and theory behind Nim

Nim is a general-purpose language most inspired by Ada, Modula-3, C++, Python and Lisp. Its most important features are type and resource safety, meta programming and combining readability with syntactic convenience.

While Nim's primary focus is "imperative programming", it does support:

- Functional programming: Functions are first class entities and mutability can be restricted.
- Object oriented programming: Inheritance and dynamic binding are supported.
- Generic programming: Custom container types can be implemented easily and efficiently.
- Asynchronous programming: Leverage lightweight threads to support a large number of clients without blocking.
- Meta programming: Reflection over a program's structure is supported so powerful program transformations are possible. The transformations are carefully restricted to what can be done at compile-time. The restrictions also enforce that local reasoning about a program remains to be possible: The `macro` construct that enables the most powerful transformations cannot transform unrelated sections of the code.

This is the 2nd edition of "Mastering Nim". It describes version 2.0 of the Nim programming language, first released in 2023-08-01.

Who is this for

This book for people who can already program but wish to be able to program in Nim.

This book aims to give the reader a deep understanding of how Nim is layered and structured. It covers how the standard library APIs work and how they were implemented.

Structure of the book

This book is structured as follows: Part I is an introduction to Nim via simple algorithms based on graphical programming.

Part II is the largest part, it covers Nim in detail and in a formal language. It tries to convey how Nim really works and ideally you know the language inside out after reading it. This part originally evolved from Nim's official manual but please do not think that you know it all already! It contains plenty of unique and novel content. Sometimes examples show how the features are supposed to be used or why they exist, sometimes personal advice is provided of what to watch out for or to avoid.

Part III focuses on Nim's macro system. Examples are explained in detail and the examples were chosen to be representative of tasks where macros are an effective tool to improve the readability and the level of abstraction of your code.

Part IV covers Nim's multi-threading capabilities and memory model.

Essential parts of Nim's standard library are covered in the form of "cheat sheets". This might seem rather superfluous in the age of the Internet but the online documentation does not provide the information in this concise form.

The parts are loosely coupled and can be read independently from each other and in whatever order you choose.

Part I: Introduction to Nim via graphics

Chapter 1. Introduction

In this chapter we will show the basics of Nim language (types, for loops, if and case statements, procedures, etc.) while using basic graphic primitives to create shapes.

The premise is that the `putPixel` proc exists and we can use it, not worrying about its implementation details. All we need to know is: that proc takes the `x` and `y` coordinate of a point and its `Color` (default: white), and places a pixel of that color at those coordinates for us. This is how to use it:

```
import pixels  ①

putPixel(5, 9)         ②
putPixel(11, 18, Red)  ③
```

① We import `pixels` so we can use the `putPixel` proc.

② Puts a white (that is the default color) pixel at the `(5, 9)` coordinate.

③ Puts a red pixel at the `(11, 18)` coordinate.

Your Nim installation ships with a package manager called Nimble. Nimble can install additional packages. You can install the `pixels` library via:

```
nimble install pixels
```

Chapter 2. Drawing a line

A line is a series of adjacent points. We will limit ourselves to two types of lines:

Horizontal and vertical lines

When drawing these lines point by point, just one coordinate changes, while the other stays constant.

A point is an object consisting of `x` and `y` coordinates, and we define it and use it like this:

```
import pixels

type
  Point = object ①
    x: int       ②
    y: int

var p = Point(x: 5, y: 7) ③

putPixel(p.x, p.y) ④
```

① Defining a `Point` as an `object`.

② Coordinates `x` and `y` are integers.

③ Variable `p` is an instance of a `Point` with coordinates `(5, 7)`.

④ We can access fields of an object with a dot. In this case, we access the `x` and `y` coordinates of the point `p` with `p.x` and `p.y`.

2.1. Drawing horizontal and vertical lines

A line can be defined by a starting and an ending point, or by a starting point and its length and a direction. We'll show both approaches.

2.1.1. Drawing a line using one point, length, direction

We can define a line by its starting point, its length, and its direction. Having these two information, we can then draw a line pixel by pixel.

A line length is a positive integer (a zero would be just a point, not a line; a negative value for the length is physically impossible). A direction can be either horizontal or vertical.

This means that we will limit ourselves to always create lines from left to right (we specify the leftmost point of a line and then increase the `x` coordinate) or top to bottom (we increase the `y` coordinate).

We can start by creating two similar procs: one for drawing horizontal lines, and one for drawing vertical lines.

```
import pixels

type
  Point = object
    x: int
    y: int

proc drawHorizontalLine(start: Point; length: Positive) = ①
  for delta in 0 .. length:  ②
    putPixel(start.x + delta, start.y) ③

proc drawVerticalLine(start: Point; length: Positive) =
  for delta in 0 .. length:
    putPixel(start.x, start.y + delta) ④

let a = Point(x: 60, y: 40)

drawHorizontalLine(a, 50) ⑤
drawVerticalLine(a, 30)   ⑥
```

① The `length` is defined as Nim's built-in `Positive` type, which is a subtype of `int`, containing just positive numbers.

② The `p..q` is an (inline) iterator, which iterates from `p` to `q` (both ends are inclusive) in the ascending order, i.e. `p` must be smaller than `q` for the iteration to happen. In this case we're iterating from zero to `length`.

③ For a horizontal line, the `y` coordinate remains constant, and in each iteration step we draw a pixel one pixel to the right from the previous step.

④ Similarly, for a vertical line, the `x` coordinate stays constant, and we increase the `y` coordinate to draw a new pixel.

⑤ This will draw a horizontal line from `(60, 40)` to `(110, 40)`.

⑥ This will draw a vertical line from `(60, 40)` to `(60, 70)`.

We can have one proc, `drawLine`, which will draw both horizontal and vertical lines, depending on the provided user's parameter.

To model a set of a limited amount of values, we use the `enum` type. The benefits of this choice will be visible later, when we will see one of the features of Nim: exhaustiveness checking.

We can now extend the previous example with these lines:

```
type
  Direction = enum  ①
    Horizontal      ②
    Vertical

proc drawLine(start: Point; length: Positive; direction: Direction) = ③
  case direction  ④
  of Horizontal:
    drawHorizontalLine(start, length) ⑤
  of Vertical:
    drawVerticalLine(start, length)

drawLine(a, 50, Horizontal) ⑥
drawLine(a, 30, Vertical)
```

① A `Direction` is an enumeration (`enum`) with two possible values.

② The values of a `Direction` are: `Horizontal` and `Vertical`.

③ We now have an extra parameter of the `Direction` type.

④ This is a `case` statement. It is similar to `if` statement, but more powerful. Not only we get separate branches for each case, we also have exhaustiveness checking: we cannot by accident leave out some cases. Concretely, we must account for all the enum values.

⑤ We delegate the task of drawing lines to the previously defined procs.

⑥ This will draw the same lines as in the example before.

2.1.2. Drawing a line by specifying start and end

If we use two points to define a line, we don't have to know (or calculate) the line length, and the direction can be inferred from the relation of these two points: if they have the same `y` coordinate, we are drawing a horizontal line, and a vertical line between two points with the same `x` coordinate.

We will always iterate from a point with a smaller value of the changing coordinate, but we will allow our users to specify the points in any order. We can achieve this with *recursion*.

```
proc drawHorizontalLine(a, b: Point) = ①
  if b.x < a.x:
    drawHorizontalLine(b, a) ②
  else:
    for x in a.x .. b.x:
      putPixel(x, a.y) ③

proc drawVerticalLine(a, b: Point) =
  if b.y < a.y:
    drawVerticalLine(b, a)
  else:
    for y in a.y .. b.y:
      putPixel(a.x, y)

let
  p = Point(x: 20, y: 20)
  q = Point(x: 50, y: 20)
  r = Point(x: 20, y: -10)

drawHorizontalLine(p, q) ④
drawVerticalLine(p, r)   ⑤
```

① Even though we already have a proc of the same name, there is no overriding: they are different procs because they have different parameters.

② Notice the reversed order of the arguments.

③ The `x` is a changing variable, while the `y` remains constant.

④ Draws a horizontal line between `(20, 20)` and `(50, 20)`.

⑤ Since `r.y` is smaller than `p.y`, this will in turn call `drawVerticalLine(r, p)` and draw a vertical line between `(20, -10)` and `(20, 20)`.

Chapter 3. Rendering Text

The `pixels` library can do slightly more than just `putPixel`: it also offers a minimal `drawText` proc, to put letters and words on the screen. Its declaration looks like this:

```
proc drawText(x, y: int; text: string; size: int; color: Color)
```

We pass the following parameters to it:

`x` and `y`
: The coordinates of a bottom-left corner of the text we want to draw

`text`
: The text we want to render. It is of type `string`. We will look at strings in more depth in the next chapter, for now it is enough to know that `string` is a builtin type that is roughly a sequence of `char`, and a *string literal* can be written in double quotes, for example `"like this"`.

`size`
: This is a height of the text in pixels.

`color`
: Similarly to the `putPixel` proc, we can define the color of the text we're drawing.

The simplest way to call this proc is like this:

```
drawText 30, 40, "Welcome to Nim!", 10, Yellow
```

Notice that we didn't use the parentheses after the name of the function to enclose the list of arguments. This is equivalent to `drawText(10, 10, "Welcome to Nim!", Yellow)`, either style can be used.

For the next more interesting example we need the *dollar operator* `$` which can turn many types into its string representation, and the *concatenation operator* `&` which combines ("concatenates") two strings into one:

```
$12 == "12"  # convert an integer into a string
"abc" & "def" == "abcdef" # concatenate two strings into one
```

The following example produces 3 lines of text:

```
for i in 1..3:
  let texttodraw = "welcome to nim for the " & $i & "th time!"  ①
  drawtext 10, i*10, texttodraw, 8, Yellow   ②
```

① Creates a string (concatenated from three separate strings) and assigns it to the local variable `textToDraw`.

② Renders the text at position `(10, i*10)` where `i` is in one of the numbers in the `1..3` range, for each loop iteration.

Chapter 4. Sequences

We have said that a string is a sequence of characters. Nim also supports sequences, called `seq`, of an arbitrary type. For example, a sequence of integers is written with the notation `seq[int]`, and a sequence of `Point` is `seq[Point]`.

We want to be able to draw more than a single pixel. `putPixels` accomplishes that:

```
proc putPixels(points: seq[Point]; col: Color) =  ①
  for p in items(points):  ②
    pixels.putPixel p.x, p.y, col ③

putPixels(@[Point(x: 2, y: 3), Point(x: 5, y: 10)], Gold) ④
```

① `putPixels` takes a list of `Point`s.

② The `items` iterator allows us to iterate over the `points` parameter.

③ Every pixel we draw uses the same color `col`. We call the `putPixel` proc from the `pixels` module. As you can see, you can qualify an identifier with the module it was declared in. Sometimes this can improve the readability of your code.

④ We call our newly introduced `putPixels` proc with the `seq` `@[Point(x: 2, y: 3), Point(x: 5, y: 10)]`.

You can construct a sequence via `@[…]`. The empty sequence is `@[]`.

Sequences offer random access, the `i`'ith element can be accessed via `s[i]`. The indexing starts from 0. The same notation is available for `string`.

Chapter 5. Parameter passing and mutability

Every parameter in Nim is *immutable* unless it is declared as a `var` parameter. This means that the following code does not compile:

```
proc resetPointsToOrigin(points: seq[Point]) =
  for i in 0 ..< points.len:      ①
    points[i] = Point(x: 0, y: 0) ②
```

① We iterate over every index of `points` via an iterator that uses an operator symbol `..<`. The `..<` symbol indicates that the upper bound is exclusive which is exactly what we need since the indexing starts at 0.

② We then try to mutate `points[i]` and set its new value to the point `(x: 0, y: 0)`. But the compiler rejects this statement!

The compiler rejects the code because `points` is a parameter that can only be used for read accesses. This restriction helps us to write code that is easier to understand and scales better to larger programs and at the same time it helps the compiler to produce better machine code.

In order to be allowed to mutate `points` we need a `var` parameter:

```
proc resetPointsToOrigin(points: var seq[Point]) = ①
  for i in 0 ..< points.len:
    points[i] = Point(x: 0, y: 0) ②
```

① The `points` parameter is a `var seq`

② so the mutation is allowed.

If we now try to call `resetPointsToOrigin` with a seq constructor the compiler once again rejects our code:

```
resetPointsToOrigin @[Point(x: 2, y: 4)]
```

The reason is that a sequence constructed via `@[]` is not mutable. A variable is mutable, so the following is valid:

```
var points = @[Point(x: 2, y: 4)]
resetPointsToOrigin points
```

Chapter 6. Let vs Var

A variable can not only be introduced via `var` but also via `let`:

```
let points = @[Point(x: 2, y: 4)]
resetPointsToOrigin points
```

Such a `let` variable cannot be changed after its initialization so `resetPointsToOrigin points` is rejected too.

As programs grow larger, the enforced discipline about what can be mutated helps both compilers and human programmers to reason about the code. At the same time, the rules are not overly restrictive and don't prevent the development of classical or novel algorithms.

Chapter 7. Iterators

An `iterator` is a resumable `proc`. Iterators are usually invoked in `for` loops. We have already seen the built-in iterators `items` and `..<`. If they were not built-in we could easily define them ourselves:

```
iterator `..<`(a, b: int): int = ①
  var i = a
  while i < b:
    yield i ②
    inc i   ③

iterator items(s: seq[Point]): Point =
  for i in 0 ..< s.len: ④
    yield s[i]
```

① An `iterator` is declared much like a `proc`. The `..<` operator takes two integers and produces an integer.

② We do not `return` a value, we `yield` it. The for loop that calls the iterator `..<` calls `..<` again and again, each time the control flow resumes where the iterator left off until the iterator's `while` loop finishes. (When `i >= b`.)

③ `inc i` means to increment the integer `i` by 1. It can be also be written as `i = i + 1` or `i += 1`.

④ The `items` iterator calls the `..<` iterator. Iterators are called in `for` loops.

7.1. Yield

A `yield` statement can be easily understood as a variation of a `return` statement: A `return` statement returns the control flow to the caller, potentially producing a value that the caller can receive:

```
proc find(haystack: string; needle: char): int =
  for i in 0 ..< haystack.len:
    if haystack[i] == needle: return i ①
  return -1 ②

let index = find("abcabc", 'c') ③
```

① Return the value of `i` to the caller and do not continue with the execution of `find`. This implies that the `for` loop is left too.

② Return the value `-1` to indicate that `needle` did not occur in `haystack`.

③ We assign the value that `find` returns to a variable called `index`.

`find` returns the index of the first occurrence of `needle` inside `haystack`. It is not possible to *resume* its execution in order to retrieve the possible other occurrences of `needle`. An iterator like `findAll` can do that, thanks to the `yield` keyword:

```
iterator findAll(haystack: string; needle: char): int =
  for i in 0 ..< haystack.len:
    if haystack[i] == needle: yield i ①
  ②

for index in findAll("abcabc", 'c'): discard ③
```

① Return the value of `i` to the caller and continue with the execution of `findAll` later.

② Notice the absence of a `yield -1` statement. If the iterator does not `yield` more values, the calling `for` loop will stop.

③ We iterate over all values that are produced by `findAll` and bind the current value to `index`.

Chapter 8. Generics

The items iterator that we have just seen only works on seq[Point]. However, the code does not use any features of Point, it doesn't access point.x, for example. We really want to iterate over every seq[T] where T can be any type.

Nim supports such type variables via *generics*:

```
iterator items[T](s: seq[T]): T = ①
  for i in 0 ..< s.len:
    yield s[i]

for x in items(@[1, 2, 3]): discard ②
for x in items(@["1", "2", "3"]): discard ③
```

① The items iterator works for any type seq[T]. It produces values of type T.

② @[1, 2, 3] has type seq[int]. When items is called its type variable T is inferred to be int.

③ @["1", "2", "3"] has type seq[string]. When items is called its type variable T is inferred to be string.

Nim uses *specialization* for its generics. Every new concrete type like int or string produces specialized code, there is no runtime overhead.

Not only procs and iterators but also types can be generic:

```
type
  Point[T] = object ①
    x, y: T ②

var p: Point[float] ③
p = Point[float](x: 1.0, y: 3.0) ④
```

① The `Point` type is parametrized by a type variable `T`.

② `T` is used to declare the fields `x` and `y`.

③ A variable of name `p` is declared that is of type `Point[float]`

④ Object construction of `Point` also requires an explicit type; in this case `float`.

Unfortunately type inference does not work for object construction, `Point(x: 1.0, y: 3.0)` is not allowed. This restriction will probably be removed in the future.

If a type is parametrized by a type variable `T` operations on it usually have to be parametrized too. For example, our `drawHorizontalLine` proc would become:

```
proc drawHorizontalLine[T](a, b: Point[T]) =
  if b.x < a.x:
    drawHorizontalLine(b, a)
  else:
    for x in a.x .. b.x:
      putPixel(x, a.y)
```

`drawHorizontalLine` takes two parameters of the same type `Point[T]`. In other words, a call like `drawHorizontalLine(Point[float](x: 2.0, y: 3.0), Point[int](x: 2, y: 3))` would be rejected because one `T` cannot be both `float` and `int` at the same time.

We can use different type variables to allow for `drawHorizontalLine(Point[float](x: 2.0, y: 3.0), Point[int](x: 2, y: 3))`:

```
proc drawHorizontalLine[T, U](a: Point[T]; b: Point[U]) =
  if b.x < a.x:
    drawHorizontalLine(b, a)
  else:
    for x in a.x .. b.x:
      putPixel(x, a.y)
```

This assumes that we have an iterator `..` that can handle mixed types. We could provide such an iterator like this:

```
iterator `..`[T, U](a: T, b: U): U = ①
  var i = U(a) ②
  while i <= b: ③
    yield i
    inc i   ④
```

① Somewhat arbitrarily we have decided that the produced values are of type `U` and not of `T`.

② Via `U(a)` we convert the starting value `a` to type `U`. In Nim a *type conversion* looks like a function call.

③ We assume here that type `U` offers an operator `<=`. This assumption is not written down — generics in Nim can be under-specified.

④ We assume here that type `U` offers a suitable operation `inc`.

A type variable `T` is usually left under-specified in Nim; the requirements are only implicit and generic code is only type checked when the generic is instantiated:

```
for x in "a".."b": ... # invalid
for x in 0 .. 3: ... # valid
for x in 0 .. 3.0: ... # invalid because float does not have `inc`
```

Chapter 9. Templates

What happens when we call `putPixel` on coordinates that lie outside the screen's boundaries? That depends on the implementation of our `pixels` library but three outcomes are conceivable:

1. Nothing.
2. An exception is raised.
3. The program crashes.

In order to do "nothing" early returns can be a handy mechanism:

```
const
  ScreenWidth = 1024 ①
  ScreenHeight = 768

proc safePutPixel(x, y: int; col: Color) =
  if x < 0 or x >= ScreenWidth or
     y < 0 or y >= ScreenHeight:
    return ②
  putPixel(x, y, col) ③
```

① For simplicity, we assume a screen resolution of 1024x768 here. With `const` you can declare constants. A constant is comparable to a variable but its value cannot be changed and must be set at compile-time. The benefits of these restrictions will be explained later.

② If the coordinates are not within bounds `return`.

③ Else call the `putPixel` proc.

A program fragment like `if not inBounds(…): return` is common in graphics programming and so one can desire to move it into a helper proc. Unfortunately, a `return` statement leaves the current proc and so code like the following has not the desired effect:

```
proc boundsCheck(x, y: int) =
  if x < 0 or x >= ScreenWidth or
     y < 0 or y >= ScreenHeight:
    return ①

proc safePutPixel(x, y: int; col: Color) =
  boundsCheck(x, y)
  putPixel(x, y, col)
```

① The `return` statements leaves `boundsCheck` but not `safePutPixel`!

Nim offers a construct which has the “inlining” semantics that we need: A `template` is syntactically much like a `proc`, but an invocation to a `template` means to *expand* the `template`'s body at the call site:

```
template boundsCheck(a, b: int) = ①
  if a < 0 or a >= ScreenWidth or
     b < 0 or b >= ScreenHeight:
    return ②

proc safePutPixel(x, y: int; col: Color) =
  boundsCheck(x, y) ③
  putPixel(x, y, col)
```

① A `template` of name `boundsCheck` with parameters named `a` and `b` of type `int` is declared.

② `return` inside a `template` means to return from the `template’s caller.

③ A `template` can be invoked just like a `proc`.

Even though `boundsCheck(x, y)` looks like a call, it’s not called, instead `boundsCheck`'s body is inserted directly into `safePutPixel`. This insertion also does parameter substitutions; in our example the template parameter `a` is replaced by the procs parameter `x` and likewise is `b` replaced by `y`.

A template is a simple form of a macro, it is most commonly used for control flow abstractions and one can pass multiple statements to a template easily:

```
template wrap(body: untyped) = ①
  drawText 0, 10, "Before Body", 8, Yellow ②
  body ③

wrap: ④
  for i in 1..3:
    let textToDraw = "Welcome to Nim for the " & $i & "th time!"
    drawText 10, i*10, textToDraw, 8, Yellow
```

① The `wrap` templates takes a list of statements called `body`. The type `untyped` will be explained later.

② The `drawText` call runs

③ before the statements that are passed via `body` are run.

④ Via the syntax `wrap:` (note the colon) followed by the indented `for` loop we pass the `for` loop to the `wrap` template.

Even though templates are based on a conceptually quite simple substitution mechanism that is completely performed at compile-time, their power is surprising. With some experience they enable a programming style that lets us abstract away many details leading to shorter programs without negatively impacting the readability.

As an example we introduce a `withColor` environment. Inside this environment `putPixel` and `drawText` should use a specified color implicitly so that we don't have to repeat the color argument again and again. We declare variants of `putPixel` and `drawText` as templates that use an *undeclared* `colorContext` variable:

```
template putPixel(x, y: int) = putPixel(x, y, colorContext) ①
template drawText(x, y: int; s: string) = drawText(x, y, s, colorContext) ②
```

① The `putPixel` template which does not take a color delegates its work to the existing `putPixel` proc using the still undeclared `colorContext` color.

② Likewise does `drawText`.

Even though `putPixel` and `drawText` are already in our scope, it is valid to use the same names again for different (but in this case related) operations. The compiler performs a mechanism that is called *overload resolution* in order to disambiguate the invocations. In our case the disambiguation is simple: A call `putPixel(x, y, color)` resolves to `pixels.putPixel(x, y, color)`, whereas a call without a color parameter resolves to the newly introduced template of this name. The same applies for `drawText`.

Templates can easily refer to undeclared entities because only a template expansion implies that the result is checked for semantics.

The `colorContext` variable is declared inside the `withColor` environment. It is marked with `inject` so that it is visible inside the `body`:

```
template withColor(col: Color; body: untyped) = ①
  let colorContext {.inject.} = col ②
  body

withColor Blue: ③
  putPixel 3, 4 ④
  drawText 10, 10, "abc", 12
```

① `withColor` is a template that takes both a color and a body of code.

② `colorContext` is injected into `body`. Without the `.inject` annotation, `putPixel` and `drawText` would not be able to see the `colorContext` variable.

③ `blue` is passed to `col` and the code section `putPixel ⋯ drawText ⋯` to `body` via the colon syntax.

④ The `putPixel` and `drawText` templates are invoked.

After all templates are expanded the complete example looks like:

```
let colorContext = Blue
putPixel(3, 4, colorContext)
drawText(10, 10, "abc", colorContext)
```

Chapter 10. Macros

As we have seen templates can be used for tasks where ordinary procs fail. In some sense a template is more powerful than a proc. A macro is even more powerful than a template because a macro can inspect a `body` of code that is passed to it.

The first macro can be very hard to understand because you don't write the pattern that is expanded where the macro is invoked as you would for a template. Instead you write the *instructions* on how to create the pattern that is the code that is inserted where the macro is invoked.

In other words the code is rather imperative and low level.

The following example is a macro that turns the body of code that it receives into an empty list of statements (`newStmtList()`). This means that the body of code is ignored by the compiler:

```
import std / macros ①

macro disable(body: untyped): untyped = ②
  result = newStmtList() ③

disable: ④
  drawText(10, 10, "Disabled piece of code!", Blue)
```

① To work with macros an API is required, this API is in `std/macros`.

② The macro declaration looks like the template declaration except that the keyword `template` was replaced by `macro`.

③ The result of the macro is a syntax tree that is an empty list of statements.

④ To invoke a macro, the same syntax is used as for procs and templates.

Many more examples for macros can be found throughout the book but the better examples are too advanced for this introduction.

And with this our first tour through Nim ends. Time to dive into the foxhole!

Part II: Nim language specification

Chapter 11. Basic terms

Nim code specifies a computation that acts on a memory consisting of separate cells, called *locations*. A variable is basically a name for a location. Each variable and location is of a certain *type*. The variable's type is called *static type*, the location's type is called *dynamic type*. If the static type is not the same as the dynamic type, it is a super-type or subtype of the dynamic type.

An *identifier* is a symbol declared as a name for a variable, type, procedure, etc. The region of the program over which a declaration applies is called the *scope* of the declaration. Scopes can be nested. The meaning of an identifier is determined by the smallest enclosing scope in which the identifier is declared unless overloading resolution rules suggest otherwise.

An expression specifies a computation that produces a value or location. Expressions that produce locations are called *l-values*. An l-value can denote either a location or the value the location contains, depending on the context.

A *literal* is part of a Nim program that specifies an atomic part of a computation that is not a location, for example: the number `3` or the string `"abc"` are literals. Literals cannot be modified, a snippet like `12 = 4` (assign the value 4 to the value 12) is invalid.

A Nim *program* consists of one or more text *source files* containing Nim code. A Nim *processor* is a tool that can analyze and optionally transform Nim programs. One type of processor is called a *compiler*. A Nim compiler transforms Nim code into a format suitable for execution on a particular *target machine*.

> These definitions were inspired by the specification ("report") of Modula 3. The term *l-value* was taken from the specifications of C and C++.

Chapter 12. Lexical analysis

Lexical analysis describes how letters and other characters form the “words” of a programming language in general. These “words” are technically called *tokens* and the rest of the compiler’s transformation pipeline works on tokens and not on single characters. The transformation pipeline is usually called *phases of translation*.

A Nim compiler uses the following phases of translation:

1. Lexing: Turn a stream of characters into a stream of tokens.
2. Parsing: Turn a stream of tokens into an *abstract syntax tree* (AST).
3. Semantic analysis: Turn an AST to an annotated AST. The annotations are primarily type annotations as Nim is a statically typed language every expression needs to have a type.

This chapter describes in detail the first phase, *lexing*.

12.1. Notation used in this chapter

The language constructs are explained using an extended Backus–Naur form (EBNF), in which `a*` means 0 or more `a`'s, `a+` means 1 or more `a`'s, and `a?` means an optional `a` (either no `a` or one `a`). Parentheses may be used to group elements.

`&` is the lookahead operator; `&a` means that an `a` is expected but not consumed.

The `|`, `/` symbols are used to mark alternatives and have the lowest precedence. `/` is the ordered choice that requires the parser to try the alternatives in the given order. `/` is often used to ensure that the grammar is not ambiguous.

Non-terminals start with a lowercase letter, abstract terminal symbols are in UPPERCASE. Verbatim terminal symbols (including keywords) are quoted with `'`. An example:

```
ifStmt = 'if' expr ':' stmts ('elif' expr ':' stmts)* ('else' stmts)?
```

The binary `^*` operator is used as a shorthand for 0 or more occurrences separated by its second argument; likewise ^`` means 1 or more occurrences: ``a ^ b is short for `a (b a)*` and `a ^* b` is short for `(a (b a)*)?`. For example:

```
arrayConstructor = '[' expr ^* ',' ']'
```

A Nim program consists of one or more text source files containing Nim code. The text has to be encoded in UTF-8. Nim's grammar is not defined directly on the Unicode input text. Instead, it is defined on a list of separate, non-overlapping *tokens*. A token can be classified to be one of the following:

- A comment.
- An identifier.
- A keyword like `if` or `type`.
- A (character, string, integer, floating-point number) literal.
- An operator.
- A delimiter like `(`, `,`, `)`. (It covers the semicolon, the comma and the different types of brackets.)

12.2. Indentation

Nim's standard grammar describes an *indentation sensitive* language. This means that all the control structures are recognized by indentation. Indentation consists only of spaces; tabulators are not allowed.

> The "use tabs for indentation, spaces for alignment" rule never works sufficiently well in larger code-bases and even if it does work, it adds yet another source of friction for developers. Things are easier without such a rule. Since Nim uses an indentation based syntax and only allows spaces, the source code layout is portable across editors.

The indentation handling is implemented as follows: The lexer annotates the following token with the preceding number of spaces; indentation is not a separate token. This trick allows parsing of Nim with only one token of lookahead.

The parser uses a stack of indentation levels: the stack consists of integers counting the spaces. The indentation information is queried at strategic places in the parser but ignored otherwise: The pseudo-terminal `IND{>}` denotes an indentation that consists of more spaces than the entry at the top of the stack; `IND{=}` an indentation that has the same number of spaces. `DED` is another pseudo-terminal that describes the *action* of popping a value from the stack, `IND{>}` then implies to push onto the stack.

With this notation we can now define the core of the grammar: A block of statements (simplified example):

```
ifStmt = 'if' expr ':' stmt
          (IND{=} 'elif' expr ':' stmt)*
          (IND{=} 'else' ':' stmt)?

simpleStmt = ifStmt / ...

stmt = IND{>} stmt ^+ IND{=} DED  # list of statements
     / simpleStmt                 # or a simple statement
```

12.3. Comments

Comments start anywhere outside a string or character literal with the hash character (`#`). Comments consist of a concatenation of *comment pieces*. A comment piece starts with `#` and runs until the end of the line. The end of line characters belong to the piece. If the next line only consists of a comment piece with no other tokens between it and the preceding one, it does not start a new comment:

```
i = 0     # This is a single comment over multiple lines.
  # The scanner merges these pieces.
  # The same comment continues here.
```

Documentation comments are comments that start with two hash characters (`##`). Documentation comments are tokens; they are only allowed at certain places in the input file as they belong to the syntax tree.

12.4. Multiline comments

A multiline comment starts with `#[` and ends with `]#`:

```
#[Comment here that can
span
multiple lines.]#
```

Multiline comments can be nested:

```
#[  #[ Multiline comment in already
    commented out code. ]#
proc p[T](x: T) = discard
]#
```

Multiline documentation comments exist and support nesting too:

```
proc foo =
  ##[Long documentation comment
      here.
  ]##
```

12.5. Identifiers & Keywords

Identifiers in Nim can be any string of letters, digits and underscores, with the following restrictions:

- It has to begin with a letter.
- It is not allowed to end with an underscore `_`.
- Two successive underscores `__` are not allowed:

```
letter ::= 'A'..'Z' | 'a'..'z' | '\x80'..'\xff'
digit ::= '0'..'9'
IDENTIFIER ::= letter ( ['_'] (letter | digit) )*
```

Unicode characters with an ordinal value higher than 127 (non-ASCII) can be either classified as a `letter` or as `operator`. The details of this classification are not covered here as they might still change in the future.

The following keywords are reserved and cannot be used as identifiers:

```
addr and as asm
bind block break
case cast concept const continue converter
defer discard distinct div do
elif else end enum except export
finally for from func
if import in include interface is isnot iterator
let
macro method mixin mod
nil not notin
object of or out
proc ptr
raise ref return
shl shr static
template try tuple type
using
var
when while
xor
yield
```

Some keywords are currently unused; they are reserved for future developments of the language.

12.6. Identifier equality

Two identifiers are considered equal if the following algorithm returns true:

```
proc sameIdentifier(a, b: string): bool =
  a[0] == b[0] and
    a.replace("_", "").toLowerAscii == b.replace("_", "").toLowerAscii
```

That means only the first letters are compared in a case-sensitive manner. Other letters are compared case-insensitively within the ASCII range and underscores are ignored.

This rule also applies to keywords, meaning that `notin` is the same as `notIn` and `not_in`.

> The eccentric rules with respect to identifier equality try to ensure more sensible naming practices where different things have different names rather than only different spellings.

> Compare `const ROOT = root(Root)` to `const RootUser = rootof(RootDir)` to appreciate the benefits. Differences in spelling cannot be pronounced easily; as soon as two or more developers need to talk about their code-base the benefits of clearly distinctive names become apparent.

12.7. String literals

Terminal symbol in the grammar: `STR_LIT`.

String literals can be delimited by matching double quotes, and can contain the following *escape sequences* :

Table 1. Escape sequences for string literals

Escape sequence	Meaning
`\p`	platform specific newline: CRLF on Windows, LF on Unix
`\r`, `\c`	carriage return
`\n`, `\l`	line feed (often called newline)
`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
`\'0'..'9'+`	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\xHH`	character with hex value HH; exactly two hex digits are allowed

Escape sequence	Meaning
`\uHHHH`	unicode codepoint with hex value HHHH; exactly four hex digits are allowed
`\u{H+}`	unicode codepoint; all hex digits enclosed in `{}` are used for the codepoint

Strings in Nim may contain any 8-bit value, even embedded zeros.

12.8. Triple quoted string literals

Terminal symbol in the grammar: `TRIPLESTR_LIT`.

String literals can also be delimited by three double quotes `"""` ... `"""`. Literals in this form may run for several lines, may contain `"` and do not interpret any escape sequences. For convenience, when the opening `"""` is followed by a newline (there may be whitespace between the opening `"""` and the newline), the newline (and the preceding whitespace) is not included in the string. The ending of the string literal is defined by the pattern `"""[^"]`.

In other words, this:

```
""""long string within quotes""""
```

Produces:

```
"long string within quotes"
```

12.9. Raw string literals

Terminal symbol in the grammar: `RSTR_LIT`.

There are also raw string literals that are preceded with the letter `r` (or `R`) and are delimited by matching double quotes (just like ordinary string literals) and do not interpret the escape sequences. This is especially convenient for regular expressions or Windows paths:

```
var f = openFile(r"C:\texts\text.txt") # a raw string, so ``\t`` is no tab
```

To produce a single `"` within a raw string literal, it has to be doubled:

```
r"a""b"
```

Produces:

```
a"b
```

`r""""` is not possible with this notation, because the three leading quotes introduce a triple quoted string literal. `r"""` is the same as `"""` since triple quoted string literals do not interpret escape sequences either.

12.10. Generalized raw string literals

Terminal symbols in the grammar: `GENERALIZED_STR_LIT`, `GENERALIZED_TRIPLESTR_LIT`.

The construct `identifier"string literal"` (without whitespace between the identifier and the opening quotation mark) is a generalized raw string literal. It is a shortcut for the construct `identifier(r"string literal")`, so it denotes a routine call with a raw string literal as its only argument. Generalized raw string literals are especially convenient for embedding mini languages directly into Nim (for example regular expressions).

The construct `identifier"""string literal"""` exists too. It is a shortcut for `identifier("""string literal""")`.

12.11. Character literals

Character literals are enclosed in single quotes `''` and can contain the same escape sequences as strings - with one exception: the platform dependent *newline* (`\p`) is not allowed as it may be wider than one character (it can be the pair CR/LF). Here are the valid *escape sequences* for character literals:

Table 2. Escape sequences for character literals

Escape sequence	Meaning
`\r`, `\c`	carriage return
`\n`, `\l`	line feed
`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
`\'0'..'9'+`	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\xHH`	character with hex value HH; exactly two hex digits are allowed

A `char` is not a Unicode character but a single byte.

A character literal that does not end in `'` is interpreted as `'` if there is a preceding backtick token. There must be no whitespace between the preceding backtick token and the character literal. This special case ensures that a declaration like ``proc 'customLiteral`(s: string)`` is valid. ``proc 'customLiteral`(s: string)`` is the same as ``proc '\''customLiteral`(s: string)``.

See also Section 12.12.1, “Custom Numeric Literals”.

12.12. Numeric Literals

Numeric literals have the form:

```
hexdigit = digit | 'A'..'F' | 'a'..'f'
octdigit = '0'..'7'
bindigit = '0'..'1'
unary_minus = '-' # See the section about unary minus
HEX_LIT = unary_minus? '0' ('x' | 'X' ) hexdigit ( ['_'] hexdigit )*
DEC_LIT = unary_minus? digit ( ['_'] digit )*
OCT_LIT = unary_minus? '0' 'o' octdigit ( ['_'] octdigit )*
BIN_LIT = unary_minus? '0' ('b' | 'B' ) bindigit ( ['_'] bindigit )*

INT_LIT = HEX_LIT
        | DEC_LIT
        | OCT_LIT
        | BIN_LIT

INT8_LIT = INT_LIT ['\''] ('i' | 'I') '8'
INT16_LIT = INT_LIT ['\''] ('i' | 'I') '16'
INT32_LIT = INT_LIT ['\''] ('i' | 'I') '32'
INT64_LIT = INT_LIT ['\''] ('i' | 'I') '64'

UINT_LIT = INT_LIT ['\''] ('u' | 'U')
UINT8_LIT = INT_LIT ['\''] ('u' | 'U') '8'
UINT16_LIT = INT_LIT ['\''] ('u' | 'U') '16'
UINT32_LIT = INT_LIT ['\''] ('u' | 'U') '32'
UINT64_LIT = INT_LIT ['\''] ('u' | 'U') '64'

exponent = ('e' | 'E' ) ['+' | '-'] digit ( ['_'] digit )*
FLOAT_LIT = unary_minus? digit (['_'] digit)* (('.' digit (['_'] digit)*
[exponent]) |exponent)
FLOAT32_SUFFIX = ('f' | 'F') ['32']
FLOAT32_LIT = HEX_LIT '\'' FLOAT32_SUFFIX
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\'']
FLOAT32_SUFFIX
FLOAT64_SUFFIX = ( ('f' | 'F') '64' ) | 'd' | 'D'
FLOAT64_LIT = HEX_LIT '\'' FLOAT64_SUFFIX
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\'']
FLOAT64_SUFFIX

CUSTOM_NUMERIC_LIT = (FLOAT_LIT | INT_LIT) '\'' CUSTOM_NUMERIC_SUFFIX

# CUSTOM_NUMERIC_SUFFIX is any Nim identifier that is not
# a pre-defined type suffix.
```

As can be seen in the productions, numeric literals can contain underscores for readability. Integer and floating-point literals may be given in decimal (no prefix), binary (prefix `0b`), octal (prefix `0o`), and hexadecimal (prefix `0x`) notation.

The fact that the unary minus `-` in a number literal like `-1` is considered to be part of the literal is a late addition to the language. The rationale is that an expression `-128'i8` should be valid and without this special case, this would be impossible — `128` is not a valid `int8` value, only `-128` is.

For the `unary_minus` rule there are further restrictions that are not covered in the formal grammar. For `-` to be part of the number literal, its immediately preceding character has to be in the set `{' ', '\t', '\n', '\r', ',', ';', '(', '[', '{'}`.

In the following examples, `-1` is a single token:

```
echo -1
echo(-1)
echo [-1]
echo 3,-1

"abc";-1
```

In the following examples, `-1` is parsed as two separate tokens (as `-` and `1`):

```
echo x-1
echo (int)-1
echo [a]-1
"abc"-1
```

The suffix starting with an apostrophe (`'`) is called a *type suffix*. Literals without a type suffix are of an integer type unless the literal contains a dot or `E|e` in which case it is of type `float`. This integer type is `int` if the literal is in the range `low(int32)..high(int32)`, otherwise it is `int64`. For notational convenience, the apostrophe of a type suffix is optional if it is not ambiguous (only hexadecimal floating-point literals with a type suffix can be ambiguous).

The pre-defined type suffixes are:

Table 3. Pre-defined type suffixes

Type Suffix	Resulting type of literal
`'i8`	int8
`'i16`	int16
`'i32`	int32
`'i64`	int64
`'u`	uint
`'u8`	uint8
`'u16`	uint16
`'u32`	uint32
`'u64`	uint64
`'f`	float32
`'d`	float64
`'f32`	float32
`'f64`	float64

Literals must match the datatype, for example, `333'i8` is an invalid literal. Non-base-10 literals are used mainly for flags and bit pattern representations, therefore the checking is done on bit width and not on value range. Hence: `0b10000000'u8 == 0x80'u8 == 128`, but: `0b10000000'i8 == 0x80'i8 == -1`, instead of causing an overflow error.

12.12.1. Custom Numeric Literals

If the suffix is not predefined, then the suffix is assumed to be a call to a proc, template, macro or other callable identifier that is passed the string containing the literal. The callable identifier needs to be declared with a special `'` prefix:

```
import strutils
type u4 = distinct uint8 # a 4-bit unsigned integer aka "nibble"
proc `'u4`(n: string): u4 =
  # The leading ' is required.
  result = (parseInt(n) and 0x0F).u4

var x = 5'u4
```

More formally, a custom numeric literal `123'custom` is transformed to r"123".`'custom` in the parsing step. There is no AST node kind that corresponds to this transformation. The transformation naturally handles the case that additional parameters are passed to the callee:

```
import strutils
type u4 = distinct uint8 # a 4-bit unsigned integer aka "nibble"
proc `'u4`(n: string; moreData: int): u4 =
  result = (parseInt(n) and 0x0F).u4

var x = 5'u4(123)
```

Custom numeric literals are covered by the grammar rule named `CUSTOM_NUMERIC_LIT`. A custom numeric literal is a single token.

12.13. Operators

Nim allows user-defined operators, in fact, Nim does not distinguish between user-defined and builtin operators. When one writes `1 + 2` it is a call to a plus operator ``` ``(1, 2)`` which is subject to overload resolution. The `system` module is automatically imported in every Nim program and offers ``func \``(a, b: int): int ``` so the call will be resolved to ``system.`+`(1, 2)``.

An operator is any combination of the following characters:

```
=     +     -     *     /     <     >
@     $     ~     &     %     |
!     ?     ^     .     :     \
```

(The grammar uses the terminal OPR to refer to operator symbols as defined here.)

These keywords are also operators: `and or not xor shl shr div mod in notin is isnot of as from`.

`.`, `=`, `:`, `::` are not available as general operators; they are used for other notational purposes.

`*:` is as a special case treated as the two tokens `*` and `:` (to support `var v*: T`).

The `not` keyword is always a unary operator, `a not b` is parsed as `a(not b)`, not as `(a) not (b)`.

12.14. Other tokens

The following strings denote other tokens:

```
` ( ) { } [ ] , ; [. .] {. .} (. .) [:
```

The *slice* operator `..` takes precedence over other tokens that contain a dot: `{..}` are the three tokens: `{` and `..` and `}`, and not the two tokens: `{.` and `.}`.

12.15. Unicode Operators

These Unicode operators are also parsed as operators:

```
• ∘ × ★ ⊗ ⊘ ⊙ ⊛ ⊠ ⊡ ∩ ∧    # same priority as * (multiplication)
± ⊕ ⊖ ⊞ ⊟ ∪ ∨              # same priority as + (addition)
```

Unicode operators can be combined with non-Unicode operator symbols. The usual precedence extensions then apply, for example, `×=` is an assignment like operator just like `*=` is.

No Unicode normalization step is performed.

Chapter 13. Syntax

This chapter describes in detail the second phase of the translation process called *parsing*.

How the parser handles the indentation is described in Chapter 12, *Lexical analysis*.

Nim allows user-definable operators. Binary operators have 11 different levels of precedence.

13.1. Associativity

Binary operators whose first character is `^` are right-associative, all other binary operators are left-associative.

```
proc `^/`(x, y: float): float =
  # a right-associative division operator
  result = x / y
echo 12 ^/ 4 ^/ 8 # 24.0 (4 / 8 = 0.5, then 12 / 0.5 = 24.0)
echo 12  / 4  / 8 # 0.375 (12 / 4 = 3.0, then 3 / 8 = 0.375)
```

13.1.1. Precedence

Unary operators always bind stronger than any binary operator: `$a + b` is `($a) + b` and not `$(a + b)`.

If a unary operator's first character is `@`, it is a *sigil-like* operator which binds stronger than a `primarySuffix`: `@x.abc` is parsed as `(@x).abc` whereas `$x.abc` is parsed as `$(x.abc)`.

For binary operators that are not keywords, the precedence is determined by the following rules:

Operators ending in either `->`, `~>` or `=>` are called *arrow-like*, and have the lowest precedence of all operators.

If the operator ends with `=` and its first character is none of `<`, `>`, `!`, `=`, `~`, `?`, it is an *assignment operator* which has the second-lowest precedence.

Otherwise, precedence is determined by the first character.

Table 4. Precedence levels

Precedence level	**Operators**	**First character**	**Terminal symbol**
10 (highest)		`$ ^`	OP10
9	`* / div mod shl shr %`	`* % \ /`	OP9
8	`+ -`	`+ - ~ |`	OP8
7	`&`	`&`	OP7
6	`..`	`.`	OP6
5	`== <= < >= > != in notin is isnot not of as from`	`= < > !`	OP5
4	`and`		OP4
3	`or xor`		OP3
2		`@ : ?`	OP2
1	*assignment operator* (like `+=`, `*=`)		OP1
0 (lowest)	*arrow-like operator* (like `->`, `=>`)		OP0

Whether an operator is used as a prefix operator is also affected by preceding whitespace:

```
echo $foo
# is parsed as
echo($foo)
```

Spacing also determines whether `(a, b)` is parsed as an argument list of a call or whether it is parsed as a tuple constructor:

```
echo(1, 2) # pass 1 and 2 to echo
```

```
echo (1, 2) # pass the tuple (1, 2) to echo
```

13.2. Dot-like operators

Terminal symbol in the grammar: `DOTLIKEOP`.

Dot-like operators are operators starting with `.`, but not with `..`, for example `.?`. Dot-like operators have the same precedence as `.`, so that `a.?b.c` is parsed as `(a.?b).c` instead of `a.?(b.c)`.

The rules of operator precedence were designed to be as “intuitive” as possible and so that you can avoid many parenthesis in practice. However, you are not supposed to learn the rules by heart, if in doubt use parenthesis explicitly:

```
(a and b) or c  # is more readable than
a and b or c
```

Other syntax rules are described in the following sections along the semantics of described construct like an `if` statement. The complete and formal grammar can found in Appendix A, *Grammar*.

Chapter 14. Declarations and scope rules

An *identifier* is a symbol declared as a name for a variable, type, procedure, etc. The region of the program over which a declaration applies is called the *scope* of the declaration. Scopes can be nested. The meaning of an identifier is determined by the smallest enclosing scope in which the identifier is declared unless overloading resolution rules suggest otherwise.

Symbols in Nim can be one of the following kind:

- `proc`, `func`, `iterator`, `converter`, `template`, `macro`, `method`: These are called *routines*. Routines can be called and perform computations.
- `var`: A variable that can be re-assigned.
- `let`: A variable that cannot be re-assigned.
- `const`: A symbol bound to a constant value. The value must be computable at compile-time.
- `type`: A name for a type.
- `parameter`: A name for a routine's formal parameter.
- `result`: A variable that represents a routine's return value.
- `enum field`: A value that belongs to an enum type.
- `object field`: A field inside an object declaration.

Symbols of kind "routine" or "enum field" can be *overloaded*. Multiple entries of an overloaded symbol can be accessible in a single scope. *Overload resolution* determines how these entries are resolved, ambiguous symbols produce a compile-time error. Non-overloaded symbols must be uniquely declared within a scope. Some examples:

```nim
block: # introduces a new scope
  var x = 0 # valid

  let x = "abc" # invalid as an 'x' was already declared

echo x # invalid: access of 'x' outside of its scope

proc p(x: int) = echo "int"
proc p(x: string) = echo "string" # valid, `p` is overloaded

p "abc" # valid invocation; overload resolution picks `proc p(x: string)`
```

Overloaded symbols can be accessed even though the symbols might be shadowed by a non-overloadable symbol:

```nim
proc len[T](a: openArray[T]): int = 1 #1
proc len(a: string): int = 2 #2

proc main =
  let len = 3 # local variable 'len' shadows #1 and #2
  echo len("xyz") # yet overload resolution selects #2 regardless
```

Only certain syntactic contexts such as routine *calls* trigger overload resolution. In other contexts *overload disambiguation* is performed. The corresponding sections Chapter 19, *Overload resolution* and Section 19.4, “Overload disambiguation” contain the details.

Chapter 15. Modules

A Nim program can be split into different pieces called *modules*. Each module needs to be in its own file and has its own *scope*. Modules enable *information hiding* and *separate compilation*. A module may gain access to symbols of another module by the `import` statement. *Recursive module dependencies* are allowed, but are slightly subtle. Only top-level symbols that are marked with an asterisk (`*`) are exported. A valid module name can only be a valid Nim identifier (and thus its filename is `identifier.nim`).

15.1. Export marker

If a declared symbol is marked with an asterisk (`*`) it is exported from the current module:

```
proc exportedEcho*(s: string) = echo s
proc `*`*(a: string; b: int): string =
  result = newStringOfCap(a.len * b)
  for i in 1..b: result.add a

var exportedVar*: int
const exportedConst* = 78
type
  ExportedType* = object
    exportedField*: int
```

15.2. Module processing

The algorithm for compiling modules is:

- Compile the whole module as usual, following import statements recursively.

- If there is a cycle, only import the already parsed symbols (that are exported); unknown identifiers cause a compile-time error.

This is best illustrated by an example:

```
# Module A
type
  T1* = int  # Module A exports the type `T1`
import B     # the compiler starts parsing B

proc main() =
  var i = p(3) # works because B was parsed completely here

main()
```

```
# Module B
import A  # A is not parsed here! Only the already known symbols
          # of A are imported.

proc p*(x: A.T1): A.T1 =
  # this works because T1 has already been
  # added to A's interface symbol table
  result = x + 1
```

15.3. Import statement

After the `import` statement, a list of module names can follow or a single module name followed by an `except` list to prevent some symbols from being imported:

```
import std/strutils except `%`, toUpperAscii

# doesn't work then:
echo "$1" % "abc".toUpperAscii
```

It is not checked that the `except` list is really exported from the module. This feature allows us to compile against an older version of the module that does not export these identifiers.

An `import` statement must be a top level statement, but it can be used within a `when` statement:

```
when defined(posix):
  import posix
else:
  import windows
```

15.4. Include statement

The `include` statement inserts the contents of a Nim source code file at the position where the `include` statement is placed. Note that `include` is fundamentally different from `import`.

The `include` statement is useful to split up a large module into several files:

```
include fileA, fileB, fileC
```

The `include` statement can be used outside of the top level, as such:

```
# Module A
echo "Hello World!"
```

```
# Module B
proc main() =
  include A

main() # => Hello World!
```

15.5. Module names in imports

A module alias can be introduced via the `as` keyword:

```
import std/strutils as su, std/sequtils as qu

echo su.format("$1", "lalelu")
```

The original module name is then not accessible. The notations `path/to/module` or `"path/to/module"` can be used to refer to a module in subdirectories:

```
import lib/pure/os, "lib/pure/times"
```

Note that the module name is still `strutils` and not `lib/pure/strutils` and so one *cannot* do:

```
import lib/pure/strutils
echo lib/pure/strutils.toUpperAscii("abc")
```

Likewise, the following is invalid as the name is `strutils` already:

```
import lib/pure/strutils as strutils
```

15.6. Collective imports from a directory

The syntax `import dir / [moduleA, moduleB]` can be used to import multiple modules from the same directory.

Path names are syntactically either Nim identifiers or string literals. If the path name is not a valid Nim identifier it needs to be a string literal:

```
import "gfx/3d/somemodule" # in quotes because '3d' is not a valid Nim
identifier
```

15.7. Pseudo import/include paths

A directory can also be a so-called "pseudo directory". They can be used to avoid ambiguity when there are multiple modules with the same path.

There are two pseudo directories:

1. `std`: The `std` pseudo directory is the abstract location of Nim's standard library. For example, the syntax `import std / strutils` is used to unambiguously refer to the standard library's `strutils` module.
2. `pkg`: The `pkg` pseudo directory is used to unambiguously refer to a Nimble package. However, for technical details that lie outside the scope of this document, its semantics are: *Use the search path to look for module name but ignore the standard library locations*. In other words, it is the opposite of `std`.

15.8. From import statement

After the `from` statement, a module name follows followed by an `import` to list the symbols one likes to use without explicit full qualification:

```
from std/strutils import `%`

echo "$1" % "abc"
# always possible: full qualification:
echo strutils.replace("abc", "a", "z")
```

It's also possible to use `from module import nil` if one wants to import the module but wants to enforce fully qualified access to every symbol in `module`.

15.9. Export statement

An `export` statement can be used for symbol forwarding so that client modules don't need to import a module's dependencies:

```
# module B
type MyObject* = object


# module A
import B
export B.MyObject

proc `$`*(x: MyObject): string = "my object"


# module C
import A

# B.MyObject was imported implicitly here:
var x: MyObject
echo $x
```

When the exported symbol is another module, all of its definitions will be forwarded. One can use an `except` list to exclude some of the symbols.

Notice that when exporting, one needs to specify only the module name:

```
import foo/bar/baz
export baz
```

15.10. Scope rules

Identifiers are valid from the point of their declaration until the end of the block in which the declaration occurred. The range where the identifier is known is the scope of the identifier. The exact scope of an identifier depends on the way it was declared.

15.10.1. Block scope

The *scope* of a variable declared in the declaration part of a block is valid from the point of declaration until the end of the block. If a block contains a second block, in which the identifier is re-declared, then inside this block, the second declaration will be valid. Upon leaving the inner block, the first declaration is valid again. An identifier cannot be redefined in the same block, except if valid for overloading purposes.

15.10.2. Tuple or object scope

The field identifiers inside a tuple or object definition are valid in the following places:

- To the end of the tuple/object definition.
- Field designators of a variable of the given tuple/object type.
- In all descendant types of the object type.

15.10.3. Module scope

All identifiers of a module are valid from the point of declaration until the end of the module. Identifiers from indirectly dependent modules are *not* available. The system module is automatically imported in every module.

If a module imports an identifier by two different modules, each occurrence of the identifier has to be qualified unless it is an overloaded procedure or iterator in which case the overloading resolution takes place:

```nim
# Module A
var x*: string
```

```nim
# Module B
var x*: int
```

```nim
# Module C
import A, B
write(stdout, x) # error: x is ambiguous
write(stdout, A.x) # no error: qualifier used

var x = 4
write(stdout, x) # not ambiguous: uses the module C's x
```

Chapter 16. Type system

All expressions have a type that is known during *semantic analysis*. Nim is statically typed. One can declare new types, which is, in essence, defining an identifier that can be used to denote this custom type.

These are the major type classes:

- ordinal types: consist of integer, bool, character, enumeration (and subranges thereof) types
- floating-point types
- string type
- structured types
- reference (pointer) type
- procedural type
- generic type

16.1. Ordinal types

Ordinal types have the following characteristics:

- Ordinal types are countable and ordered. This property allows the operation of functions such as `inc`, `ord`, and `dec` on ordinal types to be defined.
- Ordinal types have a smallest possible value, accessible with `low(type)`. Trying to count further down than the smallest value produces a panic or a static error.
- Ordinal types have a largest possible value, accessible with `high(type)`. Trying to count further up than the largest value produces a panic or a

static error.

Integers, bool, characters, and enumeration types (and subranges of these types) belong to ordinal types.

A distinct type is an ordinal type if its base type is an ordinal type.

16.2. Pre-defined integer types

These integer types are pre-defined:

`int`

: The generic signed integer type; its size is platform-dependent and has the same size as a pointer. This type should be used in general. An integer literal that has no type suffix is of this type if it is in the range `low(int32)..high(int32)` otherwise the literal's type is `int64`.

`intN`

: Additional signed integer types of N bits use this naming scheme (example: int16 is a 16-bit wide integer). The current implementation supports `int8`, `int16`, `int32`, `int64`. Literals of these types have the suffix 'iN.

`uint`

: The generic unsigned integer type; its size is platform-dependent and has the same size as a pointer. An integer literal with the type suffix `'u` is of this type.

`uintN`

: Additional unsigned integer types of N bits use this naming scheme (example: `uint16`` is a 16-bit wide unsigned integer). The current implementation supports `uint8`, `uint16`, `uint32`, `uint64`. Literals of these types have the suffix 'uN. Unsigned operations all wrap around; they cannot lead to over- or underflow errors.

Automatic type conversions are performed in expressions where different kinds of integer types are used: the smaller type is converted to the larger.

A *narrowing type conversion* converts a larger to a smaller type (for example `int32 → int16`). A *widening type conversion* converts a smaller type to a larger type (for example `int16 → int32`). In Nim only widening type conversions are *implicit*:

```
var myInt16 = 5'i16
var myInt: int
echo myInt16 + 34'i8  # of type `int16`
echo myInt16 + myInt  # of type `int`
echo myInt16 + 2'i32  # of type `int32`
```

For further details, see Section 17.3, “Convertible relation”.

Table 5. Integer operations

Operation	Meaning
`+`	Integer addition
`-`	Integer subtraction
`*`	Integer multiplication
`inc`	Increment an integer / an ordinal type
`dec`	Decrement an integer / an ordinal type
`div`	Integer division
`mod`	Integer modulo (remainder)
`shl`	Shift left
`shr`	Shift right
`ashr`	Arithmetic shift right
`and`	Bitwise `and`
`or`	Bitwise `or`
`xor`	Bitwise `xor`
`not`	Bitwise `not` (complement)

16.3. Integer literals

`int` literals are implicitly convertible to a smaller integer type if the literal’s value fits this smaller type and such a conversion is less expensive than other implicit conversions, so `myInt16 + 34` produces an `int16` result.

16.4. Subrange types

A subrange type is a range of values from an ordinal or floating-point type (the base type). To define a subrange type, one must specify its limiting values — the lowest and highest value of the type. For example:

```
type
  Subrange = range[0..5]
  PositiveFloat = range[0.0..Inf]
  Positive* = range[1..high(int)] # as defined in `system`
```

`Subrange` is a subrange of an integer which can only hold the values 0 to 5. `PositiveFloat` defines a subrange of all positive floating-point values. NaN does not belong to any subrange of floating-point types. Assigning any other value to a variable of type `Subrange` is a panic (or a static error if it can be determined during semantic analysis). Assignments from the base type to one of its subrange types (and vice versa) are allowed.

A subrange type has the same size as its base type (`int` in the `Subrange` example).

16.5. Pre-defined floating-point types

The following floating-point types are pre-defined:

`float`
: The generic floating-point type; its size used to be platform-dependent, but now it is always mapped to `float64`. This type should be used in general.

`floatN`
: Nim defines floating-point types of N bits using this naming scheme (example: `float64` is a 64-bit wide float). The current implementation supports `float32` and `float64`. Literals of these types have the suffix `'fN`.

Automatic type conversion in expressions with different kinds of floating-point types is performed: See Section 17.3, "Convertible relation" for further details. Arithmetic performed on floating-point types follows the IEEE standard. Integer types are not converted to floating-point types automatically and vice versa.

The IEEE standard defines five types of floating-point exceptions:

- Invalid: operations with mathematically invalid operands, for example: `0.0/0.0`, `sqrt(-1.0)`, and `log(-37.8)`.
- Division by zero: divisor is zero and dividend is a finite nonzero number, for example `1.0/0.0`.
- Overflow: operation produces a result that exceeds the range of the exponent, for example `MAXDOUBLE+0.0000000000001e308`.
- Underflow: operation produces a result that is too small to be represented as a normal number, for example: `MINDOUBLE * MINDOUBLE`.
- Inexact: operation produces a result that cannot be represented with infinite precision, for example: `2.0 / 3.0`, `log(1.1)` and `0.1` in input.

The IEEE exceptions are either ignored during execution or mapped to the Nim exceptions: `FloatInvalidOpDefect`, `FloatDivByZeroDefect`, `FloatOverflowDefect`, `FloatUnderflowDefect`, and `FloatInexactDefect`. These exceptions inherit from the `FloatingPointDefect` base class.

Table 6. Float operations

Operation	Meaning
`+`	Float addition
`-`	Float subtraction
`*`	Float multiplication
`/`	Float division

16.5.1. Nan and Inf checks

The Nim compiler provides the pragmas `nanChecks` and `infChecks` to control whether the IEEE exceptions are ignored or trap a Nim exception:

```
{.nanChecks: on, infChecks: on.}
var a = 1.0
var b = 0.0
echo b / b # raises FloatInvalidOpDefect
echo a / b # raises FloatOverflowDefect
```

In the current implementation `FloatDivByZeroDefect` and `FloatInexactDefect` are never raised. FloatOverflowDefect is raised instead of FloatDivByZeroDefect. There is also a floatChecks pragma that is a short-cut for the combination of nanChecks and infChecks pragmas. floatChecks are turned off as default.

The only operations that are affected by the floatChecks pragma are the +, -, *, / operators for floating-point types.

The Nim compiler uses the maximum precision available to evaluate floating-point values during semantic analysis; this means expressions like 0.09'f32 + 0.01'f32 == 0.09'f64 + 0.01'f64 that are evaluating during constant folding are true.

16.6. Boolean type

The boolean type is named bool in Nim and can be one of the two pre-defined values true and false. Conditions in while, if, elif, when statements need to be of type bool.

This condition holds:

```
ord(false) == 0 and ord(true) == 1
```

The operators not, and, or, xor, <, <=, >, >=, !=, == are defined for the bool type. The and and or operators perform short-cut evaluation. Example:

```
while p != nil and p.name != "xyz":
  # p.name is not evaluated if p == nil
  p = p.next
```

The size of the bool type is one byte.

It can be a good idea to use a custom `enum` type instead of `bool` even if the enum has only two possible values. Compare:

```nim
proc deleteFile(f: string): bool
```

To:

```nim
type
  Status = enum
    Failure,
    Success

proc deleteFile(f: string): Status
```

16.7. Character type

The character type is named `char` in Nim. Its size is one byte. Thus it cannot represent a UTF-8 character, but a part of it.

The standard library offers a `Rune` type, that can represent any Unicode character, in its `unicode` module.

16.8. Enumeration types

Enumeration types define a new type whose values consist of the ones specified. The values are ordered. Example:

```nim
type
  Direction = enum
    north, east, south, west

assert ord(north) == 0
assert ord(east) == 1
assert ord(south) == 2
assert ord(west) == 3

# Also allowed:
assert ord(Direction.west) == 3
```

The implied order is: north < east < south < west. The comparison operators can be used with enumeration types. Instead of `north` etc, the enum value can also be qualified with the enum type that it resides in, `Direction.north`.

For better interfacing to other programming languages, the fields of enum types can be assigned an explicit ordinal value. However, the ordinal values have to be in ascending order. A field whose ordinal value is not explicitly given is assigned the value of the previous field + 1.

> Idiomatic Nim code makes heavy use of enum types. If you use an enum in a `case` statement, the compiler enforces that every possible enum value is handled explicitly (unless an `else` section is present). The value of thinking about every possible case can hardly be overstated, it makes for more robust software that is easy to maintain: If a new state is added, all the places in a codebase that need to be considered are listed by the compiler's error messages. This is far preferable to a more object oriented approach where the dispatching is distributed over multiple files and there is no enforcement if classes do override a virtual method.

An explicit ordered enum can have *holes*:

```
type
  TokenType = enum
    a = 2, b = 4, c = 89 # holes are valid
```

However, it is then not ordinal anymore, so it is impossible to use these enums as an index type for arrays. The procedures `inc`, `dec`, `succ` and `pred` are not available for them either.

An enum value can be turned into its string representation via the built-in stringify operator `$`. The stringify's result can be controlled by explicitly giving the string values to use:

```
type
  MyEnum = enum
    valueA = (0, "my value A"),
    valueB = "value B",
    valueC = 2,
    valueD = (3, "abc")
```

As can be seen from the example, it is possible to both specify a field's ordinal value and its string value by using a tuple. It is also possible to only specify one of them.

An enum can be marked with the `pure` pragma so that its fields are added to a special module-specific hidden scope that is only queried as the last attempt. Only non-ambiguous symbols are added to this scope. But one can always access these via type qualification written as `MyEnum.value`:

```
type
  MyEnum {.pure.} = enum
    valueA, valueB, valueC, valueD, amb

  OtherEnum {.pure.} = enum
    valueX, valueY, valueZ, amb

echo valueA # MyEnum.valueA
echo amb    # Error: Unclear whether it's MyEnum.amb or OtherEnum.amb
echo MyEnum.amb # OK.
```

16.9. Overloadable enum field names

Enum field names are overloadable much like routines. When an overloaded enum field is used, it produces a closed sym choice construct, here written as `(E|E)`. During overload resolution the right `E` is picked, if possible. For (array/object...) constructors the right `E` is picked, comparable to how `[byte(1), 2, 3]` works, one needs to use `[T.E, E2, E3]`. Ambiguous enum fields produce a static error:

```
type
  E1 = enum
    value1, value2
  E2 = enum
    value1, value2 = 4

const
  lookupTable = [
    E1.value1: "1",
    value2: "2"]

proc p(e: E1) =
  # disambiguation in 'case' statements:
  case e
  of value1: echo "A"
  of value2: echo "B"
```

16.10. String type

All string literals are of the type `string`. A string in Nim is very similar to a sequence of characters. However, strings in Nim are both zero-terminated and have a length field. One can retrieve the length with the builtin `len` procedure; the length never counts the terminating zero.

The terminating zero cannot be accessed unless the string is converted to the `cstring` type first. The terminating zero assures that this conversion can be done in O(1) and without any allocations.

The assignment operator for strings always copies the string. The `&` operator concatenates strings.

Most native Nim types support conversion to strings with the special `$` proc. When calling the `echo` proc, for example, the built-in stringify operation for the parameter is called:

```
echo 3 # calls `$` for `int`
```

Whenever a user creates a specialized object, implementation of this procedure provides for `string` representation.

```
type
  Person = object
    name: string
    age: int

proc `$`(p: Person): string = # `$` always returns a string
  result = p.name & " is " &
           $p.age & # we _need_ the `$` in front of p.age which
                    # is natively an integer to convert it to
                    # a string
          " years old."
```

While `$p.name` can also be used, the `$` operation on a string does nothing. Note that we cannot rely on automatic conversion from an `int` to a `string` like we can for the `echo` proc.

Strings are compared by their lexicographical order. All comparison operators are available. Strings can be indexed like arrays (lower bound is 0). Unlike arrays, they can be used in case statements:

```
case paramStr(i)
of "-v": incl(options, optVerbose)
of "-h", "-?": incl(options, optHelp)
else: write(stdout, "invalid command line option!\n")
```

Per convention, all strings are UTF-8 strings, but this is not enforced. For example, when reading strings from binary files, they are merely a sequence of bytes. The index operation `s[i]` means the i-th *char* of `s`, *not* the i-th *code point*.

> In the modern programming world, strings are overused heavily. Section 16.27, "Distinct type" contains further advice.

A reference with the most used operations on strings is available in the appendix, under Section B.2, "Strings".

16.11. cstring type

The `cstring` type meaning `compatible string` is the native representation of a string for the compilation backend. For the C backend the `cstring` type represents a pointer to a zero-terminated char array compatible with the type `char*` in ANSI C. Its primary purpose lies in easy interfacing with C. The index operation `s[i]` means the i-th *char* of `s`; however no bounds checking for `cstring` is performed making the index operation unsafe.

A Nim `string` is implicitly convertible to `cstring` for convenience. If a Nim string is passed to a C-style variadic proc, it is implicitly converted to `cstring` too:

```
proc printf(formatstr: cstring) {.importc: "printf", varargs,
                                  header: "<stdio.h>".}

printf("This works %s", "as expected")
```

Even though the conversion is implicit, it is not *safe*: The garbage collector does not consider a `cstring` to be a root and may collect the underlying memory.

A `$` proc is defined for cstrings that returns a string. Thus to get a Nim string from a cstring:

```
var str: string = "Hello!"
var cstr: cstring = str
var newstr: string = $cstr
```

16.12. Structured types

A variable of a structured type can hold multiple values at the same time. Structured types can be nested to unlimited levels. Arrays, sequences, tuples, objects, and sets belong to the structured types.

16.13. Array and sequence types

Arrays are a homogeneous type, meaning that each element in the array has the same type. Arrays always have a fixed length specified as a constant expression (except for open arrays). They can be indexed by any ordinal type. A parameter `A` may be an *open array*, in which case it is indexed by integers from 0 to `len(A)-1`. An array expression may be constructed by the array constructor `[]`. The element type of this array expression is inferred from the type of the first element. All other elements need to be implicitly convertible to this type.

An array type can be defined using the `array[size, T]` syntax, or using `array[lo..hi, T]` for arrays that start at an index other than zero.

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Sequences are implemented as growable arrays, allocating pieces of memory as items are added. A sequence `S` is always indexed by integers from 0 to `len(S)-1` and its bounds are checked. Sequences can be constructed by the array constructor `[]` in conjunction with the array to sequence operator `@`. Another way to allocate space for a sequence is to call the built-in `newSeq` procedure.

A sequence may be passed to a parameter that is of type *open array*.

Example:

```
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
  IntSeq = seq[int] # a sequence of integers
var
  x: IntArray
```

```
  y: IntSeq
x = [1, 2, 3, 4, 5, 6]  # [] is the array constructor
y = @[1, 2, 3, 4, 5, 6] # the @ turns the array into a sequence

let z = [1.0, 2, 3, 4] # the type of z is array[0..3, float]
```

The lower bound of an array or sequence may be received by the built-in proc `low()`, the higher bound by `high()`. The length may be received by `len()`. `low()` for a sequence or an open array always returns 0, as this is the first valid index. One can append elements to a sequence with the `add()` proc or the `&` operator, and remove (and get) the last element of a sequence with the `pop()` proc.

The notation `x[i]` can be used to access the i-th element of `x`.

Arrays accesses are bounds checked (statically or at runtime). These checks can be disabled via a pragma `.push boundChecks:off`.

An array constructor can have explicit indexes for readability:

```
type
  Values = enum
    valA, valB, valC

const
  lookupTable = [
    valA: "A",
    valB: "B",
    valC: "C"
  ]
```

If an index is left out, `succ(lastIndex)` is used as the index value:

```
type
  Values = enum
    valA, valB, valC, valD, valE

const
  lookupTable = [
    valA: "A",
    "B",
    valC: "C",
    "D", "e"
  ]
```

A reference with the most used operations on sequences is available in the appendix, under Section B.3, "Sequences".

16.14. Open arrays

Often fixed size arrays turn out to be too inflexible; routines should be able to deal with arrays of different sizes. The `openarray` type allows this; it can only be used for parameters. Openarrays are always indexed with an `int` starting at position 0. The `len`, `low` and `high` operations are available for open arrays too. Any array with a compatible base type can be passed to an openarray parameter, the index type does not matter. In addition to arrays, sequences can also be passed to an open array parameter.

The openarray type cannot be nested: multidimensional openarrays are not supported because this is seldom needed and cannot be done efficiently.

```
proc testOpenArray(x: openArray[int]) = echo repr(x)

testOpenArray([1,2,3])  # array[]
testOpenArray(@[1,2,3]) # seq[]
```

16.15. Varargs

A `varargs` parameter is an openarray parameter that additionally allows to pass a variable number of arguments to a procedure. The compiler converts the list of arguments to an array implicitly:

```
proc myWriteLn(f: File, a: varargs[string]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteLn(stdout, "abc", "def", "xyz")
# is transformed to:
myWriteLn(stdout, ["abc", "def", "xyz"])
```

This transformation is only done if the varargs parameter is the last parameter in the procedure header. It is also possible to perform type conversions in this context:

```
proc myWriteLn(f: File, a: varargs[string, `$`]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteLn(stdout, 123, "abc", 4.0)
# is transformed to:
myWriteLn(stdout, [$123, $"def", $4.0])
```

In this example `$` is applied to any argument that is passed to the parameter `a`. (Note that `$` applied to strings is a nop.)

Note that an explicit array constructor passed to a `varargs` parameter is not wrapped in another implicit array construction:

```
proc takeV[T](a: varargs[T]) = discard

takeV([123, 2, 1]) # takeV's T is "int", not "array of int"
```

`varargs[typed]` is treated specially: It matches a variable list of arguments of arbitrary type but *always* constructs an implicit array. This is required so that the builtin `echo` proc does what is expected:

```
proc echo*(x: varargs[typed, `$`]) {...}

echo @[1, 2, 3]
# prints "@[1, 2, 3]" and not "123"
```

16.16. Unchecked arrays

The `UncheckedArray[T]` type is a special kind of `array` where its bounds are not checked. This is often useful to implement customized flexibly sized arrays. Additionally, an unchecked array is translated into a C array of undetermined size:

```
type
  MySeq = object
    len, cap: int
    data: UncheckedArray[int]
```

Produces roughly this C code:

```
typedef struct {
  NI len;
  NI cap;
  NI data[];
} MySeq;
```

The base type of the unchecked array may not contain any GC'ed memory but this is currently not checked.

16.17. Tuples and object types

A variable of a tuple or object type is a heterogeneous storage container. A tuple or object defines various named *fields* of a type. A tuple also defines a lexicographic *order* of the fields. Tuples are meant to be heterogeneous storage types with few abstractions. The `()` syntax can be used to construct tuples. The order of the fields in the constructor must match the order of the tuple's definition. Different tuple-types are *equivalent* if they specify the same fields of the same type in the same order. The *names* of the fields also have to be the same.

The assignment operator for tuples copies each component. The default assignment operator for objects copies each component. Overloading of the assignment operator is described in Chapter 29, *Lifetime-tracking hooks*.

```
type
  Person = tuple[name: string, age: int] # type representing a person:
                                         # it consists of a name and an
age.
var person: Person
person = (name: "Peter", age: 30)
assert person.name == "Peter"
# the same, but less readable:
person = ("Peter", 30)
assert person[0] == "Peter"
assert Person is (string, int)
assert (string, int) is Person
assert Person isnot tuple[other: string, age: int] # `other` is a different
identifier
```

A tuple with one unnamed field can be constructed with the parentheses and a trailing comma:

```nim
proc echoUnaryTuple(a: (int,)) =
  echo a[0]

echoUnaryTuple (1,)
```

In fact, a trailing comma is allowed for every tuple construction.

The implementation aligns the fields for the best access performance. The alignment is compatible with the way a C compiler does it.

For consistency with `object` declarations, tuples in a `type` section can also be defined with indentation instead of `[]`:

```nim
type
  Person = tuple   # type representing a person
    name: string   # a person consists of a name
    age: Natural   # and an age
```

Objects provide many features that tuples do not. Objects provide inheritance and the ability to hide fields from other modules. Objects with inheritance enabled have information about their type at runtime so that the `of` operator can be used to determine the object's type. The `of` operator is similar to the `instanceof` operator in Java.

```nim
type
  Person = object of RootObj
    name*: string   # the * means that `name` is accessible
                    # from other modules
    age: int        # no * means that the field is hidden

  Student = ref object of Person # a student is a person
    id: int                      # with an id field

var
  student: Student
  person: Person
assert(student of Student) # is true
assert(student of Person) # also true
```

Object fields that should be visible from outside the defining module have to be marked by `*`. In contrast to tuples, different object types are never *equivalent*, they are nominal types whereas tuples are structural. Objects that have no ancestor are implicitly `final` and thus have no hidden type information. One can use the `inheritable` pragma to introduce new object roots apart from `system.RootObj`.

```
type
  Person = object # example of a final object
    name*: string
    age: int

  Student = ref object of Person # Error: inheritance only
                                 # works with non-final objects
    id: int
```

16.18. `fields` and `fieldPairs` iterators

Nim's `system` module provides iterators that can be used to iterate over every field of an object or a tuple. `fieldPairs` yields `(key, val)` pairs, `fields` only yields the fields' values:

```
proc `$`[T: object](x: T): string = ①
  result = ""
  for name, val in fieldPairs(x): ②
    result.add name
    result.add ": "
    result.add $val ③
    result.add "\n"
```

① Possible implementation for how to generically generate the string representation of an object.

② Iterate over all fields of `x`.

③ Assume that the type of every field provides a `$` operation.

These iterators do allow for field mutations:

```
proc fromJ[T: object](t: typedesc[T]; j: JsonNode): T = ①
  result = T()
  for name, loc in fieldPairs(result):
    loc = fromJ(typeof(loc), j[name]) ②
```

① `fromJ` loads an object from a JSON tree named `j`.

② Store to `result.<field>`.

As outlined in the example, `fieldPairs` and `fields` can be used as a foundation for a serialization library.

Both fieldPairs and fields can be used to iterate over two objects in tandem:

```
proc `==`[T: object](x, y: T): bool = ①
  for a, b in fields(x, y):
    if not (a == b): return false ②
  return true
```

① A possible implementation of an equality operator for two objects of the same type.

② Assuming that the type of every field provides a == operation.

16.19. Object construction

Objects can also be created with an *object construction expression* that has the syntax T(fieldA: valueA, fieldB: valueB, ...) where T is an object type or a ref object type:

```
type
  Student = object
    name: string
    age: int
  PStudent = ref Student
var a1 = Student(name: "Anton", age: 5)
var a2 = PStudent(name: "Anton", age: 5)
# this also works directly:
var a3 = (ref Student)(name: "Anton", age: 5)
# not all fields need to be mentioned,
# and they can be mentioned out of order:
var a4 = Student(age: 5)
```

Note that, unlike tuples, objects require the field names along with their values. For a ref object type system.new is invoked implicitly.

16.20. Object variants

Object variants are tagged unions discriminated via an enumerated type used for runtime type flexibility, mirroring the concepts of *sum types* and *algebraic data types (ADTs)* as found in other programming languages.

An example:

```nim
# This is an example of how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref NodeObj
  NodeObj = object
    case kind: NodeKind  # the `kind` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

# create a new case object:
var n = Node(kind: nkIf, condition: nil)
# accessing n.thenPart is valid because the `nkIf` branch is active:
n.thenPart = Node(kind: nkFloat, floatVal: 2.0)

# the following statement raises an `FieldDefect` exception, because
# n.kind's value does not fit and the `nkString` branch is not active:
n.strVal = ""

# invalid: would change the active object branch:
n.kind = nkInt

var x = Node(kind: nkAdd, leftOp: Node(kind: nkInt, intVal: 4),
                          rightOp: Node(kind: nkInt, intVal: 2))
# valid: does not change the active object branch:
x.kind = nkSub
```

As can be seen from the example, an advantage to an object hierarchy is that no casting between different object types is needed. Yet, access to invalid object fields raises an exception.

The syntax of `case` in an object declaration follows closely the syntax of the `case` statement: The branches in a `case` section may be indented too.

In the example, the `kind` field is called the *discriminator*: For safety, its address cannot be taken and assignments to it are restricted: The new value must not lead to a change of the active object branch. Also, when the fields of a particular branch are specified during object construction, the corresponding discriminator value must be specified as a constant expression.

Instead of changing the active object branch, replace the old object in memory with a new one completely:

```
var x = Node(kind: nkAdd, leftOp: Node(kind: nkInt, intVal: 4),
                          rightOp: Node(kind: nkInt, intVal: 2))
# change the node's contents:
x[] = NodeObj(kind: nkString, strVal: "abc")
```

Starting with version 0.20 `system.reset` cannot be used anymore to support object branch changes as this never was completely memory safe.

As a special rule, the discriminator kind can also be bounded using a `case` statement. If possible values of the discriminator variable in a `case` statement branch are a subset of discriminator values for the selected object branch, the initialization is considered valid. This analysis only works for immutable discriminators of an ordinal type and disregards `elif` branches. For discriminator values with a `range` type, the Nim compiler checks if the entire range of possible values for the discriminator value is valid for the chosen object branch.

A small example:

```
let unknownKind = nkSub
# invalid: unsafe initialization because
# the kind field is not statically known:
var y = Node(kind: unknownKind, strVal: "y")
var z = Node()
case unknownKind
of nkAdd, nkSub:
  # valid: possible values of this branch are a subset of the
  # nkAdd/nkSub object branch:
  z = Node(kind: unknownKind, leftOp: Node(), rightOp: Node())
else:
  echo "ignoring: ", unknownKind

# also valid, since unknownKindBounded can only contain
# the values nkAdd or nkSub
let unknownKindBounded = range[nkAdd..nkSub](unknownKind)
z = Node(kind: unknownKindBounded, leftOp: Node(), rightOp: Node())
```

16.21. cast uncheckedAssign

Via a `{.cast(uncheckedAssign).}` section some restrictions for case objects can be disabled:

```
type
  TokenKind* = enum
    strLit, intLit
  Token = object
    case kind*: TokenKind
    of strLit:
      s*: string
    of intLit:
      i*: int64

proc passToVar(x: var TokenKind) = discard

var t = Token(kind: strLit, s: "abc")

{.cast(uncheckedAssign).}:
  # inside the 'cast' section it is allowed to pass 't.kind'
  # to a 'var T' parameter:
  passToVar(t.kind)

  # inside the 'cast' section it is allowed to set field 's' even though the
  # constructed 'kind' field has an unknown value:
  t = Token(kind: t.kind, s: "abc")

  # inside the 'cast' section it is allowed to assign to the
  # 't.kind' field directly:
  t.kind = intLit
```

16.22. Set type

The set type models the mathematical notion of a set. The set's base type can only be an ordinal type of a certain size, namely:

- `int8`-`int16`
- `uint8`/`byte`-`uint16`
- `char`
- `enum`

or equivalent. For signed integers the set's base type is defined to be in the range `0 .. MaxSetElements-1` where `MaxSetElements` is currently always 2^16.

The reason is that sets are implemented as high performance bit vectors. Attempting to declare a set with a larger type will result in an error:

```
var s: set[int64] # Error: set is too large
```

Nim also offers hash sets (which you need to import with `import sets`), which have no such restrictions.

Sets can be constructed via the set constructor: `{}` is the empty set. The empty set is type compatible with any concrete set type. The constructor can also be used to include elements (and ranges of elements):

```
type
  CharSet = set[char]
var
  x: CharSet
x = {'a'..'z', '0'..'9'} # This constructs a set that contains the
                         # letters from 'a' to 'z' and the digits
                         # from '0' to '9'
```

These operations are supported by sets:

Table 7. Set operations

Operation	Meaning
`A + B`	union of two sets
`A * B`	intersection of two sets
`A - B`	difference of two sets (A without B's elements)
`A == B`	set equality
`A <= B`	subset relation (A is subset of B or equal to B)
`A < B`	strict subset relation (A is a proper subset of B)
`e in A`	set membership (A contains element e)
`e notin A`	A does not contain element e
`contains(A, e)`	A contains element e
`card(A)`	the cardinality of A (number of elements in A)

Operation	Meaning
`incl(A, elem)`	same as `A = A + {elem}`
`excl(A, elem)`	same as `A = A - {elem}`

16.22.1. Bit fields

Sets are often used to define a type for the *flags* of a procedure. This is a cleaner (and type safe) solution than defining integer constants that have to be `or`'ed together.

Enum, sets and casting can be used together as in:

```
type
  MyFlag* {.size: sizeof(cint).} = enum
    A
    B
    C
    D
  MyFlags = set[MyFlag]

proc toNum(f: MyFlags): int = cast[cint](f)
proc toFlags(v: int): MyFlags = cast[MyFlags](v)

assert toNum({}) == 0
assert toNum({A}) == 1
assert toNum({D}) == 8
assert toNum({A, C}) == 5
assert toFlags(0) == {}
assert toFlags(7) == {A, B, C}
```

Note how the set turns enum values into powers of 2.

If using enums and sets with C, use distinct cint.

For interoperability with C there is also the `bitsize` pragma.

16.23. Reference and pointer types

References (similar to pointers in other programming languages) are a way to introduce many-to-one relationships. This means different references can point to and modify the same location in memory (also called *aliasing*).

Nim distinguishes between *traced* and *untraced* references. Untraced references are also called *pointers*. Traced references point to objects of a garbage-collected heap, untraced references point to manually allocated objects or objects somewhere else in memory. Thus untraced references are *unsafe*. However, for certain low-level operations (accessing the hardware) untraced references are unavoidable.

Traced references are declared with the `ref` keyword, untraced references are declared with the `ptr` keyword. In general, a `ptr T` is implicitly convertible to the `pointer` type.

An empty subscript `[]` notation can be used to de-refer a reference, the `addr` procedure returns the address of an item. An address is always an untraced reference. Thus the usage of `addr` is an *unsafe* feature.

The `.` (access a tuple/object field operator) and `[]` (array/string/sequence index operator) operators perform implicit dereferencing operations for reference types:

```
type
  Node = ref NodeObj
  NodeObj = object
    le, ri: Node
    data: int

var
  n: Node
new(n)
n.data = 9
# no need to write n[].data; in fact n[].data is highly discouraged!
```

In order to simplify structural type checking, recursive tuples are not valid:

```
# invalid recursion
type MyTuple = tuple[a: ref MyTuple]
```

Likewise `T = ref T` is an invalid type.

As a syntactical extension, `object` types can be anonymous if declared in a type section via the `ref object` or `ptr object` notations. This feature is useful if an object should only gain reference semantics:

```
type
  Node = ref object
    le, ri: Node
    data: int
```

To allocate a new traced object, the built-in procedure `system.new` can be used.

To deal with untraced memory, non-built-in procs like `system.alloc`, `system.dealloc` and `system.realloc` can be used. But these procs are beyond the scope of this document.

16.24. Nil

If a reference points to *nothing*, it has the value `nil`. `nil` is the default value for all `ref` and `ptr` types.

Dereferencing `nil` is an unrecoverable fatal runtime error (and not a panic). Apart from that, `nil` is a value like any other - it can be used in assignments and comparisons.

A successful dereferencing operation `p[]` implies that `p` is not `nil`. This can be exploited by the implementation to optimize code like:

```
p[].field = 3
if p != nil:
  # if p were nil, `p[]` would have caused a crash already,
  # so we know `p` is always not nil here.
  action()
```

Into:

```
p[].field = 3
action()
```

This is not comparable to C's “undefined behavior” for dereferencing NULL pointers.

16.25. Procedural type

A procedural type is internally a pointer to a procedure. `nil` is an allowed value for a variable of a procedural type.

Examples:

```
proc printItem(x: int) = ...

proc forEach(c: proc (x: int) {.cdecl.}) =
  ...

forEach(printItem)  # this will NOT compile because
                    # calling conventions differ
```

```
type
  OnMouseMove = proc (x, y: int) {.closure.}

proc onMouseMove(mouseX, mouseY: int) =
  # has default calling convention
  echo "x: ", mouseX, " y: ", mouseY

proc setOnMouseMove(mouseMoveEvent: OnMouseMove) = discard

# ok, 'onMouseMove' has the default calling convention, which is compatible
# to 'closure':
setOnMouseMove(onMouseMove)
```

16.26. Calling conventions

A subtle issue with procedural types is that the calling convention of the procedure influences the type compatibility: procedural types are only compatible if they have the same calling convention. As a special extension, a procedure of the calling convention `nimcall` can be passed to a parameter that expects a proc of the calling convention `closure`.

The reference implementation supports these *calling conventions*:

nimcall

: is the default convention used for a Nim `proc`. It is the same as `fastcall`, but only for C compilers that support `fastcall`.

closure

is the default calling convention for a *procedural type* that lacks any pragma annotations. It indicates that the procedure has a hidden implicit parameter (an *environment*). Proc vars that have the calling convention `closure` take up two machine words: One for the proc pointer and another one for the pointer to implicitly passed environment.

stdcall

This is the stdcall convention as specified by Microsoft. The generated C procedure is declared with the `__stdcall` keyword.

cdecl

The cdecl convention means that a procedure shall use the same convention as the C compiler. Under Windows the generated C procedure is declared with the `__cdecl` keyword.

safecall

This is the safecall convention as specified by Microsoft. The generated C procedure is declared with the `_safecall keyword. The word _safe` refers to the fact that all hardware registers shall be pushed to the hardware stack.

inline

The inline convention means the caller should not call the procedure, but inline its code directly. Note that Nim does not inline, but leaves this to the C compiler; it generates `__inline` procedures. This is only a hint for a Nim implementation: it may completely ignore it and it may inline procedures that are not marked as `inline`.

fastcall

Fastcall means different things to different C compilers. One gets whatever the C `__fastcall` means.

thiscall

This is the thiscall calling convention as specified by Microsoft, used on C++ class member functions on the x86 architecture.

syscall

The syscall convention is the same as `__syscall`:c: in C. It is used for interrupts.

noconv

The generated C code will not have any explicit calling convention and thus use the C compiler's default calling convention. This is needed because Nim's default calling convention for procedures is `fastcall` to improve speed.

Most calling conventions exist only for the Windows 32-bit platform.

The default calling convention is `nimcall`, unless it is an inner proc (a proc inside of a proc). For an inner proc an analysis is performed whether it accesses its environment. If it does so, it has the calling convention `closure`, otherwise it has the calling convention `nimcall`.

16.27. Distinct type

A `distinct` type is a new type derived from a *base type* that is incompatible with its base type. In particular, it is an essential property of a distinct type that it *does not* imply a subtype relation between it and its base type. Explicit type conversions from a distinct type to its base type and vice versa are allowed.

In the modern programming world strings are overused heavily: The mere fact that JSON, XML, SQL, regular expressions, file paths, etc. have a *string representation* does not imply that you should use `string` for these things! Type safety is compromised when everything is a `string`. Instead you should use different types for different things. As a first step this usually means to use a `distinct` type. The following snippet was extracted from Nim's standard library (`db_common.nim`):

```
type
  SqlQuery* = distinct string

template sql*(query: string): SqlQuery =
  SqlQuery(query)

iterator rows*(db: DbConn, query: SqlQuery,
               args: varargs[string, `$`]): Row

# usage:
for row in rows(sql"SELECT id FROM user WHERE name = ?", "abc"): ...

# impossible, prevented at compile-time:
for row in rows(sql"SELECT id FROM user WHERE name = ?" & "abc"): ... ①
```

① Since the SqlQuery is a distinct string there is no & operator for it available.

16.27.1. borrow annotation

A borrow annotation can be used in order to borrow an operation from a type T to its distinct T equivalent:

```
type
  Id = distinct int

proc `==`(a, b: Id): bool {.borrow.}
  # Ids can be compared, but have no order so `<=` and `<` are not borrowed.
```

16.28. Auto type

The auto type can only be used for return types and parameters. For return types it causes the inference of the type from the routine body:

```
proc returnsInt(): auto = 1984
```

For parameters it currently creates implicitly generic routines:

```
proc foo(a, b: auto) = discard
```

Is the same as:

```
proc foo[T1, T2](a: T1, b: T2) = discard
```

However, later versions of the language might change this to mean "infer the parameters' types from the body". Then the above foo would be rejected as the parameters' types can not be inferred from an empty discard statement.

> Usage of auto is discouraged as it has few benefits over spelling out the types explicitly and the severe downside that it makes the code harder to read. Currently Nim’s documentation generator does not translate an auto return type to its inferred type.

16.29. static[T]

`static` is a type modifier. A static parameter must be a constant expression:

```
proc precompiledRegex(pattern: static string): RegEx =
  var res {.global.} = re(pattern)
  return res

precompiledRegex("/d+") # Replaces the call with a precompiled
                        # regex, stored in a global variable

precompiledRegex(paramStr(1)) # Error, command-line options
                              # are not constant expressions
```

For the purposes of code generation, all static params are treated as generic params - the proc will be compiled separately for each unique supplied value (or combination of values).

Static params can also appear in the signatures of generic types:

```
type
  Matrix[M,N: static int; T: Number] = array[0..(M*N - 1), T]
    # Note how `Number` is just a type constraint here, while
    # `static int` requires us to supply an int value

  AffineTransform2D[T] = Matrix[3, 3, T]
  AffineTransform3D[T] = Matrix[4, 4, T]

var m1: AffineTransform3D[float]  # OK
var m2: AffineTransform2D[string] # Error, `string` is not a `Number`
```

Please note that `static T` is just a syntactic convenience for the underlying generic type `static[T]`. The type param can be omitted to obtain the type class of all constant expressions. A more specific type class can be created by instantiating `static` with another type class.

One can force an expression to be evaluated at compile time as a constant expression by coercing it to a corresponding `static` type:

```
import std/math

echo static(fac(5)), " ", static[bool](16.isPowerOfTwo)
```

The Nim compiler should report any failure to evaluate the expression or a possible type mismatch error.

In future versions of the Nim programming language the `static` metatype might not be required at all. It could delay the reporting of an error until the generic type is instantiated incorrectly:

```
type
  Matrix[M, N, T] = array[0..(M*N - 1), T]

var a, b: int
var m: Matrix[a, b, int]  # Error: the array size must be provided at
compile-time.
```

16.30. typedesc[T]

In many contexts, Nim treats the names of types as regular values. These values exist only during the compilation phase, but since all values must have a type, `typedesc` is considered their special type.

`typedesc` acts as a generic type. For instance, the type of the symbol `int` is `typedesc[int]`. Just like with regular generic types, when the generic param is omitted, `typedesc` denotes the type class of all types. As a syntactic convenience, one can also use `typedesc` as a modifier.

Procs featuring `typedesc` params are considered implicitly generic. They will be instantiated for each unique combination of supplied types, and within the body of the proc, the name of each param will refer to the bound concrete type:

```
proc new(T: typedesc): ref T =
  echo "allocating ", T.name
  new(result)

var n = Node.new
var tree = new(BinaryTree[int])
```

When multiple type params are present, they will bind freely to different types. To force a bind-once behavior, one can use an explicit generic param:

```
proc acceptOnlyTypePairs[T, U](A, B: typedesc[T]; C, D: typedesc[U])
```

Once bound, type params can appear in the rest of the proc signature:

```
template declareVariableWithType(T: typedesc, value: T) =
  var x: T = value

declareVariableWithType int, 42
```

Overload resolution can be further influenced by constraining the set of types that will match the type param:

```
template maxval(T: typedesc[int]): int = high(int)
template maxval(T: typedesc[float]): float = Inf

var i = int.maxval
var f = float.maxval
when false:
  var s = string.maxval # error, maxval is not implemented for string
```

16.31. typeof

> `typeof(x)` can for historical reasons also be written as `type(x)` but `type(x)` is discouraged.

One can obtain the type of a given expression by constructing a `typeof` value from it (in many other languages this is known as the `typeof` operator):

```
var x = 0
var y: typeof(x) # y has type int
```

If `typeof` is used to determine the result type of a routine call `c(X)` (where `X` stands for a possibly empty list of arguments), the interpretation where `c` is an iterator is preferred over the other interpretations, but this behavior can be changed by passing `typeOfProc` as the second argument to `typeof`:

```
iterator split(s: string): string = discard
proc split(s: string): seq[string] = discard

# since an iterator is the preferred interpretation, `y` has the type
`string`:
assert typeof("a b c".split) is string
assert typeof("a b c".split, typeOfProc) is seq[string]
```

`typedesc[T]` provides a mechanism for inferring the return type which cannot be overloaded. The interaction between `typeof`, overloading, iterators, `typedesc[T]` and generics allows for idioms that are not obvious to the casual user of the language. Here is an example showing how to map JSON data to a generic object type.

```
import std / json

proc fromJ[T: enum](t: typedesc[T]; j: JsonNode): T {.inline.} = T(j.getInt)
①
proc fromJ(t: typedesc[string]; j: JsonNode): string {.inline.} = j.getStr
proc fromJ(t: typedesc[bool]; j: JsonNode): bool {.inline.} = j.getBool
proc fromJ(t: typedesc[int]; j: JsonNode): int {.inline.} = int(j.getInt)
proc fromJ(t: typedesc[float]; j: JsonNode): float {.inline.} = j.getFloat

proc fromJ[T: seq](t: typedesc[T]; j: JsonNode): T = ②
  result = newSeq[typeof(result[0])]()
  assert j.kind == JArray
  for elem in items(j):
    result.add fromJ(typeof(result[0]), elem) ③
```

```
proc fromJ[T: object](t: typedesc[T]; j: JsonNode): T = ④
  result = T()
  assert j.kind == JObject
  for name, loc in fieldPairs(result): ⑤
    if j.hasKey(name):
      loc = fromJ(typeof(loc), j[name]) ⑥
```

① The `fromJ` family of procs supports the loading of enums, `int`, `bool`, `float`, `string` from JSON.

② A `seq` can also be loaded from JSON.

③ Depending on the sequence element's type call the correct overloaded `fromJ` proc. `typeof(result[0])` is passed to the `typedesc[T]` parameter enabling static dispatching.

④ An `object` can also be loaded from JSON.

⑤ Iterate over every field of the object via `fieldPairs`.

⑥ `fieldPairs` allows for the mutation of the object that is iterated over so that `loc = …` can be read as `result.<field> = …`.

Note that this solution allows to load complex objects which have fields that themselves are objects or primitives or sequences thereof. The compiler will produce type specialized code with no runtime overhead because the dispatching is resolved at compile-time.

Chapter 17. Type relations

The following section defines several relations on types that are needed to describe how the type checking is done in Nim.

17.1. Type equality

Nim uses structural type equivalence for most types. Only for objects, enumerations and distinct types and for generic types name equivalence is used.

17.2. Subtype relation

If an object type `B` inherits from `A`, `B` is a *subtype* of `A`. For example:

```
type
  A = object of RootObj
  B = object of A

# B is a subtype of A.
```

This means an object of type `B` can be passed to routines that expect the type `A`.

This subtype relation is extended to the types `var`, `ref`, `ptr`. If `B` is a subtype of `A` and `B` and `A` are `object` types then:

- `var A` is a subtype of `var B`
- `ref A` is a subtype of `ref B`
- `ptr A` is a subtype of `ptr B`.

If the subtype relation exists among `ref` or `ptr` types they are assignment compatible; an object of a subtype can be assigned to a location that is of the supertype:

```
type
  Shape = ref object of RootObj
  Circle = ref object of Shape

var a: Shape = Circle() # Circle is a subtype of Shape and can be assigned
to an l-value of type Shape.
```

The subtype relation does not extend from `A` and `B` to `var ref A` and `var ref B`. Doing so would open a hole in the type system:

```
type
  A = ref object of RootObj
  B = ref object of A
    field: string

proc init(a: var A) =
  a = A()

var b = B()
b.init()
echo b.field # crash here? b now points to an A, not a B
```

17.3. Convertible relation

A type `a` is *implicitly* convertible to type `b` if and only if the following algorithm returns true:

```
proc isImplicitlyConvertible(a, b: PType): bool =
  if isSubtype(a, b):
    return true
  if isIntLiteral(a):
    return b in {int8, int16, int32, int64, int, uint, uint8, uint16,
                 uint32, uint64, float32, float64}
  case a.kind
  of int:     result = b in {int32, int64}
  of int8:    result = b in {int16, int32, int64, int}
  of int16:   result = b in {int32, int64, int}
  of int32:   result = b in {int64, int}
  of uint:    result = b in {uint32, uint64}
  of uint8:   result = b in {uint16, uint32, uint64}
  of uint16:  result = b in {uint32, uint64}
```

```
  of uint32:  result = b in {uint64}
  of float32: result = b in {float64}
  of float64: result = b in {float32}
  of seq:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
  of array:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
    if a.baseType == char and a.indexType.rangeA == 0:
      result = b == cstring
  of cstring, ptr:
    result = b == pointer
  of string:
    result = b == cstring
  of proc:
    result = typeEquals(a, b) or compatibleParametersAndEffects(a, b)
```

We used the predicate `typeEquals(a, b)` for the "type equality" property and the predicate `isSubtype(a, b)` for the "subtype relation". `compatibleParametersAndEffects(a, b)` is currently not specified.

Implicit conversions are also performed for Nim's `range` type constructor.

Let `a0`, `b0` of type `T`.

Let `A = range[a0..b0]` be the argument's type, `F` the formal parameter's type. Then an implicit conversion from `A` to `F` exists if `a0 >= low(F) and b0 <= high(F)` and both `T` and `F` are signed integers or if both are unsigned integers.

A type `a` is *explicitly* convertible to type `b` if and only if the following algorithm returns true:

```
proc isIntegralType(t: PType): bool =
  result = isOrdinal(t) or t.kind in {float, float32, float64}

proc isExplicitlyConvertible(a, b: PType): bool =
  isImplicitlyConvertible(a, b) or
    typeEquals(a, b) or
    a == distinct and typeEquals(a.baseType, b) or
    b == distinct and typeEquals(b.baseType, a) or
    (isIntegralType(a) and isIntegralType(b)) or
    isSubtype(a, b) or
    isSubtype(b, a)
```

The convertible relation can be extended by a user-defined type `converter`.

```nim
converter toInt(x: char): int = result = ord(x)

var
  x: int
  chr: char = 'a'

# implicit conversion magic happens here
x = chr
echo x # => 97
# one can use the explicit form too
x = chr.toInt
echo x # => 97
```

The type conversion `T(a)` is an L-value if `a` is an L-value and if `a` is of type `T` or a distinct type of `T` or if `T` is a distinct type of `typeof(a)`.

17.4. Assignment compatibility

An expression `b` can be assigned to an expression `a` if and only if `a` is an `l-value` and `isImplicitlyConvertible(b.typ, a.typ)` holds.

Chapter 18. Constant expressions

In certain contexts, like `array[E, T]` type declarations, Nim requires the expression `E` to be `constant`. A constant expression is defined as follows:

1. Literals are constant expressions.
2. If `a` and `b` are constant expressions, so is a <binaryop> b and <unaryop> a.

<unaryop> can be:

- A type conversion that can be interpreted as a type annotation like `int(4)`, `MyTuple((a, b, c))`, `ObjRef(nil)`.
- Unary minus `system.-` for the builtin types.
- Unary `system.not` for the builtin types.
- `system.succ`.
- `system.pred`.

<binaryop> can be:

- `system.+`, `system.-`, `system.*`, `system.mod`, `system.div`, `system.shr`, `system.shl`, `system.max`, `system.min`.

The current Nim implementation goes much farther than this definition and considers any expression to be constant that it can evaluate at compile-time via its powerful virtual machine. For example, the standard library's `math` module contains:

```
func createFactTable[N: static[int]]: array[N, int] = ①
  result[0] = 1
  for i in 1 ..< N:
    result[i] = result[i - 1] * i
```

```nim
func fac*(n: int): int =
  ## Computes the factorial of a non-negative integer `n`.
  runnableExamples:
    doAssert fac(0) == 1
    doAssert fac(4) == 24
    doAssert fac(10) == 3628800

  const factTable = ②
    when sizeof(int) == 2:
      createFactTable[5]()
    elif sizeof(int) == 4:
      createFactTable[13]()
    else:
      createFactTable[21]()
  assert(n >= 0, $n & " must not be negative.")
  assert(n < factTable.len, $n & " is too large to look up in the table")
  factTable[n]  ③
```

① `createFactTable` computes a lookup table at compile-time.

② `factTable` is the lookup table. Its size depends on the size of the `int` type.

③ The `fac` implementation does not do any computation; instead the `factTable` is queried for the precomputed result.

The mechanisms of compile-time evaluation are the foundation for Nim's macro system.

Chapter 19. Overload resolution

In a call `p(args)` the routine `p` that matches best is selected. If multiple routines match equally well, the ambiguity is reported during semantic analysis.

Every `arg` in `args` needs to match. Let `f` be the formal parameter's type and `a` the type of the argument. There are multiple different categories how an argument can match:

1. Exact match: `a` and `f` are of the same type.
2. Literal match: `a` is an integer literal of value `v` and `f` is a signed or unsigned integer type and `v` is in `f`'s range. Or: `a` is a floating-point literal of value `v` and `f` is a floating-point type and `v` is in `f`'s range.
3. Generic match: `f` is a generic type and `a` matches, for instance `a` is `int` and `f` is a generic (constrained) parameter type (like in `[T]` or `[T: int|char]`).
4. Subrange or subtype match: `a` is a `range[T]` and `T` matches `f` exactly. Or: `a` is a subtype of `f`.
5. Integral conversion match: `a` is convertible to `f` and `f` and `a` is some integer or floating-point type.
6. Conversion match: `a` is convertible to `f`, possibly via a user defined `converter`.

These matching categories have a priority: An exact match is better than a literal match and that is better than a generic match etc. In the following, `count(p, m)` counts the number of matches of the matching category `m` for the routine `p`.

A routine `p` matches better than a routine `q` if the following algorithm returns true:

```
for each matching category m in ["exact match", "literal match",
                                "generic match", "subtype match",
                                "integral match", "conversion match"]:
  if count(p, m) > count(q, m): return true
  elif count(p, m) == count(q, m):
    discard "continue with next category m"
  else:
    return false
return "ambiguous"
```

Some examples:

```
proc takesInt(x: int) = echo "int"
proc takesInt[T](x: T) = echo "T"
proc takesInt(x: int16) = echo "int16"

takesInt(4) # "int"
var x: int32
takesInt(x) # "T"
var y: int16
takesInt(y) # "int16"
var z: range[0..4] = 0
takesInt(z) # "T"
```

If this algorithm returns "ambiguous" further disambiguation is performed: If the argument `a` matches both the parameter type `f` of `p` and `g` of `q` via a subtyping relation, the inheritance depth is taken into account:

```
type
  A = object of RootObj
  B = object of A
  C = object of B

proc p(obj: A) =
  echo "A"

proc p(obj: B) =
  echo "B"

var c = C()
# not ambiguous, calls 'B', not 'A' since B is a subtype of A
# but not vice versa:
p(c)

proc pp(obj: A, obj2: B) = echo "A B"
proc pp(obj: B, obj2: A) = echo "B A"
```

```nim
# but this is ambiguous:
pp(c, c)
```

Likewise, for generic matches, the most specialized generic type (that still matches) is preferred:

```nim
proc gen[T](x: ref ref T) = echo "ref ref T"
proc gen[T](x: ref T) = echo "ref T"
proc gen[T](x: T) = echo "T"

var ri: ref int
gen(ri) # "ref T"
```

Nim is based on overloading. Overloading is not just “syntactic sugar”, it is essential for static polymorphism.

The follow example outlines how a family of procs called `toJ` can be used to load arbitrarily typed data:

```nim
import std / json

proc toJ[T: enum](e: T): JsonNode = newJInt(ord e) ①
proc toJ(s: string): JsonNode {.inline.} = newJString(s) ②
proc toJ(b: bool): JsonNode {.inline.} = newJBool(b)
proc toJ(f: float): JsonNode {.inline.} = newJFloat(f)
proc toJ(i: int): JsonNode {.inline.} = newJInt(i)

proc toJ[T](s: seq[T]): JsonNode = ③
  result = newJArray()
  for x in s:
    result.add toJ(x) ④

proc toJ[T: object](obj: T): JsonNode = ⑤
  result = newJObject()
  for f, v in fieldPairs(obj): ⑥
    result[f] = toJ(v) ⑦
```

① Enum values are mapped to JSON by using their ordinal values.

② `string`, `bool`, `float` and `int` are mapped to their JSON equivalents.

③ `seq[T]` is mapped to a JSON array.

④ Depending on the sequence element’s type call the correct overloaded `toJ` proc.

⑤ An `object` can also be loaded from JSON.

⑥ Iterate over every field of the object via `fieldPairs`.

⑦ Call the overloaded `toJ` proc for every field `v` of `obj`.

19.1. Overloading based on 'var T'

If the formal parameter `f` is of type `var T` in addition to the ordinary type checking, the argument is checked to be an *l-value*. `var T` matches better than just `T` then.

```
proc sayHi(x: int): string =
  # matches a non-var int
  result = $x
proc sayHi(x: var int): string =
  # matches a var int
  result = $(x + 10)

proc sayHello(x: int) =
  var m = x # a mutable version of x
  echo sayHi(x) # matches the non-var version of sayHi
  echo sayHi(m) # matches the var version of sayHi

sayHello(3) # 3
            # 13
```

19.2. Lazy type resolution for untyped

An *unresolved* expression is an expression for which no symbol lookups and no type checking was performed.

Since templates and macros participate in overloading resolution, it's essential to have a way to pass unresolved expressions to a template or macro. This is what the meta-type `untyped` accomplishes:

```
template rem(x: untyped) = discard

rem unresolvedExpression(undeclaredIdentifier)
```

A parameter of type `untyped` always matches any argument (as long as there is any argument passed to it).

But one has to watch out because other overloads might trigger the argument's resolution:

```
template rem(x: untyped) = discard
proc rem[T](x: T) = discard

# undeclared identifier: 'unresolvedExpression'
rem unresolvedExpression(undeclaredIdentifier)
```

`untyped` and `varargs[untyped]` are the only meta-type that are lazy in this sense, the other meta-types `typed` and `typedesc` are not lazy.

19.3. Varargs matching

See Section 16.15, "Varargs".

19.4. Overload disambiguation

For routine calls "overload resolution" is performed. There is a weaker form of overload resolution called *overload disambiguation* that is performed when an overloaded symbol is used in a context where there is additional type information available. Let `p` be an overloaded symbol. These contexts are:

- In a function call `q(…, p, …)` when the corresponding formal parameter of `q` is a `proc` type. If `q` itself is overloaded then the cartesian product of every interpretation of `q` and `p` must be considered.
- In an object constructor `Obj(…, field: p, …)` when `field` is a `proc` type. Analogous rules exist for array/set/tuple constructors.
- In a declaration like `x: T = p` when `T` is a `proc` type.

As usual, ambiguous matches produce a compile-time error.

Chapter 20. Statements and expressions

Nim uses the common statement/expression paradigm: Statements do not produce a value, in contrast to expressions. However, some expressions are statements.

Statements are separated into *simple statements* and *complex statements*. Simple statements are statements that cannot contain other statements like assignments, calls, or the `return` statement; complex statements can contain other statements. To avoid the *dangling else problem*, complex statements always have to be indented. The details can be found in the grammar.

20.1. Statement list expression

Statements can also occur in an expression context that looks like `(stmt1; stmt2; ...; ex)`. This is called a statement list expression or `(;)`. The type of `(stmt1; stmt2; ...; ex)` is the type of `ex`. All the other statements must be of type `void`. (One can use `discard` to produce a `void` type.) `(;)` does not introduce a new scope.

20.2. Discard statement

Example:

```
proc p(x, y: int): int =
  result = x + y

discard p(3, 4) # discard the return value of `p`
```

The `discard` statement evaluates its expression for side-effects and throws the expression's resulting value away, and should only be used when ignoring this value is known not to cause problems.

Ignoring the return value of a routine without using a discard statement is a static error.

The return value can be ignored implicitly if the called routine has been declared with the `discardable` pragma:

```
proc p(x, y: int): int {.discardable.} =
  result = x + y

p(3, 4) # now valid
```

An empty `discard` statement is often used as a null statement:

```
proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: discard
```

The `discardable` pragma is not available for templates or macros. This is a consequence of the fact that templates and macros are expanded directly at their call sites: The expanded expression does not contain the information that it was `discardable`.

20.3. Void context

In a list of statements, every expression except the last one needs to have the type `void`. In addition to this rule an assignment to the builtin `result` symbol also triggers a mandatory `void` context for the subsequent expressions:

```
proc invalid*(): string =
  result = "foo"
  "invalid"  # Error: value of type 'string' has to be discarded
```

```
proc valid*(): string =
  let x = 317
  "valid"
```

20.4. Var statement

Var statements declare new local and global variables and initialize them. A comma-separated list of variables can be used to specify variables of the same type:

```
var
  a: int = 0
  x, y, z: int
```

If an initializer is given, the type can be omitted: the variable is then of the same type as the initializing expression. Variables are always initialized with a default value if there is no initializing expression. The default value depends on the type and is always a zero in binary.

There is one exception from this rule: Assuming `g()` returns a `var T` type then the `x` of `var x = g()` is of type `T`. This can imply a copy operation:

```
proc `[]`(x: var Container[T]; i: int): var T

proc access(c: var Container[int]) =
  var x = c[0] ①
  x = 4 ②
  echo c[0] ③
```

① Even though `[]` returns a `var int` `x` is of type `int`. The value of `c[0]` is copied into `x`.

② `x` is modified but `c[0]` is not.

③ `c[0]` is unmodified.

Table 8. Default values

Type	default value
any integer type	`0`
any float	`0.0`
char	`'\0'`
bool	`false`
ref or pointer type	`nil`

Type	default value
procedural type	`nil`
sequence	`@[]`
string	`""`
tuple[x: A, y: B, ...]	`(default(A), default(B), ...)` (analogous for objects)
array[0..., T]	`[default(T), ...]`
range[T]	`default(T)`; this may be out of the valid range
T = enum	`cast[T](0)`; this may be an invalid value

The implicit initialization can be avoided for optimization reasons with the `noinit` pragma:

```
var
  a {.noInit.}: array[0..1023, char]
```

If a proc is annotated with the `noinit` pragma, this refers to its implicit `result` variable:

```
proc returnUndefinedValue: int {.noinit.} = discard
```

The implicit initialization can also be prevented by the `requiresInit` type pragma. With `requiresInit` an explicit initialization for an object and all of its fields is required. However, it does a form of *control flow analysis* to prove the variable was initialized:

```
type
  MyObject = object {.requiresInit.}

proc use(x: MyObject) = discard

proc valid() =
  var x: MyObject
  if someCondition():
    x = a()
  else:
    x = a()
  # valid: all paths set `x` before it is used:
  use x
```

```nim
proc invalid() =
  var x: MyObject
  if someCondition():
    x = a()
  # invalid: not all paths set `x` before it is used:
  use x
```

The `requiresInit` pragma can be applied to `object` or `distinct` types.

20.5. Let statement

A `let` statement declares new local and global *single assignment* variables and binds a value to them. The syntax is the same as that of the `var` statement, except that the keyword `var` is replaced by the keyword `let`. Let variables are not l-values and can thus not be passed to `var` parameters nor can their address be taken. They cannot be assigned new values.

For let variables, the same pragmas are available as for ordinary variables.

As `let` statements are immutable after creation they need to define a value when they are declared. The only exception to this rule is if the `{.importc.}` pragma (or any of the other `importX` pragmas) is applied, in this case the value is expected to come from a foreign source, typically a C/C++ `const`.

20.6. Tuple unpacking

In a `var` or `let` statement *tuple unpacking* can be performed. The special identifier `_` can be used to ignore some parts of the tuple:

```nim
proc returnsTuple(): (int, int, int) = (4, 2, 3)

let (x, _, z) = returnsTuple()
```

20.7. Const statement

A `const` statement declares constants whose values are constant expressions:

```
import std/[strutils]
const
  roundPi = 3.1415
  constEval = contains("abc", 'b') # computed at compile time!
```

A constant is a constant expression.

See Chapter 18, *Constant expressions* for further details.

A good example for a more complex `const` is a lookup table:

```
const
  LookupTable = {
    "one": (1, false, "a"),
    "two": (2, true, "b"),
    "three": (3, false, "c")
  }
```

There are restrictions on how these can be formed, however: The type `ref` cannot be part of a constant:

```
type
  T = ref object
    a, b: int

const
  LookupTable = {
    "invalid": T(a: 2, b: 3)
  }
```

The reason for this is that Nim's type system in general lacks constant pointers, the data that a pointer points to is always mutable. Later versions of the language might provide such a facility.

20.8. Type section

New types in Nim are introduced within `type` section. A newly introduced type can be one of:

1. Type aliases

```
type
  NewNameForOldThing = int
  IntArray = openArray[int]
```

2. Nominal types (every nominal type must be introduced via a type section):

```
type
  Sex = enum
    Male, Female

  Id = distinct string

  Person = object
    id: Id
    name: string
    sex: Sex
    age: Natural
```

The nominal types are `object`, `ref object`, `ptr object`, `distinct` and `enum`.

3. Generic types:

```
type
  KeyValuePair[A, B] = tuple[hcode: Hash, key: A, val: B]
  KeyValuePairSeq[A, B] = seq[KeyValuePair[A, B]]
  Table*[A, B] = object
    data: KeyValuePairSeq[A, B]
    counter: int
  TableRef*[A, B] = ref Table[A, B]
```

Mutually recursive types have to be in the same type section, for example:

```nim
# example demonstrating mutually recursive types
type
  Node = ref object  # a Node in a tree
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

  Sym = object       # a symbol
    name: string     # the symbol's name
    line: int        # the line the symbol was declared in
    code: Node       # the symbol's abstract syntax tree
```

In summary:

- A type section begins with the `type` keyword.
- It contains multiple type definitions.
- A type definition binds a type to a name.
- Type definitions can be recursive or even mutually recursive.
- Mutually recursive types are only possible within a single `type` section.
- Nominal types like `objects` or `enums` can only be defined in a `type` section.

20.9. Static statement/expression

A static statement/expression explicitly requests compile-time execution. Even code that has side effects is permitted in a static block:

```nim
static:
  echo "echo at compile time"
```

There are limitations on what Nim code can be executed at compile time; the limitations change with every release of the Nim compiler and are beyond the scope of this book.

It is a static error if the Nim compiler cannot execute the block at compile time.

20.10. If statement

Example:

```
var name = readLine(stdin)

if name.endsWith('a'):
  echo "What a nice name!"
elif name == "":
  echo "Don't you have a name?"
else:
  echo "Boring name..."
```

The `if` statement is a simple way to make a branch in the control flow: The expression after the keyword `if` is evaluated, if it is true the corresponding statements after the `:` are executed. Otherwise the expression after the `elif` is evaluated (if there is an `elif` branch), if it is true the corresponding statements after the `:` are executed. This goes on until the last `elif`. If all conditions fail, the `else` part is executed. If there is no `else` part, execution continues with the next statement.

In `if` statements, new scopes begin immediately after the `if`/`elif`/`else` keywords and end after the corresponding *then* block.

For visualization purposes the scopes are enclosed in `{|  |}` in the following example:

```
if {| (let m = input =~ re"(\w+)=\w+"; m.isMatch):
  echo "key ", m[0], " value ", m[1]  |}
elif {| (let m = input =~ re""; m.isMatch):
  echo "new m in this scope"  |}
else: {|
  echo "m not declared here"  |}
```

20.11. Case statement

Example:

```
let line = readLine(stdin)
case line
of "delete-everything", "restart-computer":
  echo "permission denied"
of "go-for-a-walk":
  echo "please yourself"
elif line.len == 0:
  echo "empty" # optional, must come after `of` branches
else:
  echo "unknown command" # ditto

# indentation of the branches is also allowed; and so is an optional colon
# after the selecting expression:
case readLine(stdin):
  of "delete-everything", "restart-computer":
    echo "permission denied"
  of "go-for-a-walk":
    echo "please yourself"
  else:
    echo "unknown command"
```

The `case` statement is similar to the `if` statement, but it represents a multi-branch selection. The expression after the keyword `case` is evaluated and if its value is in a *slicelist* the corresponding statements (after the `of` keyword) are executed. If the value is not in any given *slicelist*, trailing `elif` and `else` parts are executed using same semantics as for `if` statement, and `elif` is handled just like `else: if`. If there are no `else` or `elif` parts and not all possible values that `expr` can hold occur in a *slicelist*, a static error occurs. This holds only for expressions of ordinal types. "All possible values" of `expr` are determined by `expr`'s type. To suppress the static error an `else: discard` should be used.

For non-ordinal types, it is not possible to list every possible value and so these always require an `else` part. An exception to this rule is for the `string` type, which doesn't require a trailing `else` or `elif` branch; it's unspecified whether this will keep working in future versions.

Because case statements are checked for exhaustiveness during semantic analysis, the value in every `of` branch must be a constant expression. This restriction also allows a Nim compiler to generate more performant code.

As a special semantic extension, an expression in an `of` branch of a case statement may evaluate to a set or array constructor; the set or array is then expanded into a list of its elements:

```
const
  SymChars: set[char] = {'a'..'z', 'A'..'Z', '\x80'..'\xFF'}

proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"

# is equivalent to:
proc classify(s: string) =
  case s[0]
  of 'a'..'z', 'A'..'Z', '\x80'..'\xFF', '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"
```

20.12. When statement

Example:

```
when sizeof(int) == 2:
  echo "running on a 16 bit system!"
elif sizeof(int) == 4:
  echo "running on a 32 bit system!"
elif sizeof(int) == 8:
  echo "running on a 64 bit system!"
else:
  echo "cannot happen!"
```

The `when` statement is almost identical to the `if` statement with some exceptions:

- Each condition (`expr`) has to be a constant expression (of type `bool`).
- The statements do not open a new scope.
- The statements that belong to the expression that evaluated to true are processed, the other statements are not checked for semantics! However, each condition is checked for semantics.

The `when` statement enables conditional compilation techniques. The `when` construct is also available within `object` definitions:

```
type
  Option*[T] = object
    ## An optional type that may or may not contain a value of type `T`.
    ## When `T` is a a pointer type (`ptr`, `pointer`, `ref` or `proc`),
    ## `none(T)` is represented as `nil`.
    when T is SomePointer:
      val: T
    else:
      val: T
      has: bool
```

20.13. Return statement

Example:

```
return 40+2
```

The `return` statement ends the execution of the current routine. If there is a value to return, this is syntactic sugar for:

```
result = expr
return result
```

`return` without an expression is a short notation for `return result` if the routine has an implicit `result` variable.

The `result` variable is always the return value of the procedure.

The `return` statement raises a special `return` exception. `break` and `return` statements are defined in terms of raising an exception in order to specify their interactions with the exception handling statements. The `return` exception is a pseudo-exception otherwise, it is ignored for Nim's effect system.

20.14. Yield statement

Example:

```
yield (1, 2, 3)
```

The `yield` statement is used instead of the `return` statement in iterators. It is only valid in iterators. Execution is returned to the body of the for loop that called the iterator. Yield does not end the iteration process, but the execution is passed back to the iterator if the next iteration starts. See (Chapter 23, *Iterators and the for statement*) for further information.

20.15. Block statement

Example:

```
var found = false
block myblock:
  for i in 0..3:
    for j in 0..3:
      if a[j][i] == 7:
        found = true
        break myblock # leave the block, in this case both for-loops
echo found
```

The block statement is a means to group statements to a (named) `block`. Inside the block, the `break` statement is allowed to leave the block immediately. A `break` statement can contain a name of a surrounding block to specify which block should be left.

20.16. Break statement

Example:

```
while cond:
  break # leave the `while` loop

block symbol:
  break symbol # leave the block named `symbol`
```

The `break` statement is used to leave a block immediately. If `symbol` is given, it is the name of the enclosing block that is to be left. If it is absent, the innermost block is left. A `break` statement must be textually enclosed by a `block`, `while` or `for` statement.

The `break` statement raises a `break` exception. `break` and `return` statements are defined in terms of raising an exception in order to specify their interactions with the exception handling statements. The `break` exception is a pseudo-exception otherwise, it is ignored for Nim's effect system.

20.17. While statement

Example:

```
echo "Please tell me your password:"
var pw = readLine(stdin)
while pw != "12345":
  echo "Wrong password! Next try:"
  pw = readLine(stdin)
```

The `while` statement is executed until the condition evaluates to false. Endless loops are no error. `while` statements open an `implicit block` so that they can be left with a `break` statement.

20.18. Continue statement

A `continue` statement leads to the immediate next iteration of the surrounding loop construct. It is only allowed within a loop. A continue statement is syntactic sugar for a nested block:

```
while expr1:
  stmt1
  continue
  stmt2
```

Is equivalent to:

```
while expr1:
  block myBlockName:
    stmt1
    break myBlockName
    stmt2
```

20.19. Using statement

The `using` statement provides syntactic convenience in modules where the same parameter names and types are used over and over. Instead of:

```
proc foo(c: Context; n: Node) = ...
proc bar(c: Context; n: Node, counter: int) = ...
proc baz(c: Context; n: Node) = ...
```

One can specify the convention that a parameter of name `c` should default to type `Context`, `n` should default to `Node` etc.:

```
using
  c: Context
  n: Node
  counter: int

proc foo(c, n) = ...
proc bar(c, n, counter) = ...
proc baz(c, n) = ...
```

The `using` section uses the same indentation based grouping syntax as a `var` or `let` statements.

Note that `using` is not applied for `template` since the untyped template parameters default to the type `system.untyped`.

Mixing parameters that should use the `using` declaration with parameters that are explicitly typed is possible and requires a semicolon between them:

```
proc mixedMode(c, n; x, y: int) =
  # 'c' is inferred to be of the type 'Context'
  # 'n' is inferred to be of the type 'Node'
  # But 'x' and 'y' are of type 'int'.
```

20.20. If expression

An `if` expression is almost like an if statement, but it is an expression which means that it produces a value. Example:

```
var y = if x > 8: 9 else: 10
```

An if expression always results in a value, so the `else` part is required. `Elif` parts are also allowed.

20.21. When expression

Just like an `if` expression there is a `when` expression available:

```
const
  PathSep* = when defined(Windows): '\\'
             else: '/'
```

20.22. Case expression

The `case` expression is again very similar to the case statement:

```
var favoriteFood =
  case animal
  of "dog": "bones"
  of "cat": "mice"
  elif animal.endsWith"whale": "plankton"
  else:
    echo "I'm not sure what to serve, but everybody loves ice cream"
    "ice cream"
```

When multiple statements are given for a branch, the last expression as the result value is used.

The `case` expression does not produce an l-value, so the following example does not work:

```
type Foo = ref object
  x: seq[string]

proc getX(x: Foo): var seq[string] =
  case true
  of true:
    x.x   # invalid: case does not produce an l-value
  else:
    x.x

var foo = Foo(x: @[])
foo.getX().add("asd")
```

Instead an explicit `result` or `return` has to be used:

```
proc getX(x: Foo): var seq[string] =
  case true
  of true:
    result = x.x
  else:
    result = x.x
```

20.23. Block expression

A *block expression* is a block statement that produces a value. The evaluation of the block's last expression produces this value.

Like in a block statement, the `block` keyword introduces a new scope.

For example:

```
let a = block:
  var fib = @[0, 1] # `fib` is not accessible outside of the `block`
  for i in 0..10:
    fib.add fib[^1] + fib[^2]
  fib
```

20.24. Table constructor

A table constructor is syntactic sugar for an array constructor:

```
{"key1": "value1", "key2", "key3": "value2"}

# is the same as:
[("key1", "value1"), ("key2", "value2"), ("key3", "value2")]
```

The empty table can be written `{:}` (in contrast to the empty set which is `{}`) which is thus another way to write the empty array constructor `[]`.

This slightly unusual way of supporting tables has lots of advantages:

- The order of the (key,value)-pairs is preserved, thus it is easy to support ordered dicts with for example `{key: val}.newOrderedTable`.
- A table literal can be put into a readonly section and it requires a

minimal amount of memory.

- Every table implementation is treated equally syntactically.
- Apart from the minimal syntactic sugar, the language core does not need to contain more support for tables.

20.25. Type conversions

Syntactically a *type conversion* is like a routine call, but a type name replaces the routine name. A type conversion is always safe in the sense that a failure to convert a type to another results in an exception (if it cannot be determined statically).

Ordinary procs are often preferred over type conversions in Nim: For instance, `$` is the `toString` operator by convention and `toFloat` and `toInt` can be used to convert from floating-point to integer or vice versa.

Type conversion can also be used to disambiguate overloaded routines:

```
proc p(x: int) = echo "int"
proc p(x: string) = echo "string"

let procVar = (proc(x: string))(p)
procVar("a")
```

Since operations on unsigned numbers wrap around and are unchecked so are type conversions to unsigned integers and between unsigned integers.

Exception: Values that are converted to an unsigned type at compile time are checked so that code like `byte(-1)` does not compile.

20.26. Type casts

Type casts are a crude mechanism to interpret the bit pattern of an expression to be of another type. Type casts are only needed for low-level programming and are inherently unsafe. A type cast is written as `cast[T](x)` where `x` is a value to be interpreted as to be of type `T`.

Example:

```
type
  Obj = object
    a: int32   # at offset 0
    b: int32   # at offset 4

var obj: Obj

cast[ptr int32](cast[uint](addr obj) + 4)[] = 123
# On most machine architectures this is a
# convoluted, inherently weakly portable
# way for doing:
obj.b = 123
```

20.27. The addr operator

The `addr` operator returns the address of an l-value. If the type of the location is `T`, the `addr` operator result is of the type `ptr T`. An address is always an untraced pointer. Taking the address of an object is *unsafe*, as the pointer may live longer than the object and can thus reference a non-existing object. One can get the address of an l-value:

```
let t1 = "Hello"
var
  t2 = t1
  t3 : pointer = addr(t2)
echo repr(addr(t2))
# --> ref 0x7fff6b71b670 --> 0x10bb81050"Hello"
echo cast[ptr string](t3)[]
# --> Hello
```

20.28. The unsafeAddr operator

In some versions of Nim `addr` is not allowed to be performed on inherently immutable entities such as `let` variables, routine parameters or `for` loop variables. Instead of `addr` `unsafeAddr` can be used for these entities:

```
let myArray = [1, 2, 3]
foreignProcThatTakesAnAddr(unsafeAddr myArray)
```

Note however that the name is somewhat misleading as `addr` is an unsafe operation too.

Chapter 21. Procedures

A *routine* is a symbol of kind: `proc`, `func`, `method`, `iterator`, `macro`, `template`, `converter`.

What most programming languages call *methods* or *functions* are called *procedures* in Nim. A procedure declaration consists of an identifier, zero or more formal parameters, a return value type and a block of code. Formal parameters are declared as a list of identifiers separated by either comma or semicolon. A parameter is given a type by `: typename`. The type applies to all parameters immediately before it, until either the beginning of the parameter list, a semicolon separator, or an already typed parameter, is reached. The semicolon can be used to make separation of types and subsequent identifiers more distinct.

```
# Using only commas
proc foo(a, b: int, c, d: bool): int

# Using semicolon for visual distinction
proc foo(a, b: int; c, d: bool): int

# Will fail: a is untyped since ';' stops type propagation.
proc foo(a; b: int; c, d: bool): int
```

A parameter may be declared with a default value which is used if the caller does not provide a value for the argument.

```
# b is optional with 47 as its default value
proc foo(a: int, b: int = 47): int
```

Parameters can be declared as mutable and so allow the proc to modify the corresponding arguments, by using the type modifier `var`.

```
# "returning" a value to the caller through the 2nd argument
# Notice that the function uses no actual return value; its return type is
`void`
proc foo(inp: int, outp: var int) =
  outp = inp + 47
```

If the proc declaration has no body, it is a *forward declaration*. If the proc returns a value, the procedure body can access an implicitly declared variable named `result` that represents the return value. Procs can be overloaded. The overloading resolution algorithm determines which proc is the best match for the arguments. Example:

```
proc toLower(c: char): char = # toLower for characters
  if c in {'A'..'Z'}:
    result = chr(ord(c) + (ord('a') - ord('A')))
  else:
    result = c

proc toLower(s: string): string = # toLower for strings
  result = newString(len(s))
  for i in 0..len(s) - 1:
    result[i] = toLower(s[i]) # calls toLower for characters; no recursion!
```

Calling a procedure can be done in many different ways:

```
proc callme(x, y: int, s: string = "", c: char, b: bool = false) = discard

# call with positional arguments      # parameter bindings:
callme(0, 1, "abc", '\t', true)       # (x=0, y=1, s="abc", c='\t', b=true)
# call with named and positional arguments:
callme(y=1, x=0, "abd", '\t')         # (x=0, y=1, s="abd", c='\t', b=false)
# call with named arguments (order is not relevant):
callme(c='\t', y=1, x=0)              # (x=0, y=1, s="", c='\t', b=false)
# call as a command statement: no () needed:
callme 0, 1, "abc", '\t'              # (x=0, y=1, s="abc", c='\t', b=false)
```

A procedure may call itself recursively.

Operators are procedures with a special operator symbol as identifier:

```
proc `$`(x: int): string =
  # converts an integer to a string; this is a prefix operator.
  result = intToStr(x)
```

Operators with one parameter are prefix operators, operators with two parameters are infix operators. (However, the parser distinguishes these from the operator's position within an expression.) There is no way to declare postfix operators: all postfix operators are built-in and handled by the grammar explicitly.

Any operator can be called like an ordinary proc with the \`opr\` notation. (Thus an operator can have more than two parameters):

```
proc `*+`(a, b, c: int): int =
  # Multiply and add
  result = a * b + c

assert `*+`(3, 4, 6) == `+`(`*`(a, b), c)
```

21.1. Method call syntax

The syntax `obj.methodName(args)` can be used instead of `methodName(obj, args)`. The parentheses can be omitted if there are no remaining arguments: `obj.len` (instead of `len(obj)`).

This method call syntax is not restricted to objects, it can be used to supply any type of first argument for routines:

```
echo "abc".len # is the same as echo len "abc"
echo "abc".toUpper()
echo {'a', 'b', 'c'}.card
stdout.writeLine("Hallo") # the same as writeLine(stdout, "Hallo")
```

Another way to look at the method call syntax is that it provides the missing postfix notation.

The method call syntax conflicts with explicit generic instantiations: `p[T](x)` cannot be written as `x.p[T]` because `x.p[T]` is always parsed as `(x.p)[T]`.

See also: Section 27.17, "Method call syntax limitations".

The `[: ]` notation was designed to mitigate this issue: `x.p[:T]` is rewritten by the parser to `p[T](x)`, `x.p[:T](y)` is rewritten to `p[T](x, y)`. Note that `[: ]` has no AST representation, the rewrite is performed directly in the parsing step.

21.2. Properties

Nim has no need for *get-properties*: Ordinary get-procedures that are called with the *method call syntax* achieve the same. But setting a value is different; for this we need a special setter syntax:

```
# Module asocket
type
  Socket* = ref object of RootObj
    host: int # cannot be accessed from the outside of the module

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of hostAddr.
  ## This accesses the 'host' field and is not a recursive call to
  ## `host=` because the builtin dot access is preferred if it is
  ## available:
  s.host = value

proc host*(s: Socket): int {.inline.} =
  ## getter of hostAddr
  ## This accesses the 'host' field and is not a recursive call to
  ## `host` because the builtin dot access is preferred if it is
  ## available:
  s.host
```

```
# module B
import asocket
var s: Socket
new s
s.host = 34  # same as `host=`(s, 34)
```

A proc defined as `f=` (with the trailing `=`) is called a *setter*. A setter can be called explicitly via the common back ticks notation:

```
proc `f=`(x: MyObject; value: string) =
  discard

`f=`(myObject, "value")
```

`f=` can be called implicitly in the pattern `x.f = value` if and only if the type of `x` does not have a field named `f` or if `f` is not visible in the current module. These rules ensure that object fields and accessors can have the same name. Within the module `x.f` is then always interpreted as field access and outside the module it is interpreted as an accessor proc call.

21.3. Indexing

Array-like access properties (`a[i]`) are supported too as the `[]` subscript operator can be overloaded. To offer both read and write access no less than 3 versions have to be provided:

```
proc `[]`(x: Container; i: Index): ElementType
proc `[]`(x: var Container; i: Index): var ElementType
proc `[]=`(x: Container; i: Index; newValue: ElementType)
```

For example:

```
type
  Matrix* = object
    # Array for internal storage of elements.
    data: ptr UncheckedArray[float]
    # Row and column dimensions.
    m*, n*: int

proc `[]`*(m: Matrix, i, j: int): float {.inline.} =
  ## Get a single element.
  m.data[i * m.n + j]

proc `[]`*(m: var Matrix, i, j: int): var float {.inline.} =
  ## Get a single element.
  m.data[i * m.n + j]

proc `[]=`*(m: var Matrix, i, j: int, s: float) =
  ## Set a single element.
  m.data[i * m.n + j] = s
```

21.4. Command invocation syntax

Routines can be invoked without the `()` if the call is syntactically a statement. This command invocation syntax also works for expressions, but then only a single argument may follow. This restriction means `echo f 1, f 2` is parsed as `echo(f(1), f(2))` and not as `echo(f(1, f(2)))`. The method call syntax may be used to provide one more argument in this case:

```
proc optarg(x: int, y: int = 0): int = x + y
proc singlearg(x: int): int = 20*x

echo optarg 1, " ", singlearg 2  # prints "1 40"
```

```
let fail = optarg 1, optarg 8   # Wrong. Too many arguments for a command
call
let x = optarg(1, optarg 8)  # traditional procedure call with 2 arguments
let y = 1.optarg optarg 8    # same thing as above, w/o the parenthesis
assert x == y
```

The command invocation syntax also cannot have complex expressions as arguments. For example: (Section 21.6, "Anonymous procs"), `if`, `case` or `try`. Function calls with no arguments still need `()`` to distinguish between a call and the function itself as a first-class value.

21.5. Closures

Procedures can appear at the top level in a module as well as inside other scopes, in which case they are called *nested procs*. A nested proc can access local variables from its enclosing scope and if it does so it becomes a *closure*. Any captured variables are stored in a hidden additional argument to the closure (its environment) and they are accessed by reference by both the closure and its enclosing scope (i.e. any modifications made to them are visible in both places):

```
proc outer =
  var i = 0

  proc mutate = ①
    inc i

  mutate()
  echo i ②

outer()
```

① `mutate` accesses `i` which belongs to `outer`. This access causes `mutate` to have the calling convention `.closure`.

② Outputs `1`.

21.5.1. Creating closures in loops

Since closures *capture* local variables by reference it is often not the wanted behavior inside loop bodies:

```
var later: seq[proc ()] = @[]
for i in 1..2:
  later.add proc () = echo i
for x in later: x()
# Produces: 2 2
```

In order to capture a loop variable "by value", both a helper routine and an additional local variable (which is then captured instead of the for loop variable) have to be used:

```
var later: seq[proc ()] = @[]
for i in 1..2:
    (proc =
      let j = i
      later.add proc () = echo j)()

for x in later: x()
# Produces: 1 2
```

The standard library offers the helpers `system.closureScope` and `sugar.capture` for syntactic shortcuts that accomplish the same.

21.6. Anonymous procs

Unnamed procedures can be used as lambda expressions and be passed to other routines:

```
var cities = @["Frankfurt", "Tokyo", "New York", "Kyiv"]

cities.sort(proc (x,y: string): int =
  cmp(x.len, y.len))
```

Procs as expressions can appear both as nested procs and inside top-level executable code. The `sugar` module contains a `=>` macro which enables a more succinct syntax for anonymous procedures resembling lambdas as they are in languages like JavaScript, C#, etc.

21.7. Func

The `func` keyword introduces a shortcut for a `noSideEffect` proc.

```
func binarySearch[T](a: openArray[T]; elem: T): int
```

Is short for:

```
proc binarySearch[T](a: openArray[T]; elem: T): int {.noSideEffect.}
```

See Section 26.4, "Side effects" for further information.

21.8. Non-overloadable built-ins

The following built-in procs cannot be overloaded for reasons of implementation simplicity (they require specialized semantic checking):

```
declared, defined, definedInScope, compiles, sizeof,
is, shallowCopy, getAst, astToStr, spawn, procCall
```

Thus they act more like keywords than like ordinary identifiers; unlike a keyword however, a redefinition may *shadow* the definition in the `system` module. From this list the following should not be written in dot notation `x.f` since `x` cannot be type-checked before it gets passed to `f`:

```
declared, defined, definedInScope, compiles, getAst, astToStr
```

21.9. Var parameters

The type of a parameter may be prefixed with the `var` keyword:

```
proc divmod(a, b: int; res, remainder: var int) =
  res = a div b
  remainder = a mod b

var
  x, y: int

divmod(8, 5, x, y) # modifies x and y
assert x == 1
assert y == 3
```

In the example, `res` and `remainder` are `var parameters`. Var parameters can be modified by the procedure and the changes are visible to the caller. The argument passed to a var parameter has to be an l-value. Var parameters are implemented as hidden pointers. The above example is comparable to:

```
proc divmod(a, b: int; res, remainder: ptr int) =
  res[] = a div b
  remainder[] = a mod b

var
  x, y: int
divmod(8, 5, addr(x), addr(y))
assert x == 1
assert y == 3
```

In the examples, var parameters or pointers are used to provide two return values. This can be done in a cleaner way by returning a tuple:

```
proc divmod(a, b: int): tuple[res, remainder: int] =
  (a div b, a mod b)

var t = divmod(8, 5)

assert t.res == 1
assert t.remainder == 3
```

One can use tuple unpacking to access the tuple's fields:

```
var (x, y) = divmod(8, 5) # tuple unpacking
assert x == 1
assert y == 3
```

`var` parameters are never necessary for efficient parameter passing. Since non-var parameters cannot be modified, a Nim compiler is always free to pass arguments by reference if it considers it can speed up execution.

21.10. Var return type

A routine that is not a `template` nor a `macro` may return a `var` type which means that the returned value is an l-value and can be modified by the caller:

```
var g = 0
```

```nim
proc writeAccessToG(): var int =
  result = g

writeAccessToG() = 6
assert g == 6
```

It is a static error if the implicitly introduced pointer could be used to access a location beyond its lifetime:

```nim
proc writeAccessToG(): var int =
  var g = 0
  result = g # Error!
```

For iterators, a component of a tuple return type can have a `var` type too:

```nim
iterator mpairs(a: var seq[string]): tuple[key: int, val: var string] =
  for i in 0..a.high:
    yield (i, a[i])
```

In the standard library every name of a routine that returns a `var` type starts with the prefix `m` per convention.

Memory safety for returning by `var T` is ensured by a simple borrowing rule: If `result` does not refer to a location pointing to the heap (that is in `result = X` the `X` involves a `ptr` or `ref` access) then it has to be derived from the routine's first parameter:

```nim
proc forward[T](x: var T): var T =
  result = x # ok, derived from the first parameter.

proc p(param: var int): var int =
  var x: int
  # we know 'forward' provides a view into the location derived from
  # its first argument 'x'.
  result = forward(x) # Error: location is derived from `x`
                      # which is not p's first parameter and lives
                      # on the stack.
```

In other words, the lifetime of what `result` points to is attached to the lifetime of the first parameter and that is enough knowledge to verify memory safety at the call site.

Chapter 22. Methods

Methods are the only construct in Nim that uses *dynamic dispatch*, all other routines use *static dispatch*. Dynamic dispatch means that the runtime type of objects does influence which operation is performed.

For dynamic dispatch to work on an object it should be a reference type.

```
type
  Expression = ref object of RootObj ## \
    ## abstract base class for an expression
  Literal = ref object of Expression
    x: int
  PlusExpr = ref object of Expression
    a, b: Expression

method eval(e: Expression): int {.base.} =
  # override this base method
  raise newException(CatchableError, "Method without override")

method eval(e: Literal): int = return e.x

method eval(e: PlusExpr): int =
  # watch out: relies on dynamic binding
  result = eval(e.a) + eval(e.b)

proc newLit(x: int): Literal = Literal(x: x)

proc newPlus(a, b: Expression): PlusExpr = PlusExpr(a: a, b: b)

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

In the example the constructors `newLit` and `newPlus` are procs because they should use static binding, but `eval` is a method because it requires dynamic binding.

As can be seen in the example, base methods have to be annotated with the `base` pragma. The `base` pragma also acts as a reminder for the programmer that a base method `m` is used as the foundation to determine all the effects that a call to `m` might cause.

Generic methods are not supported.

Compile-time execution is not supported for methods.

22.1. Static method calls via procCall

Dynamic method resolution can be inhibited via the builtin `system.procCall`. This is somewhat comparable to the `super` keyword that traditional OOP languages offer.

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method m(a: Thing) {.base.} =
  echo "base"

method m(a: Unit) =
  # Call the base method:
  procCall m(Thing(a))
  echo "1"
```

Chapter 23. Iterators and the for statement

The `for` statement is an abstract mechanism to iterate over the elements of a container. It relies on an *iterator* to do so. Like `while` statements, `for` statements open an *implicit block* so that they can be left with a `break` statement.

The `for` loop declares iteration variables - their scope reaches until the end of the loop body. The iteration variables' types are inferred by the return type of the iterator.

An iterator is similar to a procedure, except that it can be called in the context of a `for` loop. Iterators provide a way to specify the iteration over an abstract type. The `yield` statement in the called iterator plays a key role in the execution of a `for` loop. Whenever a `yield` statement is reached, the data is bound to the `for` loop variables and control continues in the body of the `for` loop. The iterator's local variables and execution state are automatically saved between calls. Example:

```
# this definition exists in the system module
iterator items*(a: string): char {.inline.} =
  var i = 0
  while i < len(a):
    yield a[i]
    inc(i)

for ch in items("hello world"): # `ch` is an iteration variable
  echo ch
```

Is transformed into:

```nim
var i = 0
while i < len(a):
  var ch = a[i]
  echo ch
  inc(i)
```

(Or into an equivalent form.)

If the iterator yields a tuple, there can be as many iteration variables as there are components in the tuple. The i'th iteration variable's type is the type of the i'th component. In other words, implicit tuple unpacking in a for loop context is performed.

23.1. Implicit items/pairs invocations

If the for loop expression `e` does not denote an iterator and the for loop has exactly 1 variable, the for loop expression is rewritten to `items(e)`; that means an `items` iterator is implicitly invoked:

```nim
for x in [1, 2, 3]: echo x

# is the same as:

for x in items([1, 2, 3]): echo x
```

If the for loop has exactly 2 variables, a `pairs` iterator is implicitly invoked.

```nim
for idx, x in [1, 2, 3]: echo idx, x

# is the same as:

for idx, x in pairs([1, 2, 3]): echo idx, x
```

Symbol lookup of the identifiers `items`/`pairs` is performed after the rewriting step, so that all overloads of `items`/`pairs` are taken into account.

> The rewrite step can happen "too late" when the proper `items` /`pairs` symbols are not in scope:
>
> ```nim
> # module a
> import tables
> ```

```
var tab = initTable[string, int]()

template foreach*() =
  for key, val in tab:
    echo(key, val)


import a

# module b
foreach() ①
```

① This expands to `for key, val in tab` which is transformed into `for key, val in pairs(tab)`, but in module `b`'s scope there is no `pairs` iterator that would match, as that is to be found in the `tables` module that module `b` does not import.

This pitfall will be remedied in later versions of the language.

23.2. First-class iterators

There are 2 kinds of iterators in Nim: *inline* and *closure* iterators. An *inline iterator* is an iterator that's always inlined by the compiler leading to zero overhead for the abstraction, but may result in a heavy increase in code size.

The body of a for loop over an inline iterator is inlined into each `yield` statement appearing in the iterator code, so ideally the code should be refactored to contain a single `yield` when possible to avoid code bloat.

Inline iterators are second class citizens; They can be passed as parameters only to other inlining code facilities like templates, macros, and other inline iterators.

In contrast to that, a *closure iterator* can be passed around more freely:

```
iterator count0(): int {.closure.} =
  yield 0

iterator count2(): int {.closure.} =
  var x = 1
  yield x
  inc x
  yield x
```

```nim
proc invoke(iter: iterator(): int {.closure.}) =
  for x in iter(): echo x

invoke(count0)
invoke(count2)
```

Closure iterators and inline iterators have some restrictions:

1. A closure iterator cannot be executed at compile time.
2. `return` is allowed (but rarely useful) in a closure iterator and not in an inline iterator. It ends the iteration.
3. Neither inline nor closure iterators can be recursive.
4. Neither inline nor closure iterators have the special `result` variable.
5. Closure iterators are not supported by the JS backend.

Iterators that are neither marked `{.closure.}` nor `{.inline.}` explicitly default to being `inline`.

The `iterator` type is always of the calling convention `closure` implicitly; the following example shows how to use iterators to implement a *collaborative tasking* system:

```nim
# simple tasking:
type
  Task = iterator (ticker: int)

iterator a1(ticker: int) {.closure.} =
  echo "a1: A"
  yield
  echo "a1: B"
  yield
  echo "a1: C"
  yield
  echo "a1: D"

iterator a2(ticker: int) {.closure.} =
  echo "a2: A"
  yield
  echo "a2: B"
  yield
  echo "a2: C"
```

```
proc runTasks(t: varargs[Task]) =
  var ticker = 0
  while true:
    let x = t[ticker mod t.len]
    if finished(x): break
    x(ticker)
    inc ticker

runTasks(a1, a2)
```

The builtin `system.finished` can be used to determine if an iterator has finished its operation; no exception is raised on an attempt to invoke an iterator that has already finished its work.

Note that `system.finished` is error prone to use because it only returns `true` one iteration after the iterator has finished:

```
iterator mycount(a, b: int): int {.closure.} =
  var x = a
  while x <= b:
    yield x
    inc x

var c = mycount # instantiate the iterator
while not finished(c):
  echo c(1, 3)

# Produces
# 1
# 2
# 3
# 0
```

Instead this code has to be used:

```
var c = mycount # instantiate the iterator
while true:
  let value = c(1, 3)
  if finished(c): break # and discard 'value'!
  echo value
```

It helps to think that the iterator actually returns a pair `(value, done)` and `finished` is used to access the hidden `done` field.

Closure iterators are *resumable functions* and so one has to provide the arguments to every call. To get around this limitation one can capture parameters of an outer factory proc:

```
proc mycount(a, b: int): iterator (): int =
  result = iterator (): int =
    var x = a
    while x <= b:
      yield x
      inc x

let foo = mycount(1, 4)

for f in foo():
  echo f
```

Chapter 24. User definable conversions

Widening type conversions, such as the conversion from `int8` to `int16`, are performed *implicitly*. Other implicit type conversions can be introduced via a `converter` routine.

24.1. Converters

A converter is like an ordinary proc except that it enhances the “implicitly convertible” type relation (see Section 17.3, “Convertible relation”):

```
# bad style ahead: Nim is not C.
converter toBool(x: int): bool = x != 0

if 4:
  echo "compiles"
```

A converter can also be explicitly invoked for improved readability. Note that implicit converter chaining is not supported: If there is a converter from type A to type B and from type B to type C, the implicit conversion from A to C is not provided.

> Converters should be introduced very carefully. Few use cases are known where it adds clarity to the code rather than obfuscation.

Chapter 25. Exception handling

Nim offers an elaborate, semi-structured mechanism for handling runtime errors called *exception handling*. Exception handling consists of the `try`, `raise` statements, an exception type hierarchy and the `.raises` annotation system.

25.1. Try statement

Example:

```
from std/strutils import parseInt

var
  f: File
if open(f, "numbers.txt"):
  try: ①
    var a = readLine(f) ②
    var b = readLine(f)
    echo "sum: " & $(parseInt(a) + parseInt(b)) ③
  except OverflowDefect: ④
    echo "overflow!"
  except ValueError, IOError: ⑤
    echo "catch multiple exceptions!"
  except: ⑥
    echo "Unknown exception!"
  finally: ⑦
    close(f)
```

① A `try` statement.

② Read the first two lines of a text file.

③ Try to parse them as integers and to sum them.

④ On an `OverflowDefect` output "overflow!".

⑤ More than one exception type can be listed in `except`.

⑥ The empty `except` covers every other exception type. It is comparable to an `else` in an `if` statement.

⑦ Regardless of whether there was any exception raised or not the `finally` section is executed. In this case it ensures we always `close` the file `f`.

The statements after the `try` are executed in sequential order unless an exception `e` is raised. If the exception type of `e` matches any listed in an `except` clause, the corresponding statements are executed. The statements following the `except` clauses are called *exception handlers*.

The empty `except` clause is executed if there is an exception that is not listed otherwise.

If there is a `finally` clause, it is always executed after the exception handlers.

The exception is *consumed* in an exception handler. However, an exception handler may raise another exception. If the exception is not handled, it is propagated through the call stack.

If an exception occurs the "rest" of a routine which is the code that is not within `finally` or `except` clauses is not executed.

25.2. Try expression

Try can also be used as an expression; the type of the `try` branch then needs to fit the types of `except` branches, but the type of the `finally` branch always has to be `void`:

```
from std/strutils import parseInt

let x = try: parseInt("133a")
        except: -1
        finally: echo "hi"
```

To prevent confusing code there is a parsing limitation; if the `try` follows a `(` it has to be written as a one liner:

```
from std/strutils import parseInt

let x = (try: parseInt("133a") except: -1)
```

25.3. Except clauses

Within an `except` clause it is possible to access the current exception using the syntax `ExceptionType as e`:

```
try:
  # ...
except IOError as e:
  echo "I/O error: " & e.msg
```

Alternatively, it is possible to use `system.getCurrentException` to retrieve the exception that was raised:

```
try:
  # ...
except IOError:
  let e = getCurrentException()
```

Note that `getCurrentException` always returns a `ref Exception` type. If a variable of the proper type is needed (in the example above, `IOError`), one must convert it explicitly:

```
try:
  # ...
except IOError:
  let e = (ref IOError)(getCurrentException())
  # "e" is now of the proper type
```

However, this is rarely needed. The most common case is to extract an error message from `e`, and for such situations, it is enough to use `system.getCurrentExceptionMsg`:

```
try:
  # ...
except:
  echo getCurrentExceptionMsg()
```

25.4. Defer statement

A `defer` statement can be used instead of a `try finally` statement if the `try` statement lacks any `except` clauses. In other words, `defer` can be used to ensure resource cleanups (even in case of an error) but not for explicit error handling.

Any statements following the `defer` in the current block will be considered to be in an implicit try block. For example:

```
proc main =
  var f = open("numbers.txt", fmWrite)
  defer: close(f)
  f.write "abc"
  f.write "def"
```

Is rewritten to:

```
proc main =
  var f = open("numbers.txt")
  try:
    f.write "abc"
    f.write "def"
  finally:
    close(f)
```

When `defer` is at the outermost scope of a template/macro, its scope extends to the block where the template is called from:

```
template safeOpenDefer(f, path) =
  var f = open(path, fmWrite)
  defer: close(f)

template safeOpenFinally(f, path, body) =
  var f = open(path, fmWrite)
  try: body # without `defer`, `body` must be specified as parameter
  finally: close(f)

block:
  safeOpenDefer(f, "/tmp/z01.txt")
  f.write "abc"
block:
  safeOpenFinally(f, "/tmp/z01.txt"):
    f.write "abc" # adds a lexical scope
block:
  var f = open("/tmp/z01.txt", fmWrite)
  try:
    f.write "abc" # adds a lexical scope
  finally: close(f)
```

Top-level `defer` statements are not supported since it's unclear what such a statement should refer to.

> `defer` can make the control flow obscure, one should prefer `try finally` or destructors.

25.5. Exception hierarchy

Exception types form a hierarchy via inheritance.

Most of the hierarchy is defined in the `system` module. Every exception ultimately inherits from `system.Exception`.

Exceptions that indicate a runtime error that can be caught inherit from `system.CatchableError` (which is a subtype of `Exception`).

Exceptions that indicate programming bugs inherit from `system.Defect` (which is a subtype of `Exception`) and are strictly speaking not catchable as they can also be mapped to an operation that terminates the whole process.

25.6. Raise statement

A `raise` statement is used to signal that an error occurred:

```
raise newException(IOError, "IO failed")
```

Execution of a `raise` statement starts an *unwinding* process: The control flow continues at a suitable innermost exception handler. That means for `raise e` a handler like `except typeof(e)` or an `except` without a type guard. If no such handler exists, the program is *terminated*.

A `raise` statement without an explicit exception object means that the current exception is *re-raised*. The `ReraiseDefect` exception is raised if there is no exception to re-raise. It follows that the `raise` statement *always* raises an exception.

Apart from a `raise` statement there are other language constructs that can raise an exception:

- Array indexing: It is an error if the index is out of bounds.

- Memory allocation: There is an inherent danger of running *out of memory*.
- Integer arithmetic: An operation like `x + 1` can produce an *overflow* error.
- Type conversion: If `x` is not of the dynamic type `T` then `T(x)` produces an error.

Chapter 26. Effect system

In addition to its *type system* Nim also offers an *effect system*. The goal of the effect system is to prevent even more mistakes at compile-time. The effect system is concerned with the question "what is routine X allowed to do?". In most programming languages a routine is a black box; when you see a call to function `f` without knowing its implementation `f` can perform any arbitrary task: It could perform any kind of IO, it could raise an exception, it could acquire locks that you also need to acquire causing deadlocks and it could run an arbitrary shell script, including a script that it itself generated.

Thus Nim's effect system covers:

- Which types of exceptions a routine can raise, if any.
- Which locks a routine can acquire, if any.
- If a routine accesses any global state.
- Which *custom* effects like `ExecProgram` or `UsesDatabase` a routine can have.

When we talk about *routine* in the context of the effect system, it means `proc`, `func`, `iterator`, `converter`, `method` but not `template` or `macro`.

26.1. Exception tracking

The `raises` pragma can be used to explicitly define which exceptions a routine is allowed to raise. The compiler verifies this:

```
proc p(what: bool) {.raises: [IOError, OSError].} =
  if what: raise newException(IOError, "IO")
  else: raise newException(OSError, "OS")
```

An empty `raises` list (`raises: []`) means that no exception may be raised:

```nim
proc p(): bool {.raises: [].} =
  try:
    unsafeCall()
    result = true
  except:
    result = false
```

A `raises` list can also be attached to a proc type. This affects type compatibility:

```nim
type
  Callback = proc (s: string) {.raises: [IOError].}
var
  c: Callback

proc p(x: string) =
  raise newException(OSError, "OS")

c = p # type error
```

For a routine `p`, inference rules to determine the set of possibly raised exceptions are used; the algorithm operates on `p`'s call graph:

1. Every indirect call via some proc type `T` is assumed to raise `system.Exception` (the base type of the exception hierarchy) and thus any exception unless `T` has an explicit `raises` list. However, an indirect call is ignored if the call is of the form `f(...)` where `f` is a *parameter* of the currently analyzed routine that is marked as `.effectsOf: f`. The call is optimistically assumed to have no effect. Rule 2 compensates for this case.

2. Every expression `e` of some proc type within a call that is passed to parameter marked as `.effectsOf` is assumed to be called indirectly and thus its raises list is added to `p`'s raises list.

3. Every call to a proc `q` which has an unknown body (due to a forward declaration) is assumed to raise `system.Exception` unless `q` has an explicit `raises` list. Procs that are `importc`'ed are assumed to have `.raises: []`, unless explicitly declared otherwise.

4. Every call to a method `m` is assumed to raise `system.Exception` unless `m` has an explicit `raises` list.

5. For every other call, the analysis can determine an exact `raises` list.

6. For determining a `raises` list, the `raise` and `try` statements of `p` are taken into consideration.

Exceptions inheriting from `system.Defect` are not tracked with the `.raises: []` exception tracking mechanism. This is more consistent with the built-in operations. The following code is valid:

```
proc mydiv(a, b): int {.raises: [].} =
  a div b # can raise an DivByZeroDefect
```

And so is:

```
proc mydiv(a, b): int {.raises: [].} =
  if b == 0: raise newException(DivByZeroDefect, "division by zero")
  else: result = a div b
```

The reason for this is that `DivByZeroDefect` inherits from `Defect` and with `--panics:on`:option: Defects become unrecoverable errors.

26.2. EffectsOf annotation

Rules 1-2 of the exception tracking inference rules (see the previous section) ensure the following works:

```
proc weDontRaiseButMaybeTheCallback(callback: proc()) {.
    raises: [], effectsOf: callback.} =
  callback()

proc doRaise() {.raises: [IOError].} =
  raise newException(IOError, "IO")

proc use() {.raises: [].} =
  # doesn't compile! Can raise IOError!
  weDontRaiseButMaybeTheCallback(doRaise)
```

As can be seen from the example, a parameter of type `proc (...)` can be annotated as `.effectsOf`. Such a parameter allows for *effect polymorphism*: The proc `weDontRaiseButMaybeTheCallback` raises the exceptions that `callback` raises.

So in many cases a callback does not cause the compiler to be overly conservative in its effect analysis:

```nim
{.push warningAsError[Effect]: on.}
{.experimental: "strictEffects".}

import algorithm

type
  MyInt = distinct int

var toSort = @[MyInt 1, MyInt 2, MyInt 3]

proc cmpN(a, b: MyInt): int =
  cmp(a.int, b.int)

proc harmless {.raises: [].} =
  toSort.sort cmpN

proc cmpE(a, b: MyInt): int {.raises: [Exception].} =
  cmp(a.int, b.int)

proc harmful {.raises: [].} =
  # does not compile, `sort` can now raise Exception
  toSort.sort cmpE
```

26.3. Tag tracking

Exception tracking is part of Nim's *effect system*. Raising an exception is an *effect*. Other effects can also be defined. A user defined effect is a means to *tag* a routine and to perform checks against this tag:

```nim
type IO = object ## input/output effect
proc readLine(): string {.tags: [IO].} = discard

proc no_IO_please() {.tags: [].} =
  # not allowed:
  let x = readLine()
```

A tag has to be a type name. A `tags` list - like a `raises` list - can also be attached to a proc type. This affects type compatibility.

The inference for tag tracking is analogous to the inference for exception tracking.

26.4. Side effects

The `noSideEffect` pragma is used to mark a routine that can have only side effects through parameters. This means that the routine only changes locations that are reachable from its parameters and the return value only depends on the parameters. If none of its parameters' types contain the type `var`, `ref`, `ptr`, `cstring`, or `proc`, then no locations are modified.

In other words, a routine has no side effects if it does not access a thread-local or global variable and it does not call any routine that has a side effect.

It is a static error to mark a routine to have no side effect if the compiler cannot verify this.

As a special semantic rule, the built-in `system.debugEcho` pretends to be free of side effects so that it can be used for debugging routines marked as `noSideEffect`.

`func` is syntactic sugar for a proc with no side effects:

```
func `+`(x, y: int): int
```

To override the compiler's side effect analysis a `{.noSideEffect.}` `cast` pragma block can be used:

```
func f() =
  {.cast(noSideEffect).}:
    echo "test"
```

Side effects are usually inferred. The inference for side effects is analogous to the inference for exception tracking.

26.5. GC safety effect

We call a proc `p` *GC safe* when it doesn't access any global variable that contains GC'ed memory (`string`, `seq`, `ref` or a closure) either directly or indirectly through a call to a GC unsafe proc.

The GC safety property is usually inferred. The inference for GC safety is analogous to the inference for exception tracking.

The `gcsafe` annotation can be used to mark a proc to be gcsafe, otherwise this property is inferred by the compiler. Note that `noSideEffect` implies `gcsafe`.

Routines that are imported from C are always assumed to be `gcsafe`.

To override the GC safety analysis a `{.cast(gcsafe).}` pragma block can be used:

```
var
  someGlobal: string = "some string here"
  perThread {.threadvar.}: string

proc setPerThread() =
  {.cast(gcsafe).}:
    deepCopy(perThread, someGlobal)
```

Chapter 27. Generics

Generics are Nim's means to parametrize routines or types with *type parameters*. Depending on the context, the brackets are used either to introduce type parameters or to instantiate a generic routine or type.

A generic defines a family of types and routines. If `G[T]` is a generic depending on type `T` then multiple occurrences of `G[C]` where `C` is a concrete type such as `int` or `string` denote the same type. In other words structural type equivalence is used for generics and not name equivalence.

The following example shows how a generic binary tree can be modeled:

```
type
  BinaryTree*[T] = ref object # BinaryTree is a generic type with
                              # generic param `T`
    le, ri: BinaryTree[T]     # left and right subtrees; may be nil
    data: T                   # the data stored in a node

proc newNode*[T](data: T): BinaryTree[T] =
  result = BinaryTree[T](le: nil, ri: nil, data: data)

proc add*[T](root: var BinaryTree[T], n: BinaryTree[T]) =
  if root == nil:
    root = n
  else:
    var it = root
    while it != nil:
      # compare the data items; uses the generic `cmp` proc
      # that works for any type that has a `==` and `<` operator
      var c = cmp(it.data, n.data)
      if c < 0:
        if it.le == nil:
          it.le = n
          return
        it = it.le
      else:
```

```
        if it.ri == nil:
          it.ri = n
          return
        it = it.ri

proc add*[T](root: var BinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: BinaryTree[T]): T =
  # Preorder traversal of a binary tree.
  var stack: seq[BinaryTree[T]] = @[root]
  while stack.len > 0:
    var n = stack.pop()
    while n != nil:
      yield n.data
      add(stack, n.ri)  # push right subtree onto the stack
      n = n.le          # and follow the left pointer

var
  root: BinaryTree[string] # instantiate a BinaryTree with `string`
add(root, newNode("hello")) # instantiates `newNode` and `add`
add(root, "world")          # instantiates the second `add` proc
for str in preorder(root):
  stdout.writeLine(str)
```

In `G[T]` the `T` is called a *generic type parameter* or a *type variable*.

27.1. Is operator

The `is` operator checks for type equivalence. It is therefore very useful for type specialization within generic code:

```
type
  Table[Key, Value] = object
    keys: seq[Key]
    values: seq[Value]
    when not (Key is string): # empty value for strings used for
optimization
      deletedKeys: seq[bool]
```

27.2. Type Classes

A type class is a special pseudo-type that can be used to match against types in the context of overload resolution or the `is` operator. Nim supports the following built-in type classes:

Table 9. Type classes

Type class	Matches
`object`	any object type
`tuple`	any tuple type
`enum`	any enumeration
`proc`	any proc type
`ref`	any `ref` type
`ptr`	any `ptr` type
`var`	any `var` type
`distinct`	any distinct type
`array`	any array type
`set`	any set type
`seq`	any seq type
`auto`	any type

Furthermore, every generic type automatically creates a type class of the same name that will match any instantiation of the generic type.

Type classes can be combined with the `|` operator to form more complex type classes:

```
# create a type class that will match all tuple and object types
type RecordType = (tuple | object)

proc printFields[T: RecordType](rec: T) =
  for key, value in fieldPairs(rec):
    echo key, " = ", value
```

Type constraints on generic parameters can be grouped with `,` and propagation stops with `;`, similarly to parameters for macros and templates:

```
proc fn1[T; U, V: SomeFloat]() = discard # T is unconstrained
template fn2(T; u, v: SomeFloat) = discard # T is unconstrained
```

Nim allows for type classes and regular types to be specified as *type constraints* of the generic type parameter:

```
proc onlyIntOrString[T: int|string](x, y: T) = discard

onlyIntOrString(450, 616) # valid
onlyIntOrString(5.0, 0.0) # type mismatch: float is not an int or a string
onlyIntOrString("xy", 50) # invalid as 'T' cannot be both at the same time
```

27.3. Implicit generics

A type class can be used directly as the parameter's type.

```
# create a type class that will match all tuple and object types
type RecordType = (tuple | object)

proc printFields(rec: RecordType) =
  for key, value in fieldPairs(rec):
    echo key, " = ", value
```

Routines utilizing type classes in such a manner are considered to be *implicitly generic*. They will be instantiated once for each unique combination of param types used within the program.

> Implicit generics can make the code harder to understand as a generic has different symbol binding rules than non-generics. Instead of `proc p(x, y: tuple)` prefer `proc p[T: tuple](x, y: T)`.

By default, during overload resolution, each named type class will bind to exactly one concrete type. We call such type classes *bind once* types. Here is an example taken directly from the system module to illustrate this:

```
proc `==`*(x, y: tuple): bool =
  ## requires `x` and `y` to be of the same tuple type
  ## generic `==` operator for tuples that is lifted from the components
  ## of `x` and `y`.
  result = true
  for a, b in fields(x, y):
    if a != b: result = false
```

Alternatively, the `distinct` type modifier can be applied to the type class to allow each param matching the type class to bind to a different type. Such type classes are called *bind many* types.

Procs written with the implicitly generic style will often need to refer to the type parameters of the matched generic type. They can be easily accessed using the dot syntax:

```
type Matrix[T, Rows, Columns] = object
  ...

proc `[]`(m: Matrix, row, col: int): Matrix.T =
  m.data[col * high(Matrix.Columns) + row]
```

Here are more examples that illustrate implicit generics:

```
proc p(t: Table; k: Table.Key): Table.Value

# is the same as (except that it also adds `Key` and `Value` to the scope):

proc p[Key, Value](t: Table[Key, Value]; k: Key): Value


proc p(a: Table, b: Table)

# is the same as (except that it also adds `Key` and `Value` to the scope):

proc p[Key, Value](a, b: Table[Key, Value])


proc p(a: Table, b: distinct Table)

# is the same as (except that it also adds `Key` and `Value` to the scope):

proc p[Key, Value, KeyB, ValueB](a: Table[Key, Value], b: Table[KeyB,
ValueB])
```

`typedesc` used as a parameter type also introduces an implicit generic. `typedesc` has its own set of rules:

```
proc p(a: typedesc)

# is the same as (except that it also adds `T` to the scope):

proc p[T](a: typedesc[T])
```

`typedesc` is a "bind many" type class:

```
proc p(a, b: typedesc)

# is roughly the same as:

proc p[T, T2](a: typedesc[T], b: typedesc[T2])
```

A parameter of type `typedesc` is itself usable as a type. If it is used as a type, it's the underlying type. (In other words, one level of "typedesc"-ness is stripped off):

```
proc p(a: typedesc; b: a) = discard

# is roughly the same as:
proc p[T](a: typedesc[T]; b: T) = discard

# hence this is a valid call:
p(int, 4)
# as parameter 'a' requires a type, but 'b' requires a value.
```

27.4. Generic inference restrictions

The types `var T` and `typedesc[T]` cannot be inferred in a generic instantiation. The following is not allowed:

```
proc g[T](f: proc(x: T); x: T) =
  f(x)

proc c(y: int) = echo y
proc v(y: var int) =
  y += 100
var i: int

# allowed: infers 'T' to be of type 'int'
g(c, 42)

# not valid: 'T' is not inferred to be of type 'var int'
g(v, i)

# also not allowed: explicit instantiation via 'var int'
g[var int](v, i)
```

27.5. Symbol lookup in generics

The symbol binding rules in generics are slightly subtle: There are "open" and "closed" symbols.

27.5.1. Open and Closed symbols

A "closed" symbol cannot be re-bound in the instantiation context, an "open" symbol can. Per default, symbols that are overloaded in the scope of the generic definition are open and all other symbols are closed.

Open symbols are looked up in two different contexts: Both the context at definition and the context at instantiation are considered:

```
type
  Index = distinct int

proc `==` (a, b: Index): bool {.borrow.}

var a = (0, 0.Index)
var b = (0, 0.Index)

echo a == b # works!
```

In the example, the generic `==` for tuples (as defined in the system module) uses the `==` operators of the tuple's components. However, the `==` for the `Index` type is defined *after* the `==` for tuples; yet the example compiles as the instantiation takes the currently defined symbols into account too.

27.6. Mixin statement

A symbol can be forced to be open by a `mixin` declaration:

```
proc create*[T](): ref T =
  # there is no overloaded 'init' here, so we need to state that it's an
  # open symbol explicitly:
  mixin init
  new result
  init result
```

`mixin` statements are only available in templates and generics.

27.7. Bind statement

The `bind` statement is the counterpart to the `mixin` statement. It can be used to explicitly declare identifiers that should be bound early (i.e. the identifiers should be looked up in the scope of the template/generic definition):

```
# Module A
var
  lastId = 0

template genId*: untyped =
  bind lastId
  inc(lastId)
  lastId
```

```
# Module B
import A

echo genId()
```

But a `bind` is rarely useful because symbol binding from the definition scope is the default.

`bind` statements are only available in templates and generics.

27.8. Delegating bind statements

The following example outlines a problem that can arise when generic instantiations cross multiple different modules:

```
# module A
proc genericA*[T](x: T) =
  mixin init
  init(x)
```

```
# module C
type O = object
proc init*(x: var O) = discard
```

```
import C
```

```nim
# module B
proc genericB*[T](x: T) =
  # Without the `bind init` statement C's init proc is
  # not available when `genericB` is instantiated:
  bind init
  genericA(x)
```

```nim
# module main
import B, C

genericB 0()
```

Module `B` has an `init` proc from module `C` in its scope that is not taken into account when `genericB` is instantiated which leads to the instantiation of `genericA`. The solution is to *forward* these symbols by a `bind` statement inside `genericB`.

27.9. Templates

A template is a simple form of a macro: It is a simple substitution mechanism that operates on Nim's abstract syntax trees.

The syntax to *invoke* a template is the same as calling any other kind of routine.

Example:

```nim
template `!=` (a, b: untyped): untyped =
  # this definition exists in the system module
  not (a == b)

assert(5 != 6) # transformed into: assert(not (5 == 6))
```

The `!=`, `>`, `>=`, `in`, `notin`, `isnot` operators are in fact templates:

`a > b` is transformed into `b < a`.
`a in b` is transformed into `contains(b, a)`.
`notin` and `isnot` have the obvious meanings.

The "types" of template parameters can be the symbols `untyped`, `typed` or `typedesc`. These are "meta types", they can only be used in certain contexts. Regular types can be used too; this implies that `typed` expressions are expected.

27.10. Typed vs untyped parameters

For an `untyped` parameter symbol lookups and type resolution is not performed before the expression is passed to the template. This means that semantic checking of the argument that is passed to an `untyped` parameter is *lazily* done. The implications of this mechanism are important to understand. For example, it means that *undeclared* identifiers can be passed to the template:

```
template declareInt(x: untyped) =
  var x: int

declareInt(x) # valid
x = 3
```

```
template declareInt(x: typed) =
  var x: int

declareInt(x) # invalid, because x has not been declared and so it has no
type
```

A template where every parameter is `untyped` is called an *immediate* template. For historical reasons, templates can be explicitly annotated with an `immediate` pragma and then these templates do not take part in overloading resolution and the parameters' types are *ignored* by the compiler. Explicit immediate templates are deprecated.

27.11. Passing a code block to a template

One can pass a block of statements as the last argument to a template following the special `:` syntax:

```
template withFile(f, fn, mode, actions: untyped): untyped =
  var f: File
  if open(f, fn, mode):
    try:
      actions
    finally:
      close(f)
  else:
```

```
    quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):  # special colon
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

In the example, the two `writeLine` statements are bound to the `actions` parameter.

Usually, to pass a block of code to a template, the parameter that accepts the block needs to be of type `untyped`. Because symbol lookups are then delayed until template instantiation time:

```
template t(body: typed) =
  proc p = echo "p"
  block:
    body

t:
  p()  # fails with 'undeclared identifier: p'
```

The above code fails with the error message that `p` is not declared. The reason for this is that the `p()` body is type-checked before getting passed to the `body` parameter and type checking implies symbol lookups. The same code works with `untyped` as the passed body is not required to be type-checked:

```
template t(body: untyped) =
  proc p = echo "p"
  block:
    body

t:
  p()  # compiles
```

`untyped` parameters cause problems with overloading:

```
template t(body: untyped) =  ①
  proc p = echo "p"
  body

proc t(a: int) = discard ②

t:
  p()  ③
```

① A template that takes an `untyped` code snippet.

② A proc that overloads the symbol `t`.

③ The code snippet `p()` needs to be checked for semantics before it can be decided which overloaded routine `t` to invoke. This conflicts with `t's requirements.

In theory the compiler could resolve the call to `t` without problem. In practice, at the time of this writing, the compiler is not able to do that. In the long run we hope to phase out `untyped` parameters; what they enable can be accomplished by other means that compose better.

27.12. Varargs of untyped

In addition to the `untyped` meta-type that delays type checking, there is also `varargs[untyped]` so that not even the number of parameters is fixed:

```
template hideIdentifiers(x: varargs[untyped]) = discard

hideIdentifiers(undeclared1, undeclared2)
```

However, since a template cannot iterate over varargs, this feature is generally much more useful for macros.

27.13. Symbol binding in templates

A template is a *hygienic* macro and so opens a new scope. The distinction between *open* and *closed* symbols applies to templates as it does apply to generics:

```
# Module A
var
  lastId = 0

template genId*: untyped =
  inc(lastId)
  lastId
```

```
# Module B
import A

echo genId() # Works as 'lastId' is bound in 'genId's defining scope
```

As in generics, symbol binding can be influenced via `mixin` or `bind` statements.

27.14. Identifier construction

In templates, identifiers can be constructed with the backticks notation:

```
template typedef(name: untyped, typ: typedesc) =
  type
    `T name`* {.inject.} = typ
    `P name`* {.inject.} = ref `T name`

typedef(myint, int)
var x: PMyInt
```

In the example, `name` is instantiated with `myint`, so `T name` becomes `Tmyint`.

27.15. Template parameter lookup rules

A parameter `p` in a template is even substituted in the expression `x.p`. Thus, template arguments can be used as field names and a global symbol can be shadowed by the same argument name even when fully qualified:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levA'
```

But the global symbol can be captured by a `bind` statement:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  bind m.abclev
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levB'
```

> Instead of relying on this subtle rule, name your parameters so that they do not conflict with other names.

27.16. Hygiene in templates

Per default, templates are *hygienic*: Local identifiers declared in a template cannot be accessed in the instantiation context.

```
template `||`(a, b: untyped): untyped =
  let aa = a ①
  if aa.len > 0: aa else: b

var a = ""
var b = "abc"
echo a || b || "def" ②
```

① The variable `aa` is introduced so that the expression `a` is evaluated only once.

② The output of the program is `"abc"`.

Every expansion causes a “fresh” set of local variables to be created. These local variables do not interfere with each other. A template is thus very similar to an `.inline` proc or func.

27.16.1. Inject and gensym

Whether a symbol that is declared in a template is exposed to the instantiation scope is controlled by the `inject` and `gensym` pragmas: `gensym`'ed symbols are not exposed but `inject`'ed symbols are.

The default for symbols of entity `type`, `var`, `let` and `const` is `gensym` and for a routine it is `inject`. However, if the name of the entity is passed as a template parameter, it is an `inject`'ed symbol:

```
template withFile(f, fn, mode: untyped, actions: untyped): untyped =
  block:
    var f: File  # since 'f' is a template param, it's injected implicitly
    ...

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

The `inject` and `gensym` pragmas are second class annotations; they have no semantics outside of a template definition and cannot be abstracted over:

```
{.pragma myInject: inject.}

template t() =
  var x {.myInject.}: int # does NOT work
```

To get rid of hygiene in templates, one can use the `dirty` pragma for a template. `inject` and `gensym` have no effect in `dirty` templates.

`gensym`'ed symbols cannot be used as `field` in the `x.field` syntax. Nor can they be used in the `ObjectConstruction(field: value)` and `namedParameterCall(field = value)` syntactic constructs.

The reason for this is that code like

```
type
  T = object
    f: int

template tmp(x: T) =
  let f = 34
  echo x.f, T(f: 4)
```

should work as expected.

However, this means that the method call syntax is not available for `gensym`'ed symbols:

```
template tmp(x) =
  type
    T {.gensym.} = int

  echo x.T # invalid: instead use:  'echo T(x)'.

tmp(12)
```

27.17. Method call syntax limitations

The expression `x` in `x.f` needs to be checked for semantics (that means symbol lookup and type checking have to be performed) before it can be decided that it needs to be rewritten to `f(x)`. Therefore the dot syntax has some limitations when it is used to invoke templates/macros:

```
template declareVar(name: untyped) =
  const name {.inject.} = 45

# Doesn't compile:
unknownIdentifier.declareVar
```

Chapter 28. Macros

A `macro` is similar to an advanced but low level template. Macros can be used to implement *domain specific languages*. A macro is a routine that operates directly on the abstract syntax tree (AST) of the Nim programming language. Unfortunately, the AST is implementation defined.

To write macros, one needs to know how the Nim concrete syntax is converted to an abstract syntax tree. A macro receives ASTs as its input parameters and returns a new AST. The compiler then analyzes the result AST for errors - a macro cannot be used to circumvent error checking. Let `r` be the result of a macro invocation `m(args)`. `r` is inserted into the position of the invocation `m(args)`. `r` can contain further macro invocations, these are processed after the expansion of `m`. The process iterates until no more macro expansions can be performed or until some implementation defined iteration limit is reached. Reaching the limit is a static error.

28.1. Macros API

The tree transformations that a macro can perform are enabled by the `macros` standard module. The API is based on a single `NimNode` type that represents a single node or a complete tree. Every `NimNode` has a node `kind` determining if the node is an `if` statement, a routine call, etc. Some \`NimNode\`s can also have children.

> The `NodeKind` is a `enum` with fields like `nnkStmtList` or `nnkIfStmt`. The `nnk` prefix exists for historical reasons.

The following example implements a `debug` command that accepts a variable number of arguments and writes them with their name and value:

```nim
import std/macros

macro debug(args: varargs[untyped]): untyped = ①
  result = newNimNode(nnkStmtList, args) ②
  for a in args: ③
    result.add newCall("write", ident"stdout", toStrLit(a)) ④
    result.add newCall("write", ident"stdout", newLit(": ")) ⑤
    result.add newCall("writeLine", ident"stdout", a) ⑥

var ⑦
  a: array[0..10, int]
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x) ⑧
```

① Inside the `debug` macro every parameter (that is not a `static` parameter) is of type `NimNode` and *not* of the type that is written down in the signature. That means within the body of `debug` `args` is of type `NimNode` and not of type `varargs[untyped]`.

② `debug` returns a list of statements (`nnkStmtList`).

③ For every passed argument do:

④ Add a call to the statement list that writes the expression; `toStrLit` converts an AST to its string representation.

⑤ Also add a call to the statement list that writes the string literal `": "`.

⑥ Add a call to the statement list that writes the expressions value of the current argument `a`.

⑦ Example data that is passed to the `debug` macro.

⑧ Call to the `debug` macro.

The macro call expands to:

```nim
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
```

```
write(stdout, ": ")
writeLine(stdout, x)
```

Arguments that are passed to a `varargs` parameter are wrapped in an array constructor expression. This is why `debug` iterates over all of `args`'s children.

28.2. BindSym

The above `debug` macro relies on the fact that `write`, `writeLine` and `stdout` are declared in the system module and are thus visible in the instantiating context.

Via the `bindSym` builtin there is a way to use bound identifiers (a.k.a. *symbols*) instead of using unbound identifiers:

```
import std/macros

macro debug(n: varargs[typed]): untyped =
  result = newNimNode(nnkStmtList, n)
  for x in n:
    # we can bind symbols in scope via 'bindSym':
    result.add newCall(bindSym"write", bindSym"stdout", toStrLit(x))
    result.add newCall(bindSym"write", bindSym"stdout", newStrLitNode": ")
    result.add newCall(bindSym"writeLine", bindSym"stdout", x)

var
  a: array[0..10, int]
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x)
```

The macro call expands to:

```
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
```

```
writeLine(stdout, x)
```

However, the symbols write, writeLine and stdout are already bound and are not looked up again. As the example shows, bindSym does work with overloaded symbols implicitly.

> The distinction between bindSym"name" and ident"name" is easiest to understand when one takes scope into consideration. This is valid code:
>
> ```
> import std/macros
>
> macro m(a: string): untyped =
>   proc helper(a: string) = echo a
>
>   result = newCall(bindSym"helper", a)
>
> m "abc"
> ```
>
> And this is invalid code:
>
> ```
> import std/macros
>
> macro m(a: string): untyped =
>   proc helper(a: string) = echo a
>
>   result = newCall(ident"helper", a)
>
> m "abc"
> ```
>
> Note how in both cases helper is local to the macro m and invisible outside of `m's body.

28.3. For loop macros

A macro that only takes a single expression of the special type system.ForLoopStmt can rewrite the entire for loop:

```
import std/macros

macro example(loop: ForLoopStmt) =
  result = newTree(nnkForStmt)    # Create a new For loop.
  result.add loop[^3]             # This is "item".
```

```
  result.add loop[^2][^1]         # This is "[1, 2, 3]".
  result.add newCall(bindSym"echo", loop[0])

for item in example([1, 2, 3]): discard
```

Expands to:

```
for item in items([1, 2, 3]):
  echo item
```

Another example:

```
import std/macros

macro enumerate(x: ForLoopStmt): untyped =
  expectKind x, nnkForStmt
  # check if the starting count is specified:
  var countStart = if x[^2].len == 2: newLit(0) else: x[^2][1]
  result = newStmtList()
  # we strip off the first for loop variable
  # and use it as an integer counter:
  result.add newVarStmt(x[0], countStart)
  var body = x[^1]
  if body.kind != nnkStmtList:
    body = newTree(nnkStmtList, body)
  body.add newCall(bindSym"inc", x[0])
  var newFor = newTree(nnkForStmt)
  for i in 1..x.len-3:
    newFor.add x[i]
  # transform enumerate(X) to 'X':
  newFor.add x[^2][^1]
  newFor.add body
  result.add newFor
  # wrap the whole macro in a block to create a new scope:
  result = newTree(nnkBlockExpr, newEmptyNode(), result)

for a, b in enumerate(items([1, 2, 3])):
  echo a, " ", b

# without wrapping the macro in a block, we'd need to choose different
# names for `a` and `b` here to avoid redefinition errors
for a, b in enumerate(10, [1, 2, 3, 5]):
  echo a, " ", b
```

For readability and maintainability it is best to use the *least powerful* programming construct that still accomplishes the

> goal. So the "check list" is:
>
> (1) Use an ordinary proc/iterator, if possible. (2) Else: Use a generic proc/iterator, if possible. (3) Else: Use a template, if possible. (4) Else: Use a macro.

The Part III: Mastering Macros contains many more examples of how to write and use macros.

Chapter 29. Lifetime-tracking hooks

The memory management for Nim's standard `string` and `seq` types as well as other standard collections is performed via so-called "Lifetime-tracking hooks", which are particular *type bound operators*.

There are different hooks for each (generic or concrete) object type `T` (`T` can also be a `distinct` type) that are called implicitly, in strategic places.

> The word “hook” here does not imply any kind of dynamic binding or runtime indirections, the implicit calls are statically bound and potentially inlined.

29.1. `=destroy` hook

A `=destroy` hook frees the object's associated memory and releases other associated resources. Variables are destroyed via this hook when they go out of scope or when the routine they were declared in is about to return.

The prototype of this hook for a type `T` needs to be:

```
proc `=destroy`(x: T)
```

The general pattern in `=destroy` looks like:

```
proc `=destroy`(x: T) =
  # first check if 'x' was moved to somewhere else:
  if x.field != nil:
    freeResource(x.field)
```

29.2. `=wasMoved` hook

A `=wasMoved` hook sets the object to a state that signifies to the destructor there is nothing to destroy.

The prototype of this hook for a type `T` needs to be:

```
proc `=wasMoved`(x: var T)
```

Usually some pointer field inside the object is set to `nil`:

```
proc `=wasMoved`(x: var T) =
  x.field = nil
```

29.3. `=sink` hook

A `=sink` hook moves an object around, the resources are stolen from the source and passed to the destination. It is ensured that the source's destructor does not free the resources afterward by setting the object to its “was moved” value via the `=wasMoved` hook.

When not provided a combination of `=destroy` and `copyMem` is used instead. This is efficient hence users rarely need to implement their own `=sink` operator, it is enough to provide `=destroy` and `=copy`, the compiler takes care of the rest.

The prototype of this hook for a type `T` needs to be:

```
proc `=sink`(dest: var T; source: T)
```

The general pattern in `=sink` looks like:

```
proc `=sink`(dest: var T; source: T) =
  `=destroy`(dest)
  `=wasMoved`(dest)
  dest.field = source.field
```

> `=sink` does not need to check for self-assignments. How self-assignments are handled is explained later.

29.4. `=copy` hook

The ordinary assignment in Nim conceptually copies the values. The `=copy` hook is called for assignments that couldn't be transformed into `=sink` operations.

The prototype of this hook for a type `T` needs to be:

```
proc `=copy`(dest: var T; source: T)
```

The general pattern in `=copy` looks like:

```
proc `=copy`(dest: var T; source: T) =
  # protect against self-assignments:
  if dest.field != source.field:
    `=destroy`(dest)
    `=wasMoved`(dest)
    dest.field = duplicateResource(source.field)
```

The `=copy` proc can be marked with the `{.error.}` pragma. Then any assignment that otherwise would lead to a copy is prevented at compile-time. This looks like:

```
proc `=copy`(dest: var T; source: T) {.error.}
```

Notice that there is no `=` before the `{.error.}` pragma.

29.5. `=dup` hook

A `=dup` hook duplicates an object. `=dup(x)` can be regarded as an optimization replacing a `wasMoved(dest); =copy(dest, x)` operation.

The prototype of this hook for a type `T` needs to be:

```
proc `=dup`(x: T): T
```

The general pattern in implementing `=dup` looks like:

```nim
type
  Ref[T] = object
    data: ptr T
    rc: ptr int

proc `=dup`[T](x: Ref[T]): Ref[T] =
  result = x
  if x.rc != nil:
    inc x.rc[]
```

29.6. `=trace` hook

A custom *container* type can support Nim's cycle collector `--mm:orc` via the `=trace` hook. If the container does not implement `=trace`, cyclic data structures which are constructed with the help of the container might leak memory or resources, but memory safety is not compromised.

The prototype of this hook for a type `T` needs to be:

```nim
proc `=trace`(dest: var T; env: pointer)
```

`env` is used by ORC to keep track of its internal state, it should be passed around to calls of the built-in `=trace` operation.

The general pattern in using `=destroy` with `=trace` looks like:

```nim
type Test[T] = object
  size: Natural
  arr: ptr UncheckedArray[T] # raw pointer field

proc makeTest[T](size: Natural): Test[T] =
  Test[T](size: size,
          arr: cast[ptr UncheckedArray[T]](alloc0(sizeof(T) * size)))

proc `=destroy`[T](dest: var Test[T]) =
  if dest.arr != nil:
    for i in 0 ..< dest.size: dest.arr[i].`=destroy`
    dest.arr.dealloc

proc `=trace`[T](dest: var Test[T]; env: pointer) =
  if dest.arr != nil: # trace the ``T``'s which may be cyclic
    for i in 0 ..< dest.size: dest.arr[i].`=trace` env

# following may be other custom "`hooks`" as required...
```

> The `=trace` hooks (which are only used by `--mm:orc`) are currently more experimental and less refined than the other hooks.

29.7. Move semantics

A "move" can be regarded as an optimized copy operation. If the source of the copy operation is not used afterward, the copy can be replaced by a move. The notation `lastReadOf(x)` is used to describe that `x` is not used afterward. This property is computed by a static control flow analysis but can also be enforced by using `system.move` explicitly.

One can query if the analysis is able to perform a move with `system.ensureMove`. `move` enforces a move operation and calls `=wasMoved` whereas `ensureMove` is an annotation that implies no runtime operation. An `ensureMove` annotation leads to a static error if the compiler cannot prove that a move would be safe.

For example:

```
proc main(normalParam: string; sinkParam: sink string) =
  var x = "abc"
  # valid:
  let valid = ensureMove x
  # invalid:
  let invalid = ensureMove normalParam
  # valid:
  let alsoValid = ensureMove sinkParam
```

29.8. Swap

The need to check for self-assignments and also the need to destroy previous objects inside `=copy` and `=sink` is a strong indicator to treat `system.swap` as a builtin primitive of its own that simply swaps every field in the involved objects via `copyMem` or a comparable mechanism. In other words, `swap(a, b)` is *not* implemented as `let tmp = move(b); b = move(a); a = move(tmp)`.

This has further consequences:

- Objects that contain pointers that point to the same object are not supported by Nim's model. Otherwise swapped objects would end up in an inconsistent state.

- Sequences can use `realloc` in the implementation.

29.9. Sink parameters

To move a variable into a collection usually `sink` parameters are involved. A location that is passed to a `sink` parameter should not be used afterward. This is ensured by a static analysis over a control flow graph. If it cannot be proven to be the last usage of the location, a copy is done instead and this copy is then passed to the sink parameter.

A sink parameter *may* be consumed once in the proc's body but doesn't have to be consumed at all. The reason for this is that signatures like `proc put(t: var Table; k: sink Key, v: sink Value)` should be possible without any further overloads and `put` might not take ownership of `k` if `k` already exists in the table. Sink parameters enable an affine type system, not a linear type system.

The employed static analysis is limited and only concerned with local variables; however, object and tuple fields are treated as separate entities:

```
proc consume(x: sink Obj) = discard "no implementation"

proc main =
  let tup = (Obj(), Obj())
  consume tup[0]
  # ok, only tup[0] was consumed, tup[1] is still alive:
  echo tup[1]
```

Sometimes it is required to explicitly `move` a value into its final position:

```
proc main =
  var dest, src: array[10, string]
  # ...
  for i in 0..high(dest): dest[i] = move(src[i])
```

An implementation is allowed, but not required to implement even more move optimizations (and the current implementation does not).

29.10. Rewrite rules

There are two different allowed implementation strategies:

1. The produced `finally` section can be a single section that is wrapped around the complete routine body.
2. The produced `finally` section is wrapped around the enclosing scope.

The current implementation follows strategy (2). This means that resources are destroyed at the scope exit.

Table 10. Rewrite rules

Pattern	Rewritten as	Rule name
`var x: T; stmts`	`var x: T; try stmts finally: ` `` `=destroy`(x) ``	destroy-var
`g(f(...))`	`g(let tmp; bitwiseCopy tmp, f(...); tmp) finally: ` `` `=destroy`(tmp) ``	nested-function-call
`x = f(...)`	`` `=sink`(x, f(...)) ``	function-sink
`x = lastReadOf z`	`` `=sink`(x, z) `=wasMoved`(z) ``	move-optimization
`v = v`	`discard "nop"`	self-assignment-removal
`x = y`	`` `=copy`(x, y) ``	copy
`f_sink(g())`	`f_sink(g())`	call-to-sink
`f_sink(notLastReadOf y)`	`` (let tmp; `=dup`(y); f_sink(tmp)) ``	copy-to-sink
`f_sink(lastReadOf y)`	`` f_sink(y) `=wasMoved`(y) ``	move-to-sink

29.11. Object and array construction

Object and array construction is treated as a function call where the function has `sink` parameters.

29.12. Destructor removal

`=wasMoved(x)` followed by a `=destroy(x)` operation cancel each other out. An implementation is encouraged to exploit this in order to improve efficiency and code sizes. The current implementation does perform this optimization.

29.13. Self assignments

`=sink` in combination with `=wasMoved` can handle self-assignments but it's subtle.

The simple case of `x = x` cannot be turned into `=sink(x, x); =wasMoved(x)` because that would lose `x`'s value. The solution is that simple self-assignments that consist of

- Symbols: `x = x`
- Field access: `x.f = x.f`
- Array, sequence or string access with indices known at compile-time: `x[0] = x[0]`

are transformed into an empty statement that does nothing. The compiler is free to optimize further cases.

The complex case looks like a variant of `x = f(x)`, we consider `x = select(rand() < 0.5, x, y)` here:

```nim
proc select(cond: bool; a, b: sink string): string =
  if cond:
    result = a # moves a into result
  else:
    result = b # moves b into result

proc main =
  var x = "abc"
  var y = "xyz"
  # possible self-assignment:
  x = select(true, x, y)
```

Is transformed into:

```nim
proc select(cond: bool; a, b: sink string): string =
  try:
    if cond:
      `=sink`(result, a)
      `=wasMoved`(a)
    else:
      `=sink`(result, b)
      `=wasMoved`(b)
  finally:
    `=destroy`(b)
    `=destroy`(a)

proc main =
  var
    x: string
    y: string
  try:
    `=sink`(x, "abc")
    `=sink`(y, "xyz")
    `=sink`(x, select(true,
      let blitTmp = x
      `=wasMoved`(x)
      blitTmp,
      let blitTmp = y
      `=wasMoved`(y)
      blitTmp))
    echo [x]
  finally:
    `=destroy`(y)
    `=destroy`(x)
```

As can be manually verified, this transformation is correct for self-assignments.

29.14. Lent type

`proc p(x: sink T)` means that the proc `p` takes ownership of `x`. To eliminate even more creation/copy ←→ destruction pairs, a proc's return type can be annotated as `lent T`. This is useful for "getter" accessors that seek to allow an immutable view into a container.

The `sink` and `lent` annotations allow us to remove most (if not all) superfluous copies and destructions.

`lent T` is like `var T` a hidden pointer. It is proven by the compiler that the pointer does not outlive its origin. No destructor call is injected for expressions of type `lent T` or of type `var T`.

```
type
  Tree = object
    kids: seq[Tree]

proc construct(kids: sink seq[Tree]): Tree =
  result = Tree(kids: kids)
  # converted into:
  `=sink`(result.kids, kids); `=wasMoved`(kids)
  `=destroy`(kids)

proc `[]`*(x: Tree; i: int): lent Tree =
  result = x.kids[i]
  # borrows from 'x', this is transformed into:
  result = addr x.kids[i]
  # This means 'lent' is like 'var T' a hidden pointer.
  # Unlike 'var' this hidden pointer cannot be used to mutate the object.

iterator children*(t: Tree): lent Tree =
  for x in t.kids: yield x

proc main =
  # everything turned into moves:
  let t = construct(@[construct(@[]), construct(@[])])
  echo t[0] # accessor does not copy the element!
```

29.15. The .cursor annotation

Nim's `ref` type is implemented via the same runtime "hooks" and thus via reference counting. This means that cyclic structures cannot be freed immediately (but eventually they are freed as a cycle collector also exists).

With the `.cursor` annotation one can break up cycles declaratively:

```
type
  Node = ref object
    left: Node # owning ref
    right {.cursor.}: Node # non-owning ref
```

But please notice that this is not C++'s `weak_ptr`, it means the `right` field is not involved in the reference counting, it is a raw pointer without runtime checks.

Automatic reference counting also has the disadvantage that it introduces overhead when iterating over linked structures. The `.cursor` annotation can also be used to avoid this overhead:

```
var it {.cursor.} = listRoot
while it != nil:
  use(it)
  it = it.next
```

In fact, `.cursor` more generally prevents object construction/destruction pairs and so can also be useful in other contexts. The alternative solution would be to use raw pointers (`ptr`) instead which is more cumbersome and also more dangerous for Nim's evolution: Later on, a compiler can try to prove `.cursor` annotations to be safe, but for `ptr` a compiler cannot report possible problems.

29.16. Cursor inference / copy elision

The current implementation also performs `.cursor` inference. Cursor inference is a form of copy elision.

To see how and when we can do that, think about this question: In `dest = src` when do we really have to *materialize* the full copy? - Only if `dest` or `src` are mutated afterward. If `dest` is a local variable that is simple to analyze. And if `src` is a location derived from a formal parameter, we also know it is not mutated! In other words, we do a compile-time copy-on-write analysis.

This means that “borrowed” views can be written naturally and without explicit pointer indirections:

```
proc main(tab: Table[string, string]) =
  let v = tab["key"] # inferred as .cursor because 'tab' is not mutated.
  # no copy into 'v', no destruction of 'v'.
  use(v)
  useItAgain(v)
```

29.17. Hook lifting

The hooks of a tuple type `(A, B, ...)` are generated by lifting the hooks of the involved types `A`, `B`, ... to the tuple type. In other words, a copy `x = y` is implemented as `x[0] = y[0]; x[1] = y[1]; ...`, likewise for `=sink` and `=destroy`.

Other value-based compound types like `object` and `array` are handled correspondingly. For `object` however, the generated hooks can be overridden. This can also be important to use an alternative traversal of the involved data structure that is more efficient or in order to avoid deep recursions.

29.18. Hook generation

The ability to override a hook leads to a phase ordering problem:

```
type
  Foo[T] = object

proc main =
  var f: Foo[int]
  # error: destructor for 'f' called here before
  # it was seen in this module.

proc `=destroy`[T](f: var Foo[T]) =
  discard
```

The solution is to define `proc `=destroy`[T](f: var Foo[T])` before it is used. The compiler generates implicit hooks for all types in *strategic places* so that an explicitly provided hook that comes too "late" can be detected reliably. These *strategic places* are derived from the following rewrite rules:

- In the construct `let/var x = ...` (var/let binding) hooks are generated for `typeof(x)`.

- In `x = ···` (assignment) hooks are generated for `typeof(x)`.
- In `f(···)` (function call) hooks are generated for `typeof(f(···))`.
- For every sink parameter `x: sink T` the hooks are generated for `typeof(x)`.

29.19. `nodestroy` pragma

The experimental `nodestroy` pragma inhibits hook injections. This can be used to specialize the object traversal in order to avoid deep recursions:

```
type Node = ref object
  x, y: int32
  left, right: Node

type Tree = object
  root: Node

proc `=destroy`(t: var Tree) {.nodestroy.} =
  # use an explicit stack so that we do not get stack overflows:
  var s: seq[Node] = @[t.root]
  while s.len > 0:
    let x = s.pop
    if x.left != nil: s.add(x.left)
    if x.right != nil: s.add(x.right)
    # free the memory explicitly:
    dispose(x)
  # notice how even the destructor for 's' is not called implicitly
  # anymore thanks to .nodestroy, so we have to call it on our own:
  `=destroy`(s)
```

As can be seen from the example, this solution is hardly sufficient and should eventually be replaced by a better solution.

29.20. Copy on write

String literals are implemented as "copy on write". When assigning a string literal to a variable, a copy of the literal won't be created. Instead the variable simply points to the literal. The literal is shared between different variables which are pointing to it. The copy operation is deferred until the first write.

> The abstraction fails for `addr x` because whether the address is

> going to be used for mutations is unknown.

`prepareMutation` should be called before the address operation:

```
var x = "abc"
var y = x

prepareMutation(y)
moveMem(addr y[0], addr x[0], 3)
assert y == "abc"
```

29.21. Practice

In practice the hooks for memory and resource management should be used rarely; Nim usually does the right thing and overriding the default behavior can be both error-prone and result in non-intuitive behavior. However, the hooks are very useful for interoperability with C and C++.

Here is a prototypical example of a C library that we want to wrap:

```
#include <stdlib.h>
typedef struct {
  double x;
  char* s;
} Obj;

Obj* createObj(const char* s) {
  Obj* result = (Obj*) malloc(sizeof(Obj));
  result->x = 40.0;
  result->s = malloc(100);
  strcpy(result->s, s);
  return result;
}

void destroyObj(Obj* obj) {
  free(obj->s);
  free(obj);
}

void useObj(Obj* obj) {}

double getX(Obj* obj) { return obj->x; }
```

The wrapper should automate the memory management so that `destroyObj` does not have to be called explicitly:

```nim
type
  Obj = object ①

proc createObj(s: cstring): ptr Obj {.importc.} ②
proc destroyObj(obj: ptr Obj) {.importc.}
proc useObj(obj: ptr Obj) {.importc.}
proc getX(obj: ptr Obj): float {.importc.}

type
  Wrapper = object ③
    obj: ptr Obj

proc `=destroy`(dest: var Wrapper) =
  if dest.obj != nil: destroyObj(dest.obj) ④

proc `=copy`(dest: var Wrapper; source: Wrapper) {.error.} ⑤

proc create(s: string): Wrapper = Wrapper(obj: createObj(cstring(s))) ⑥

proc use(w: Wrapper) = useObj(w.obj)
proc getX(w: Wrapper): float = getX(w.obj) ⑦

proc useWrapper = ⑧
  let w = @[create("abc"), create("def")] ⑨
  use w[0]
  echo w[1].getX ⑩

useWrapper()
```

① The C struct is mapped directly to a Nim object.

② The procs `createObj`, `destroyObj`, `useObj`, and `getX` are imported from C.

③ The `Wrapper` object encapsulates a C `ptr Obj`.

④ The wrapper has a custom destructor that calls `destroyObj`. The destructor is called automatically if the lifetime of an object of type `Wrapper` ends.

⑤ Because the C library offers no way to copy an `Obj` the wrapper does not offer it either. Any attempt to copy a wrapper will be rejected by the compiler.

⑥ `create` is used to wrap `createObj`. Its input parameter was changed from `cstring` to `string` in order to improve memory safety.

⑦ `Obj` also offers `useObj` and `getX` operations. These are wrapped in order to work on `Wrapper`. Alternatively a `converter` from `Wrapper` to `ptr Obj` could be provided.

⑧ `useWrapper` shows how the wrapper can be used.

⑨ Sequences of `Wrapper` can be created easily and the memory management is automatic.

⑩ When `useWrapper` returns `w`'s destructor is called which calls the destructor of `Wrapper`. The destructor of `Wrapper` then calls `destroyObj` ensuring that there are no memory leaks.

> At the time of this writing, the example needs to use the `--mm:orc` or `--mm:arc` compiler switches. Otherwise the destruction of the sequence of `Wrapper` does not call `Wrapper`s destructor. Later versions of the compiler will make `--mm:orc` the default.

Instead of prohibiting the `=copy` operation via `{.error.}` the wrapper could also offer reference counting:

```
type
  Wrapper = object
    obj: ptr Obj
    rc: ptr int ①

proc `=destroy`(dest: var Wrapper) =
  if dest.obj != nil:
    if dest.rc[] == 0: ②
      dealloc(dest.rc) ③
      destroyObj(dest.obj)
    else:
      dec dest.rc[]

proc `=copy`(dest: var Wrapper; source: Wrapper) = ④
  inc source.rc[] ⑤
  `=destroy`(dest) ⑥
  dest.obj = source.obj ⑦
  dest.rc = source.rc

proc create(s: string): Wrapper =
  Wrapper(obj: createObj(cstring(s)),
          rc: cast[ptr int](alloc0(sizeof(int)))) ⑧
```

① `Wrapper` now has a reference count (`rc`). This must be a `ptr` or a `ref` and allocated on the heap so that its value is shared between different instances.

② Use the reference count to see if `destroyObj` needs to be called.

③ The reference count itself also needs to be deallocated because it is stored on the heap.

④ The copy operation.

⑤ Increment the `source`'s reference count first in order to protect against self assignments.

⑥ Destroy what was in `dest` as we are about to overwrite its contents.

⑦ Copy the data over.

⑧ `alloc0` allocates memory of a given size and sets the memory cells to zero. It is used here to initialize the reference count.

Chapter 30. Strict funcs

As an experimental mode called `strictFuncs` a stricter definition of "side effect" is available. In addition to the existing rule that a side effect is calling a function with side effects the following rule is also enforced:

A store to the heap via a `ref` or `ptr` indirection is not allowed.

For example:

```
{.experimental: "strictFuncs".}

type
  Node = ref object
    le, ri: Node
    data: string

func len(n: Node): int =
  # valid: len does not have side effects
  var it = n
  while it != nil:
    inc result
    it = it.ri

func mut(n: Node) =
  n.data = "yeah" # short for: m[].data = "yeah"
  # Error: 'mut' can have side effects
```

Mutations via `var T` and `out T` parameters continue to be allowed in a strict func. Hence a prototype such as `func add[T](s: var seq[T]; elem: sink T)` is valid and can be used in the following `func`:

```
func tokenize(input: string): seq[string] =
  result = @[]
  for w in splitWhitespace(input): result.add w
```

The mutation that `add` performs does not count as a side effect and so `tokenize` can be a `func`.

Chapter 31. View types

> View types are more effective with "strict funcs".

A view type is a type that is or contains one of the following types:

- `lent T` (view into `T`)
- `openArray[T]` (pair of (pointer to array of `T`, size))

For example:

```
type
  View1 = openArray[byte]
  View2 = lent string
  View3 = Table[openArray[char], int]
```

Exceptions to this rule are types constructed via `ptr` or `proc`. For example, the following types are *not* view types:

```
type
  NotView1 = proc (x: openArray[int])
  NotView2 = ptr openArray[char]
  NotView3 = ptr array[4, lent int]
```

The mutability aspect of a view type is not part of the type but part of the locations it's derived from.

A *view* is a symbol (a let, var, const, etc.) that has a view type.

Nim allows view types to be used as local variables. In the current implementation this feature needs to be enabled via `{.experimental: "views".}`.

A local variable of a view type *borrows* from the locations and it is statically enforced that the view does not outlive the location it was borrowed from.

For example:

```
{.experimental: "views".}

proc take(a: openArray[int]) =
  echo a.len

proc main(s: seq[int]) =
  var x: openArray[int] = s # 'x' is a view into 's'
  # it is checked that 'x' does not outlive 's' and
  # that 's' is not mutated.
  for i in 0 .. high(x):
    echo x[i]
  take(x)

  take(x.toOpenArray(0, 1)) # slicing remains possible
  let y = x  # create a view from a view
  take y
  # it is checked that 'y' does not outlive 'x' and
  # that 'x' is not mutated as long as 'y' lives.


main(@[11, 22, 33])
```

A local variable of a view type can borrow from a location derived from a parameter, another local variable, a global `const` or `let` symbol or a thread-local `var` or `let`.

Let `p` be the proc that is analyzed for the correctness of the borrow operation.

Let `source` be one of:

- A formal parameter of `p`. Note that this does not cover parameters of inner procs.
- The `result` symbol of `p`.
- A local `var` or `let` or `const` of `p`. Note that this does not cover locals of inner procs.
- A thread-local `var` or `let`.
- A global `let` or `const`.

- A constant array/seq/object/tuple constructor.

31.1. Path expressions

A location derived from `source` is then defined as a path expression that has `source` as the owner. A path expression `e` is defined recursively:

- `source` itself is a path expression.
- Container access like `e[i]` is a path expression.
- Tuple access `e[0]` is a path expression.
- Object field access `e.field` is a path expression.
- `system.toOpenArray(e, ...)` is a path expression.
- Pointer dereference `e[]` is a path expression.
- An address `addr e`, `unsafeAddr e` is a path expression.
- A type conversion `T(e)` is a path expression.
- A cast expression `cast[T](e)` is a path expression.
- `f(e, ...)` is a path expression if `f`'s return type is a view type. Because the view can only have been borrowed from `e`, we then know that owner of `f(e, ...)` is `e`.

If a view type is used as a return type, the location must borrow from a location that is derived from the first parameter that is passed to the proc. See Section 21.10, “Var return type” for details about how this is done for `var T`.

A mutable view can borrow from a mutable location, an immutable view can borrow from both a mutable or an immutable location.

If a view borrows from a mutable location, the view can be used to update the location. Otherwise it cannot be used for mutations.

The *duration* of a borrow is the span of commands beginning from the assignment to the view and ending with the last usage of the view.

For the duration of the borrow operation, no mutations to the borrowed locations may be performed except via the view that borrowed from the location. The borrowed location is said to be *sealed* during the borrow.

```nim
{.experimental: "views".}

type
  Obj = object
    field: string

proc dangerous(s: var seq[Obj]) =
  let v: lent Obj = s[0] # seal 's'
  s.setLen 0  # prevented at compile-time because 's' is sealed.
  echo v.field
```

The scope of the view does not matter:

```nim
{.experimental: "views".}

type
  Obj = object
    field: string

proc valid(s: var seq[Obj]) =
  let v: lent Obj = s[0]  # begin of borrow
  echo v.field            # end of borrow
  s.setLen 0  # valid because 'v' isn't used afterward
```

The analysis requires as much precision about mutations as is reasonably obtainable, so it is more effective with the experimental strict funcs feature (see Chapter 30, *Strict funcs*). In other words `--experimental:views`:option: works better with `--experimental:strictFuncs`:option:.

The analysis is currently control flow insensitive:

```nim
proc invalid(s: var seq[Obj]) =
  let v: lent Obj = s[0]
  if false:
    s.setLen 0
  echo v.field
```

In this example, the compiler assumes that `s.setLen 0` invalidates the borrow operation of `v` even though a human being can easily see that it will never do that at runtime.

31.2. Start of a borrow

A borrow starts with one of the following:

- The assignment of a non-view-type to a view-type.
- The assignment of a location that is derived from a local parameter to a view-type.

31.3. End of a borrow

A borrow operation ends with the last usage of the view variable.

31.4. Reborrows

A view `v` can borrow from multiple different locations. However, the borrow is always the full span of `v`'s lifetime and every location that is borrowed from is sealed during `v`'s lifetime.

Part III: Mastering Macros

Chapter 32. Introduction

So far we have seen simple examples of how Nim programs can be written and we have looked at how the various language features are defined in quite a formal manner. This part focusses on advanced programming techniques, in particular how Nim's macro system can be used to raise the level of abstraction.

This part assumes that the reader is already familiar with Chapter 28, *Macros*.

Chapter 33. AST introspection

The mapping from Nim's syntax to syntax trees is rather subtle. While the syntax is optimized for readability and conciseness the syntax trees are designed for ease of construction and traversal. Like in Lisp the tree consists of nested nodes where each node is of a certain "kind" such as "if statement" (`nnkIfStmt`) or "routine call" (`nnkCall`).

The mapping can easily be seen with `treeRepr`:

```
import std / macros

macro investigate(body: untyped) = ①
  echo treeRepr body ②

investigate:
  if undeclaredIdentifier == 3:
    echo "3"
  else:
    echo "not 3"
```

① Declares a macro called `investigate` that works on `untyped` trees.

② Calls `treeRepr` which produces a debug string of `body`.

Because macro expansion happens at compile time this program produces *at compile time*:

```
StmtList
  IfStmt
    ElifBranch
      Infix
        Ident "=="
        Ident "undeclaredIdentifier"
        IntLit 3
      StmtList
        Command
          Ident "echo"
          StrLit "3"
    Else
      StmtList
        Command
          Ident "echo"
          StrLit "not 3"
```

We can see that a list of statements `StmtList` is passed to `investigate`. In order to create a `StmtList` one can use `newTree(nnkStmtList, <children here>)`.

33.1. Typed vs untyped ASTs

The difference between `typed` and `unytyped` parameters is important for templates and it is even more important for macros. The AST that is passed to a `typed` macro parameter can differ significantly from an AST that is passed to an `untyped` macro parameter.

For example:

```nim
import std / macros

macro investigateTyped(body: typed) =
  echo treeRepr body

var needsToBeDeclaredIdentifier = 0
investigateTyped:
  if needsToBeDeclaredIdentifier == 3:
    echo "3"
  else:
    echo "not 3"
```

This program produces at compile time:

```
StmtList
  IfStmt
    ElifBranch
      Infix
        Sym "=="
        Sym "needsToBeDeclaredIdentifier"
        IntLit 3
      Command
        Sym "echo"
        HiddenStdConv
          Empty
          Bracket
            StrLit "3"
    Else
      Command
        Sym "echo"
        HiddenStdConv
          Empty
          Bracket
            StrLit "not 3"
```

Note how symbol lookups happened producing `nnkSym` nodes and the `echo` calls have mysterious hidden conversion nodes containing an `nnkBracket` node. In other words, `echo "3"` was transformed into `echo ["3"]` because `echo` uses a `varargs` parameter. Many details like these have to be understood before one can write a macro operating on `typed` ASTs. For this reason most of the following examples operate on `untyped` ASTs.

Chapter 34. AST creation

An AST can be created in different ways and these ways can all be combined freely. But one of the easiest ways is to use `quote do`. For example, an operator `==~` that checks if two floating point values almost equal can be written as a template:

```
template `==~`(x, y: float): bool = abs(x - y) < 1e-9
```

Or it can be written as a macro that uses `quote do`:

```
import std / macros

macro `==~`(x, y: float): bool =
  result = quote do:
    abs(`x` - `y`) < 1e-9
```

`quote do` turns a pattern of code into a `NimNode`. Inside the pattern backticks can be used to access symbols from the macro's scope. The `==~` macro can also be written as:

```
import std / macros

macro `==~`(x, y: float): bool =
  result = newCall(bindSym"<",
    newCall(bindSym"abs", newCall(bindSym"-", x, y)),
    newLit(1e-9))
```

In my opinion this more imperative style of AST creation is easier to understand for beginners and so it is what is used in the following more complex examples.

Chapter 35. Collect macro

As our first complex example we will look at how Nim's `collect` macro can be implemented. The standard library already contains `collect`, it can be found in `std/sugar`. `collect` is the preferred method of turning a potentially nested loop construct from a statement to an expression.

Instead of:

```
import std / tables

const Data = toTable({"a": 1, "b": 2, "c": 3})

var s = newSeq[string]()
for k, v in Data.pairs:
  if v mod 2 == 0:
    s.add k
```

One can use the more declarative:

```
import std / [tables, sugar]

const Data = toTable({"a": 1, "b": 2, "c": 3})

let s = collect(newSeq):
  for k, v in Data.pairs:
    if v mod 2 == 0: k
```

An an exercise we will reimplement `collect`. For a beginner, writing a macro is usually a hard task.

As the first step we postulate the code pattern that the macro needs to expand to: `collect(constructorCall): body` should be translated into something like:

```
block:
  var tmp = constructorCall[typeOf(body)]()
  sinkInto(body, tmp.add)
  tmp
```

where `sinkInto(body, tmp.add)` describes the AST where the final expression `x` of `body` is replaced by `tmp.add x`. We have to walk if-expressions, loops and "statement list expressions" to arrive at the "final expression" which is the part of the body that produces the value:

```
import macros

proc sinkInto(n, fullBody, res, bracketExpr: NimNode): NimNode = ①
  case n.kind
  of nnkStmtList, nnkStmtListExpr, nnkBlockStmt, nnkBlockExpr,
     nnkWhileStmt, nnkForStmt, nnkElifBranch, nnkElse, nnkElifExpr,
     nnkOfBranch, nnkExceptBranch: ②
    result = copyNimTree(n)
    if n.len >= 1:
      result[^1] = sinkInto(n[^1], fullBody, res, bracketExpr)
  of nnkIfExpr, nnkIfStmt, nnkTryStmt: ③
    result = copyNimTree(n)
    for i in 0..<n.len:
      result[i] = sinkInto(n[i], fullBody, res, bracketExpr)
  of nnkCaseStmt:
    result = copyNimTree(n)
    for i in 1..<n.len: ④
      result[i] = sinkInto(n[i], fullBody, res, bracketExpr)
  else:
    if bracketExpr.len == 1: ⑤
      bracketExpr.add(newCall(bindSym"typeof", fullBody))
    result = newCall(bindSym"add", res, n) ⑥

macro collect*(init, body: untyped): untyped =
  let res = genSym(nskVar, "collectResult") ⑦
  let bracketExpr = newTree(nnkBracketExpr, init) ⑧
  let transformedBody = sinkInto(body, body, res, bracketExpr) ⑨
  let call = newTree(nnkCall, bracketExpr) ⑩
  result = newTree(nnkStmtListExpr, newVarStmt(res, call),
                   transformedBody, res) ⑪
```

① Traverses `n` recursively and produces a copy of `n` except that the value producing subexpression `x` is replaced by `add(res, x)`. `fullBody` is the full body and it is used for producing `init[typeof(body)]()` which is accomplished by modifying `bracketExpr`.

② For `nnkStmtListExpr` and the like we only follow the last child. The last child can be accessed via `n[^1]`.

③ For if expressions and the like we follow all possible branches. This allows for code like `if cond: a else: b` to be transformed into: `if cond: add(tmp, a) else: add(tmp, b)`.

④ A `case` expression is just like an `if` expression except that we start from 1 here in order to skip the selection expression which is not the value producing expression that we are interested in.

⑤ If the bracketExpr is still the `init` expression, add `typeof(body)` to it producing `init[typeof(body)]`.

⑥ We arrived at the value producing expression. Transform it to `add(res, value)`.

⑦ We create a `(var res = init[typeof(body)]; transformedBody; res)` construct which is called an `nnkStmtListExpr` tree. `res` is a fresh variable produced from `macros.genSym`.

⑧ The construct `init[T]` is generated as an `nnkBracketExpr`.

⑨ Let `sinkInto` perform the recursive traversal.

⑩ We transform `init[T]` to `init[T]()`.

⑪ The result of the macro is this `(var res = init[typeof(body)]; transformedBody; res)` construct.

Note that this is a simplified implementation, the standard library's `collect` macro implements more features and is more flexible.

Chapter 36. strformat

Macros can be used to translate mini languages embedded inside string literals into Nim code. A good example for this is the standard library's `strformat` module.

Instead of `a & " " & $b & " " & c` you can write `fmt"{a} {b} {c}"`. Inside the string literal, curly braces enclose a Nim expression. The expression is turned into a string via `$`. The standard library's `fmt` supports many features for formatting strings, integers and floats, their precision and alignment. However, in our reimplementation we keep things simple: We only support curly braces and use `macros.parseExpr` to do the hard part of parsing the Nim subexpressions into Nim's AST.

It is good style to split up the tasks "parsing" and "synthesis" into different routines. Only the synthesis uses Nim's AST API. The parser/tokenizer is implemented as an iterator:

```
type TokenKind = enum  ①
  Literal ②
  NimExpr ③

iterator tokenize(s: string): (TokenKind, string) = ④
  var i = 0
  var tok = Literal ⑤
  while i < s.len:
    let start = i
    case tok
    of Literal:
      while i < s.len and s[i] != '{': inc i ⑥
    of NimExpr:
      while i < s.len and s[i] != '}': inc i ⑦
    yield (tok, s.substr(start, i-1))
    tok = if tok == Literal: NimExpr else: Literal ⑧
    inc i
```

① The tokenizer distinguishes only between two kinds of tokens.

② A `Literal` token means it should be interpreted literally. For example, the "abc" part from "{x}abc".

③ A `NimExpr` needs to be parsed as a Nim expression. For example, the "x" part from "{x}abc".

④ `tokenize` yields the determined tokens. A token is a pair of `(TokenKind, string)`.

⑤ The tokenizer starts in the state `Literal`.

⑥ If the tokenizer is in the state `Literal` it needs to proceed until either the end of the string is reached or until a '{' is found.

⑦ If the tokenizer is in the state `NimExpr` it needs to proceed until either the end of the string is reached or until a '}' is found.

⑧ After a `Literal` token a `NimExpr` token must follow and vice versa.

The `fmt` macro uses this `tokenize` iterator:

```
import macros

macro fmt*(pattern: static[string]): string = ①
  var args = newTree(nnkBracket) ②
  for (k, s) in tokenize(pattern): ③
    case k
    of Literal:
      if s != "":
        args.add newLit(s) ④
    of NimExpr:
      args.add newCall(bindSym"$", parseExpr(s)) ⑤
  if args.len == 0: ⑥
    result = newLit("")
  else:
    result = nestList(bindSym"&", args) ⑦

var x = 0.9
var y = "abc"

echo fmt"{x} {y}" ⑧
```

① `fmt` takes a `static[string]` as input. This means that inside the macro body `pattern` really is of type `string` and not of `NimNode` making the data easier to access.

② `args` collects all the arguments that we pass to the `&` operator.

③ We use the `tokenize` iterator and unpack the token tuple into `k` and `s`.

④ If the token is a `Literal` and not the empty string, we can append `s` to `args`. But we need to convert `s` to a `NimNode` first via `newLit(s)`.

⑤ If the token is a `NimExpr` we use `macros.parseExpr` to parse it into a `NimNode`. We then wrap the node and use it as an argument to a call of the `$` operator. Thus `fmt` supports any expression that can be turned into a string via `$`.

⑥ For a call like `fmt""` it is possible that `args` remains empty. We map this case to the empty string literal "".

⑦ Else we call the concatenation operator `&` with `args`. However `&` only accepts two arguments so we need to turn `&[a, b, c]` to `(a & b) & c`. This nesting of arguments is performed by `macros.nestList`.

⑧ Produces the output: "0.9 abc".

Chapter 37. strscans

The `tokenize` iterator, as it was implemented in the previous chapter, is straight-forward but *imperative* code, and the task of parsing comes up frequently in day to day programming. Ideally we want to program in a *declarative* way; only describing how to *extract* the desired data via patterns and not how to advance any required auxiliary cursors, for example. The standard library offers the relatively unknown module `strscans` that helps with this task.

`strscans.scanTuple` can be used to extract data into a custom tuple type. The tuple type depends on the pattern that is tried to match. For example:

```
import std / strscans

const InputData = "1000-01-01 00:00:00" ①

let (ok, year, month, day, time) = scanTuple(InputData, "$i-$i-$i$s$+") ②
if ok:
  assert year == 1000 ③
  assert month == 1 ④
  assert day == 1 ⑤
  assert time == "00:00:00" ⑥
```

① `InputData` contains a date a clock time that we seek to parse.

② The string "$i-$i-is$"` is a description of how to extract the data. Characters are matched verbatim except for substrings starting with `$`. `$i` means to expect a string substring that can be parsed into an `int`. `$s` means to skip optional whitespace. `$ matches the rest of the input.

③ `year` is inferred to be of type `int` because it corresponds to the first `$i` pattern. For `InputData` its value is `1000`.

④ month is inferred to be of type int because it corresponds to the second $i pattern. For InputData its value is 1.

⑤ day is inferred to be of type int because it corresponds to the third $i pattern. For InputData its value is 1.

⑥ time is inferred to be of type string because it corresponds to the $+ pattern. For InputData its value is "00:00:00".

scanTuple needs to produce a tuple of variable length depending on the pattern that we pass to it. The first component of the tuple is always of type bool and contains the information if the parse was successful.

In order to simplify the task that scanTuple has to do, we first create a couple of helper routines operating on a parsing State:

```
import std / parseutils

type
  State = object ①
    i: int
    err: bool

proc matchChar(s: string; c: var State; ch: char) = ②
  if not c.err:
    if c.i < s.len and s[c.i] == ch:
      inc c.i
    else:
      c.err = true

proc skipWhitespace(s: string; c: var State) = ③
  if not c.err:
    while c.i < s.len and s[c.i] in {' ', '\t', '\n', '\r'}: inc c.i

proc matchInt(s: string; res: var int; c: var State) = ④
  if not c.err:
    let span = parseInt(s, res, c.i)
    if span > 0:
      inc c.i, span
    else:
      c.err = true

proc matchRest(s: string; res: var string; c: var State) = ⑤
  if not c.err:
    res = s.substr(c.i)
```

① The parsing State consists of the current parsing position i and an error flag called err. Once err is true, it is never reset to false.

② `matchChar` tries to match a single character `ch`.

③ `skipWhitespace` skips optional whitespace.

④ `matchInt` tries to match the input at position `c.i` against an integer. It does so with the help of the standard library's `parseutils.parseInt` function.

⑤ `matchRest` matches the rest of the input string and stores it into `res`.

This design with an explicit error state allows us to emit sequential code rather than (potentially deeply) nested `if` statements:

```
import std / macros

macro scanTuple*(input: string; pattern: static[string]): untyped = ①
  var i = 0
  var body = newTree(nnkStmtList) ②
  var tup = newTree(nnkTupleConstr) ③
  tup.add newLit(true)
  let stateVar = genSym(nskVar, "stateVar")
  let res = genSym(nskVar, "scanResult")
  while i < pattern.len:
    if pattern[i] == '$':
      inc i
      case pattern[i]
      of 'i': ④
        body.add newCall(bindSym"matchInt", input,
          newTree(nnkBracketExpr, res, newLit(tup.len)), stateVar)
        tup.add newLit(0)
      of 's': ⑤
        body.add newCall(bindSym"skipWhitespace", input, stateVar)
      of '+': ⑥
        body.add newCall(bindSym"matchRest", input,
          newTree(nnkBracketExpr, res, newLit(tup.len)), stateVar)
        tup.add newLit("")
      else:
        error "invalid pattern"
      inc i
    else: ⑦
      body.add newCall(bindSym"matchChar", input,
        stateVar, newLit(pattern[i]))
      inc i

  result = newTree(nnkStmtListExpr, ⑧
    newVarStmt(res, tup),
    newVarStmt(stateVar, newTree(nnkObjConstr, bindSym"State")),
    body,
    newAssignment(newTree(nnkBracketExpr, res, newLit(0)),
                  newCall(bindSym"not", newDotExpr(stateVar, ident"err"))),
    res)
```

```
when defined(debugScanTuple):
  echo repr result ⑨
```

① Because the exact return tuple type depends on `pattern`, only `untyped` can be used as the return type.

② `body` collects the list of statements that contains the calls to the helpers `matchChar`, `skipWhitespace`, `matchInt`, and `matchRest`.

③ `tup` collects the resulting tuple value (not the tuple type!).

④ We map the pattern `$i` to a call to `matchInt`.

⑤ We map the pattern `$s` to a call to `skipWhitespace`.

⑥ We map the pattern `$+` to a call to `matchRest`.

⑦ Every other character in pattern is mapped to `matchChar`.

⑧ The result of `scanTuple` is a statement list expression roughly like `(var scanResult = (false, …); var stateVar = State(); body; scanResult[0] = not stateVar.err; scanResult)`.

⑨ The `when …` section allows us to inspect the produced code easily.

If we compile the program with the switch `--define:debugScanTuple` it enables the line `echo repr result` so at compile-time the produced AST is written to standard output:

```
var scanResult_123 = (true, 0, 0, 0, "")
var stateVar_456 = State()
matchInt(InputData, scanResult_123[1], stateVar_456)
matchChar(InputData, stateVar_456, '-')
matchInt(InputData, scanResult_123[2], stateVar_456)
matchChar(InputData, stateVar_456, '-')
matchInt(InputData, scanResult_123[3], stateVar_456)
skipWhitespace(InputData, stateVar_456)
matchRest(InputData, scanResult_123[4], stateVar_456)
scanResult_123[0] = not(stateVar_456.err)
scanResult_123
```

`echo repr result` is an idiom worth remembering; it is important for debugging macro code and also allows for an easier development process.

Chapter 38. HTML trees

Embedding mini languages within string literals is often not the best way to model a problem domain. An alternative is to leverage the full power of Nim's syntax. A templating system for convenient HTML tree generation is a good example here.

But before we can outline the macro's design we need to model the HTML tree:

```
type
  Tag* = enum ①
    text, html, head, body, table, tr, th, td
  TagWithKids = range[html..high(Tag)] ②
  HtmlNode* = ref object ③
    case tag: Tag
    of text:
      s: string
    else:
      kids: seq[HtmlNode]

proc newTextNode*(s: sink string): HtmlNode = ④
  HtmlNode(tag: text, s: s)

proc newTree*(tag: TagWithKids; kids: varargs[HtmlNode]): HtmlNode = ⑤
  HtmlNode(tag: tag, kids: @kids)

proc add*(parent: HtmlNode; kid: sink HtmlNode) = parent.kids.add kid

from std / xmltree import addEscaped

proc toString(n: HtmlNode; result: var string) = ⑥
  case n.tag
  of text:
    result.addEscaped n.s
  else:
    result.add "<" & $n.tag
    if n.kids.len == 0:
```

```
      result.add " />"
    else:
      result.add ">\n"
      for k in items(n.kids): toString(k, result)
      result.add "\n</" & $n.tag & ">"

proc `$`*(n: HtmlNode): string = ⑦
  result = newStringOfCap(1000)
  toString n, result
```

① Reduced list of possible HTML tags.

② A subtype of `Tag` that covers the tags that have kids.

③ A tree of HTML modelled via a `case object`.

④ `newTextNode` constructs a single text node.

⑤ `newTree` constructs a node with a variable number of children.

⑥ `toString` is recursive and uses a `var string` parameter as its buffer to write to. This `var` parameter is crucial for efficiency.

⑦ For convenience a dollar operator is provided that allocates a large buffer and then calls `toString` to make effective use of this buffer.

`newTextNode`, `newTree`, and `add` are good enough to produce complex HTML tables:

```
proc toTable(headers: openArray[string]; data: seq[seq[int]]): HtmlNode = ①
  assert headers.len == data.len ②
  var tab = newTree(table)
  for i in 0..<data.len:
    var row = newTree(tr, newTree(th, newTextNode(headers[i])))
    for col in data[i]:
      row.add newTree(td, newTextNode($col))
    tab.add row ③
  result = newTree(html, newTree(body, tab)) ④
```

① `toTable` produces a 2 dimensional HTML table from `headers` and `data`.

② We require that every column has a corresponding header.

③ We must not forget to append the temporary `row` to `tab`.

④ The table is wrapped inside `<html><body>…</body></html>`.

This style of programming is low level and error prone; it is easy to forget to append `row` to `tab`, for example. Instead we would like to write the following:

```
proc toTable(headers: openArray[string]; data: seq[seq[int]]): HtmlNode =
  assert headers.len == data.len
  result = buildHtml:
    body:
      table:
        for i in 0..<data.len:
          tr:
            th:
              text headers[i]
            for col in data[i]:
              td:
                text $col
```

The domain specific language should compose with ordinary Nim code, we want to be able to use ordinary `if` and `for` statements inside the HTML templating system.

The required `buildHtml` macro needs to walk the passed AST recursively and introduce temporary variables for each `if` and `for` statement. Every enum value of `TagWithKids` is translated to a `newTree` call, a call to `text` is translated to `newTextNode`:

```
import macros

proc whichTag(n: NimNode): Tag = ①
  for e in low(TagWithKids)..high(TagWithKids):
    if n.eqIdent($e): return e ②
  return text ③

proc traverse(n, dest: NimNode): NimNode = ④
  if n.kind in nnkCallKinds: ⑤
    if n[0].eqIdent("text"):
      expectLen n, 2
      result = newCall(bindSym"newTextNode", n[1]) ⑥
      if dest != nil:
        result = newCall(bindSym"add", dest, result)
    else:
      let tag = whichTag(n[0])
      if tag == text:
        result = copyNimNode(n) ⑦
        result.add n[0]
        for i in 1..<n.len:
          result.add traverse(n[i], nil)
      else:
        let tmpTree = genSym(nskVar, "tmpTree") ⑧
        result = newTree(nnkStmtList,
          newVarStmt(tmpTree, newCall(bindSym"newTree", n[0])))
```

```
        for i in 1..<n.len:
          result.add traverse(n[i], tmpTree)
        if dest != nil:
          result.add newCall(bindSym"add", dest, tmpTree)
  else:
    result = copyNimNode(n) ⑨
    for child in n:
      result.add traverse(child, dest)

macro buildHtml(n: untyped): untyped =
  let tmpTree = genSym(nskVar, "tmpTree")
  var call = newCall(bindSym"newTree", bindSym"html")
  result = newTree(nnkStmtListExpr, newVarStmt(tmpTree, call))
  result.add traverse(n, tmpTree) ⑩
  result.add tmpTree ⑪
```

① `whichTag` returns the tag that the call operation corresponds to.

② `body:` is mapped to `Tag.body` etc.

③ It returns `text` if it is not any tag.

④ `traverse` does the bulk of the work. It traverses `n` and produces a modified copy of the AST. `dest` is the potential destination of where to attach the `HtmlNode` to.

⑤ If the node is any kind of "call expression" we examine if it is a call to the `text` operation.

⑥ If so, it translates `text x` to `newTextNode(x)`.

⑦ If the call is not a call to a tag simply traverse `n` recursively.

⑧ If the call is a tag transform it into `(var tmpTree = newTree(tag); translatedBody; dest.add tmpTree)`.

⑨ For any node that is not a call expression traverse `n` recursively.

⑩ `buildHtml` calls the `traverse` auxiliary proc.

⑪ `buildHtml` transforms `n` into `(var tmpTree = newTree(html); traverse(n); tmpTree)` which is an `nnkStmtListExpr` that produces a value of type `HtmlNode` so that it can be bound to a variable.

Chapter 39. Advice

The examples we have seen so far do not only show how macros can be implemented; they are also supposed to show good design of domain specific languages (DSL).

In general a library that uses macros to good effect focuses on:

1. Composability. It should be possible to combine different DSLs into a coherent program.
2. Understandability. Hide the right amount of details but do not hide important aspects of your programs. For example, injecting temporary variables is almost always beneficial and arguably a large part of what drives the distinction between low level and high level programming. On the other hand, hiding control flow can make your programs more brittle. Beware of designs that focus on “single character” DSLs such as regular expressions and what `strformat` offers; these are inherently not scalable as single characters are hard to remember and it is not obvious how to arrive at a design where optional whitespace can be inserted for better readability.
3. Documentation. Macros should be documented well. The performed transformations should be outlined and the design ideas behind the DSL should be documented.

Part IV: Mastering Parallelism

Chapter 40. Introduction

Any modern programming language has to offer plenty of support for concurrency and parallelism. The reason is that even tiny embedded devices have CPUs with multiple cores these days. In order to make effective use of these systems the programmer has to give hints to the compiler and runtime system and outline which parts of the program can run in parallel.

Concurrency is the overlapped execution of tasks whereas parallelism is the simultaneous execution of tasks. Concurrency can be beneficial as a program structuring mechanism and is tied to the notion of “blocking” or rather its avoidance. The primary example for the benefits of concurrency is a server which should be responsive to new requests even though old requests have not yet finished. Parallelism is a means to make programs run faster and a good example use case for it is graphics programming: Every pixel can be accessed independently of the other pixels on the screen and can be written to in order to change its color and produce a shape.

The mechanisms we look at here are useful both for concurrency and parallelism.

Chapter 41. Threading

There are two ways to introduce parallelism in Nim: A rather low-level API for thread creation where a thread of execution is directly provided by the operating system, and a high-level API for annotating *potential* parallelism that the runtime is free to exploit, but does not have to. The reason for this is that sending a task to a different processor to execute might be more expensive than executing it on the current processor immediately.

A Nim program starts to run in the so-called “main thread”. The main thread runs every top level statement. In order to run things in parallel at least one additional thread needs to be created.

41.1. `createThread`

The low-level API is centered around a `Thread[T]` type. A `Thread[T]` runs a proc of the type `proc (x: T)`, in other words: the `T` is the type of a single parameter than can be passed to the proc at thread creation:

```
proc worker(s: string) {.thread.} =     ①
  echo s

var background: Thread[string]          ②
createThread background, worker, "abc" ③
echo "xyz"                              ④
joinThread background                   ⑤
echo "control flow converged"           ⑥
```

① The proc `worker` runs in parallel with the main thread. So it is annotated as a `thread` proc. The `thread` pragma is an alias for `gcsafe`.

② Declares a variable named `background` of type `Thread` of `string`.

③ Creates a single background thread and attaches it to a variable named `background`.

④ Outputs "xyz" but competes with the output of the background thread.

⑤ Waits for the background thread to finish execution.

⑥ At this point the background thread has finished and control flow continues in the main thread only.

This program can either produce "abc" followed by "xyz" or "xyz" followed by "abc". It is not deterministic because both the background thread and the main thread compete for the `stdout` stream resource that `echo` uses and one threads gets to output its message first.

41.2. Single worker, single channel

Creating a thread is an expensive operation and so usually one tries to keep a thread running and give it more than one task. This is typically accomplished by having the thread run a loop that waits on a queue or channel for items to work on:

```
var chan: Channel[string]              ①
chan.open()                            ②

proc worker() {.thread.} =             ③
  while true:                          ④
    let task = chan.recv()             ⑤
    if task.len == 0: break            ⑥
    echo task                          ⑦

proc log(msg: string) =                ⑧
  chan.send msg

var logger: Thread[void]               ⑨
createThread logger, worker            ⑩
log "a"                                ⑪
log "b"
log "c"
log ""                                 ⑫
joinThread logger                      ⑬
chan.close()                           ⑭
```

① Declares a channel named `chan` of type `Channel` of `string`.

② A channel has to be opened before something can be sent.

③ The proc `worker` runs in parallel with the main thread. So it is annotated as a `thread` proc. The `thread` pragma is an alias for `gcsafe`.

④ The `while true` loop and `break` inside the loop is a common idiom when processing channel items.

⑤ Blocks until an item arrives at `chan`.

⑥ In this rather adhoc protocol an empty string indicates that the thread should stop processing.

⑦ Processes the task.

⑧ Logging a message is as simple as sending the string to `chan`.

⑨ The thread does not need any data at startup so a `Thread` of `void` is used.

⑩ Attaches the `worker` proc to the `logger` thread variable.

⑪ The strings "a" then "b" then "c" should be logged.

⑫ Sends the empty string to tell the background thread to shut down.

⑬ Waits for the background to finish.

⑭ A channel must be closed explicitly to free its resources.

41.3. Multiple workers, single channel

So far we have only used a single background thread. Together with the main thread we can only make effective use of 2 CPU cores. In order to make use of all available cores an array of worker threads can be used. The following program outlines how this can be done:

```
var chan: Channel[string]
chan.open()

proc worker(threadIndex: int) {.thread.} =    ①
  while true:
    let task = chan.recv()
    if task.len == 0: break
    echo "Thread ", threadIndex, ": ", task   ②

proc log(msg: string) =
  chan.send msg

var loggers: array[8, Thread[int]]            ③
for i in 0 ..< loggers.len:
  createThread loggers[i], worker, i          ④
log "a"                                       ⑤
log "b"
log "c"
for i in 0 ..< loggers.len:
  log ""                                      ⑥
joinThreads loggers                           ⑦
chan.close()
```

① A thread index is passed to every worker thread.

② The thread index is echoed in addition to `task`.

③ Declares a thread pool named `loggers` with 8 entries of type `Thread[int]`.

④ Starts up every thread in the array `loggers`. The index `i` is passed to the `threadIndex` parameter.

⑤ Logs some messages.

⑥ Sends every thread in the thread pool the "should stop" message.

⑦ Waits for all threads in the thread pool to stop execution.

Chapter 42. `spawn`

The `Channel` and `Thread` types are available since version 1 of Nim and are widely used. It is common to declare these as global variables. In fact it is highly recommended to do so! While global variables are usually discouraged, in this case they are justified: There is no aliasing possible between global variables so it is easy to see how many threads are used and which channels they use and how the flow of communication looks. The topology of the program is clear.

A more elegant mechanism that abstracts away the nitty-gritty details of manual thread pool creation and task delivery is available via `spawn`: `spawn f(args)` runs `f(args)` in parallel or concurrently or potentially in parallel, depending on the implementation. `spawn` should not be confused with `createThread` — it wraps the `f(args)` call in a task and passes this task to some already existing thread pool. So the `f` should *not* be a long-running function that receives data from a channel, it should simply be an operation that is expensive enough to be worthwhile to run in parallel. But depending on the current load of the machine it might not actually run in parallel. Variations of `spawn` are implemented in different third-party libraries. We focus here on the `spawn` implementation that is provided by the "Malebolgia" library. Malebolgia is the successor of `std / threadpool` and is particularly simple and effective. (Run the command `nimble install malebolgia` to install it.)

Malebolgia only supports "structured concurrency" which means there is a clear point in the source code where all parallel flows of control converged. In Malebolgia this is at the end of an `awaitAll` environment:

```nim
import malebolgia       ①

proc f(i: int) = echo i ②

var m = createMaster()  ③
m.awaitAll:             ④
  for i in 0 ..< 10:
    m.spawn f(i)        ⑤
# all tasks are complete  ⑥
```

① Imports the malebolgia dependency that this example requires.

② `f` is what we run in parallel.

③ Declares a variable `m` of type `Master` which is used for synchronization.

④ Waits until all tasks are completed.

⑤ Creates a task that runs `f(i)` on the hidden thread pool.

⑥ By construction we know that all tasks have been completed at this point.

If a `spawned` task raises an exception, the master object notices and rethrows the exception after `awaitAll`. If multiple tasks raise an exception only the first exception is kept and rethrown.

Please be aware that a `spawn` in Malebolgia is a *hint*. The library is allowed to ignore the request for parallelism. In other words the above program might execute as if `spawn` and `awaitAll` would not exist:

```nim
proc f(i: int) = echo i

for i in 0 ..< 10:
  f(i)
```

42.1. Return values

In some implementations of `spawn`, if the spawned function has a return value of type `T` the type of `spawn f` is a `FlowVar` of type `T`. `FlowVar` is short for “data flow variable”. It has the interesting property that data races are prevented by construction: A data flow variable can be written to only once and a read operation blocks until it was written to. In other words, a read operation synchronizes.

But the `awaitAll` operation also synchronizes! Malebolgia takes this insight to do away with a FlowVar wrapping type. This is particularly useful when every spawn should write to a distinct array location, because we can keep working with seq[T] as opposed to seq[FlowVar[T]].

The following example is a standard benchmark for parallel programming and demonstrates this benefit:

```
import malebolgia

proc dfs(depth, breadth: int): int {.gcsafe.} = ①
  if depth == 0: return 1

  var sums = newSeq[int](breadth) ②
  var m = createMaster() ③
  m.awaitAll: ④
    for i in 0 ..< breadth:
      m.spawn dfs(depth - 1, breadth) -> sums[i] ⑤

  result = 0
  for i in 0 ..< breadth:
    result += sums[i] ⑥

echo dfs(8, 8) ⑦
```

① The dfs proc needs to be annotated with gcsafe manually because it is recursive. Reminder: gcsafe means that it does not access global variables that use managed memory.

② We collect the results of the subtasks in the seq sums.

③ Creates a Master object m for task coordination.

④ Synchronizes all spawned tasks using an awaitAll block.

⑤ Spawns subtasks recursively and stores the result in sums[i]. For an explanation of the arrow ->, see the text below.

⑥ After the awaitAll operation we can read from sums without any locking or synchronization.

⑦ Shows how to invoke dfs and output its result.

The most important aspect of this example is the -> notation: Symbols that denote the target location of the spawn (sums in this case) are treated specially in Malebolgia. The awaitAll macro ensures that these symbols are only written to and are not used in any context that could imply a read operation. The macro detects simple “read/write” and “write/write” conflicts.

42.2. Sharing memory

A spawned task might run on a different thread than the calling thread or it might not. The reason is that the task creation step can be more expensive than running the code directly. This implies that an operation that waits for an event to occur and is scheduled before the operation that triggers this event can lead to a program making no progress. A special case of this scenario can happen with channels: If the `recv` operation is scheduled before a `send` operation and both operations are scheduled to run on the same thread.

In fact, channels are much more low level than people realize: every `send` must be paired with a corresponding `recv` operation and yet across most (if not all) programming languages and libraries there is no static check to ensure this!

Instead, memory can be shared and locks should then be used to ensure that no data races can happen. Malebolgia offers a type `Locker[T]` that wraps a container of type `T` and enforces proper locking operations. The wrapped value can be accessed as `lock x as y` where `x` is the `Locker` object and `y` is a fresh identifier that denotes the wrapped object, or it can be accessed as `unprotected x as y` when the wrapped object should be accessed without a locking operation. Inside a concurrently running operation one needs to use `lock` but after the `awaitAll` we can use `unprotected`:

```
import std / [strutils, tables]
import malebolgia
import malebolgia / lockers

proc countWords(filename: string; results: Locker[CountTable[string]]) = ①
  for w in splitWhitespace(readFile(filename)): ②
    lock results as r: ③
      r.inc w ④

proc main() =
  var m = createMaster()
  var results = initLocker initCountTable[string]() ⑤
  m.awaitAll:
    m.spawn countWords("fileA.txt", results) ⑥
    m.spawn countWords("fileB.txt", results)
  unprotected results as r: ⑦
    r.sort() ⑧
    echo r
```

```
main()
```

① `countWords` takes a `Locker[CountTable[string]]` which is comparable to a `var CountTable[string]`.

② Iterates over every word of the input file. A "word" is a substring separated by whitespace.

③ Acquires `results`'s attached lock and accesses the `CountTable[string]` under the name of `r`. The lock is released after the block of code that is passed to the `lock` macro.

④ Under the protection of the lock, tell the `CountTable` to count the word `w`.

⑤ Creates an object of type `Locker[CountTable[string]]` and binds it to the name `results`.

⑥ Runs `countWords` for two different files in parallel.

⑦ Accesses the underlying `CountTable` as `r` without any locking. This is safe because we know that after the `awaitAll` operation all parallel processing is complete.

⑧ Sorts the `CountTable` so that the most commonly used words come first in the output.

There are few restrictions on what values can be shared between threads and thus what can be wrapped in a `Locker` but as usual pointers and the nonrestrictive aliasing they allow for cause trouble.

For example, the following program easily subverts the protection of the lock:

```
proc example(results: Locker[ptr int]) =
  var x: ptr int
  lock results as r:
    x = r # create an alias
  # store outside of the lock:
  x[] = 13
```

The situation is worse for `ref`. A `ref T` pointer is implemented with reference counting and based on the lifetime-tracking hooks. For performance reasons however, the reference counting does not use atomic CPU instructions. Refs cannot be shared in Nim, but they can be moved across threads! For such a move to be successful, a whole subgraph must be moved and no external references to the data must remain. Even read accesses can be harmful as they keep local variables alive to a point where their destructors introduce hidden write accesses that can cause data races:

```
proc use(x: ref int) = discard "nothing to do"

proc example(results: Locker[seq[ref int]]) =
  var x = new(int)
  lock results as r:
    r.add x
  use x ①
  # scope of `x` ends ②
```

① The use of `x` outside of the lock keeps `x` alive and so it is not moved to `r` but copied.

② The destructor for `x` runs here and produces a potential data race! The program is invalid should a different thread run code like `lock results as r: r.setLen 0` at the same time that the destructor runs.

The following code avoids this problem:

```
proc use(x: ref int) = discard "nothing to do"

proc example(results: Locker[seq[ref int]]) =
  var x = new(int)
  use x ①
  lock results as r:
    r.add ensureMove x ②
```

① The unprotected access should happen before `x` is added to `results`.

② The compiler has to ensure that `x` is moved into `results`.

Chapter 43. Isolated data

We can avoid the need to constantly watch out for subtle problems with `ref T` can be avoided by using Nim's `Isolated[T]` type, provided by the module `std / isolation`.

`Isolated` is a type that has the following declaration and associated routines:

```
type
  Isolated*[T] = object
    value: T

proc `=copy`*[T](dest: var Isolated[T]; src: Isolated[T]) {.error.}
  ## Isolated data can only be moved, not copied.

proc `=sink`*[T](dest: var Isolated[T]; src: Isolated[T]) =
  # delegate to value's sink operation
  `=sink`(dest.value, src.value)

proc `=destroy`*[T](dest: var Isolated[T]) =
  # delegate to value's destroy operation
  `=destroy`(dest.value)

proc isolate*[T](value: sink T): Isolated[T]
  ## Creates an isolated subgraph from the expression `value`.
  ## Isolation is checked at compile time.

proc unsafeIsolate*[T](value: sink T): Isolated[T] =
  ## Creates an isolated subgraph from the expression `value`.
  ## Warning: The proc doesn't check whether `value` is isolated.
  Isolated[T](value: value)

proc extract*[T](src: var Isolated[T]): T =
  ## Returns the internal value of `src`.
  ## The value is moved from `src`.
  result = move(src.value)
```

Construction must ensure that the invariant holds, namely that the wrapped `T` is free of external aliases into it. To ensure this property, construction must be done via the proc `isolate` (or via the unchecked unsafe `unsafeIsolate` proc). The `isolate` proc is the only operation that needs special language support; it performs an "isolation check".

How this isolation check is performed is beyond the scope of the book and the used algorithm is being refined frequently. But a crucial insight is that calls of `noSideEffect` routines are safe to isolate as long as variables that are of type `ref` or contain a `ref` are not used inside the call expression:

```
import std / isolation

var global: ref int

func identity(x: ref int): ref int = x
func select(cond: bool; x, y: ref int): ref int =
  (if cond: x else: y)

proc main =
  let a = isolate(new(ref int)) ①
  let local = new(ref int) ②
  let b = isolate(local) ③
  global = local ④
  let c = isolate select(true, identity(new(ref int)), new(ref int)) ⑤
```

① `new(ref int)` creates an object on the heap that cannot possibly be aliased yet. It is safe to be "isolated".

② Once `new(ref int)` is assigned to the variable `local` this variable could used later on breaking the isolation.

③ Hence `isolate(local)` produces a compile-time error, saying that `local` cannot be isolated.

④ `local` is assigned to `global` which could be used later on to break the isolation.

⑤ Nested calls can be isolated too, as long as the calls are `noSideEffect` and no variables are involved.

In the context of an isolation check, object constructions such as `MyRefObject(a: x, b: y)` can be treated like routine calls and hence are allowed to be isolated.

The `Isolated[T]` type is powerful enough to model linked lists. The freedom of data races is ensured at compile time. The following program exposes these ideas and uses `createThread` instead of `spawn` in order to show that `Isolated[T]` works with the low-level threading API too:

```
import std / [os, locks, isolation]

type
  MyList {.acyclic.} = ref object  ①
    data: string
    next: Isolated[MyList]         ②

template withMyLock*(a: Lock, body: untyped) = ③
  acquire(a)
  {.gcsafe.}: ④
    try:
      body
    finally:
      release(a)

var head: Isolated[MyList] ⑤
var headLock: Lock; initLock headLock ⑥

proc send(x: sink string) =
  withMyLock headLock:
    head = isolate MyList(data: x, next: move head) ⑦

proc worker() {.thread.} =
  var workItem = MyList(nil)
  var endReached = false
  while true:
    withMyLock headLock:
      workItem = extract head ⑧
      if workItem != nil:
        head = move workItem.next ⑨
    if workItem.isNil: ⑩
      if endReached: break
      os.sleep 30 ⑪
    else:
      if workItem.data.len == 0: ⑫
        endReached = true
      else:
        echo workItem.data

var thr: Thread[void]     ⑬
createThread(thr, worker)
send "abc"
send "def"
send ""
joinThread(thr)
```

① The node type a linked list consists of. It has to be annotated with `acyclic` so that the cycle collector does not get involved; it does not support objects that are shared between threads.

② Instead of the typical `next: MyList` we declare `next` to be of type `Isolated[MyList]` to enforce the invariant that list nodes can only be moved between threads and not copied and so do not require synchronization via atomic instructions or locks.

③ Even though `std / locks` offers a `withLock template` we define our own here that adds `{.gcsafe.}`.

④ Since we seek to use `ref` freely in the block without triggering the notion of "gcsafety" (see Section 26.5, "GC safety effect") we wrap the whole block in a `{.gcsafe.}` environment.

⑤ The `head` of the linked list must be of the type `Isolated[MyList]` in order to be protected by Nim's type system.

⑥ Multiple threads access the `head` of the list potentially at the same time. At runtime `head` is protected by the `headLock` lock.

⑦ Moves the old value of `head` into the `next` field of the constructed node and isolates this node. Then this node becomes the new `head` of the singly linked list. The isolation succeeds because `head` itself is of type `Isolated` so uniqueness is preserved as long as it is moved from.

⑧ While traversing the list in the worker thread we unlink the node that we seek to work on.

⑨ After this `move` operation the object that `workItem` points to is isolated.

⑩ Since `workItem` is isolated, we can proceed with the rest of the logic outside of the `withMyLock` environment.

⑪ The item is nil and we have not yet reached the stop token, so wait and give the other thread time to send more tasks.

⑫ As in previous examples we use the empty string to denote a stop token. Due to the singly linked list and the reversing nature of the traversal (both `send` and receive use the `next` field) there can be nodes after the stop token. Thus we have as an ending condition: The end token has been received and the traversal arrives at a `nil` node.

⑬ The usual plumbing code to setup the worker thread and send it some tasks.

Chapter 44. Smart pointers

As we can see, `Isolated[T]` is rather hard to work with and effectively turns an existing `ref` into a "unique ref". It is most useful when the `ref` type stems from a library that we have no control over.

If we have the control over the used pointer type and the pointer is used in a concurrent setting then `ref` should be avoided and instead a more refined "smart pointer" type should be used. For example, the "smartptrs" module from the "threading" package provides the `SharedPtr`, `ConstPtr` and `UniquePtr` types.

A `SharedPtr` is a reference-counted pointer type that uses atomic instructions and thus can be shared between threads. A `ConstPtr` is a `SharedPtr` that enforces that the data it wraps can only be used for read accesses. And finally a `UniquePtr` is a pointer that has a single owner and can only be moved around.

The following snippet outlines how `SharedPtr` can be used to create a singly linked list:

```
import threading/smartptrs

type
  MyList = object  ①
    data: string
    next: SharedPtr[MyList]       ②

var head, tail: SharedPtr[MyList] ③

proc send(x: sink string) =
  withMyLock headLock:
    tail = newSharedPtr MyList(data: x, next: move tail) ④
    if head.isNil: head = tail ⑤
```

① The node type a linked list consists of.

② The `next` field is of type `SharedPtr[MyList]` so that there can be more references to it than just the owning reference. This makes list traversals convenient to write as we can avoid the destructive moves during traversal.

③ The `head` and `tail` of the list have to be of type `SharedPtr[MyList]` too.

④ Appending to the list works much like insertion in the `Isolated[T]` case.

⑤ Since a `SharedPtr` supports a copy operation, both `head` and `tail` can point to the same object.

The program using `SharedPtr` arguably can be a little easier than its `Isolated` variant, but both are no match for a `seq` container that is wrapped in a lock or a dedicated channel data structure. Pointers are hard to use, especially in a multithreading setting. Modern Nim code avoids pointers for this reason. It is far easier to program in a world of *values* with restricted aliasing capabilities.

Chapter 45. Parallel for each and reduce

There is a form of sharing memory that is particularly simple and effective, and that requires neither locks nor atomic instructions:

1. "Parallel for each": If a simple operation should be applied to each element of an array and if the iteration order does not matter, the loop can run in parallel.
2. "Parallel reduce": A sum or product over an array of numbers can be computed by splitting up the array in *disjoint* slices, then computing the sum or product of every slice and then combining the intermediate results.

The slices we work on need to be *sendable* to the thread pool's internal task queue, and without causing a copy of the data. Only a `(pointer, length)` pair should be transmitted. Unfortunately, Nim's `openArray` type is not sendable between threads because the compiler cannot guarantee safe access nor safe lifetimes. Instead, we use the type `ptr system.UncheckedArray[T]` which is exactly what its name suggests: A raw unsafe pointer to an array of unspecified size. There is no index checking. We compute the address of an array element with a custom operator `@!` in order to make the rest of the code more pleasing to the eye:

```
template `@!`[T](data: openArray[T]; i: int): untyped = ①
  cast[ptr UncheckedArray[T]](addr data[i]) ②
```

① An operator for array element address computation, also known as pointer arithmetic.

② The address of the `i`-th array element is `addr data[i]` but it needs to be casted into the type `ptr UncheckedArray` as we will access the successive elements `data[i], data[i+1], data[i+2], …` with it.

45.1. parMap

A “for each” operation is also commonly known as a `map`. We call our parallel map `parMap`:

```
import malebolgia

template parMap[T](data: var openArray[T]; bulkSize: int; op: untyped) = ①
  proc worker(a: ptr UncheckedArray[T]; until: int) = ②
    for i in 0 ..< until: op a[i] ③

  var m = createMaster()
  m.awaitAll:
    var i = 0
    while i+bulkSize <= data.len: ④
      m.spawn worker(data@!i, bulkSize) ⑤
      i += bulkSize ⑥
    if i < data.len: ⑦
      m.spawn worker(data@!i, data.len-i) ⑧
```

① `parMap` takes an `openArray`, a `bulkSize` and the operation `op` to perform. The `bulkSize` is crucial to make the tasks big enough to amortize the overhead of sending the task to a different CPU.

② The worker operates on a `(pointer, length)` slice.

③ The worker applies `op` to every element of the slice.

④ As long as a slice of `bulkSize` exists…

⑤ …run `op` on `data[i ..< i+bulkSize]`.

⑥ Advance the run index `i` by `bulkSize` and proceed with the next slice.

⑦ The final slice might have fewer elements than `bulkSize` and needs to be special cased.

⑧ The final slice has length `data.len-i`.

For parallelization to be worthwhile, the input `data` array has to be of a sufficient length and the `bulkSize` must not be too small:

```nim
var testData: seq[int] = @[]
for i in 0 ..< 10_000: testData.add i ①

proc mul4(x: var int) = x *= 4 ②
parMap(testData, 600, mul4) ③

for i in 0 ..< 10_000: assert testData[i] == i*4 ④
```

① Creates test data, an array of 10_000 elements with the values 0, 1, 2, ...

② `mul4` takes a number and multiplies it by 4 and stores the result back into the `var` parameter. This is passed to the `parMap` `template` which mutates the array in place.

③ Calls `parMap` with the test data, a block size of 600 and the `mul4` routine. `mul4` could also have been declared as a `template` and `parMap` would accept it.

④ Tests that the `seq` was successfully mutated, the `i`-th element should have the value `i * 4`.

The power of structured concurrency combined with raw memory accesses and Nim’s template mechanism cannot be underestimated. Of course, these dangerous mechanisms should only be used behind the curtain of a safe abstraction, but that is `parMap`'s purpose. At the same time exposing the `bulkSize` is crucial for performance tweaking and should not be hidden. Providing a good default value for `bulkSize` is basically impossible as it depends on the cost of the `op` snippet that is run on every array element.

45.2. parReduce

There are few if any languages besides Nim that can express *implementations* of parMap and parReduce as concisely. parReduce can be implemented like this:

```
template parReduce[T](data: openArray[T]; bulkSize: int;
                      op: untyped): untyped = ①
  proc reduce[Tx](a: ptr UncheckedArray[Tx]; until: int): Tx = ②
    result = default(Tx) ③
    for i in 0 ..< until:
      op(result, a[i])   ④

  var m = createMaster()
  var res = newSeq[int](data.len div bulkSize + 1) ⑤
  var r = 0 ⑥
  m.awaitAll:
    var i = 0
    while i+bulkSize <= data.len: ⑦
      m.spawn reduce(data@!i, bulkSize) -> res[r] ⑧
      r += 1
      i += bulkSize
    if i < data.len: ⑨
      m.spawn reduce(data@!i, data.len-i) -> res[r] ⑩
      r += 1
  reduce(res@!0, r) ⑪
```

① parReduce takes an openArray, a bulkSize and which operation op to perform. The bulkSize is crucial to make the tasks big enough to amortize the overhead of sending the task to a different CPU.

② reduce is a helper proc that takes a slice and accumulates a result. Note that due to a current compiler limitation the inner generic type cannot be named T and so Tx was used.

③ Instead of 0 or 0.0 we use default(Tx) to keep it generic.

④ The actual reduction step. If op is += then op(result, a[i]) is transformed into +=(result, a[i]) which is the same as result += a[i].

⑤ For a parallel reduction we need a helper container that keeps the intermediate results. It is named res here. It is very important not to grow the seq after construction! An add might cause a reallocation of the seq which would be disastrous! The binding -> res[r] passes the address of res[r] to a worker thread and so it must be stable.

⑥ r is a helper variable that keeps the currently used length of res.

⑦ As long as a slice of `bulkSize` exists...

⑧ ...reduce `data[i ..< i+bulkSize]` via `op`.

⑨ The final slice might have fewer elements than `bulkSize` and needs to be special cased.

⑩ The final slice has length `data.len-i`.

⑪ All intermediate results need to be reduced after the spawned tasks have been completed. This reduction is also the final result that the `template` "returns".

Now we need to test our `parReduce` implementation:

```
var numbers: seq[int] = @[]
for i in 0 ..< 10_000: numbers.add i ①

let sum = parReduce(numbers, 600, `+=`) ②
assert sum == 49995000 ③
```

① Creates test data, an array of 10_000 elements with the values 0, 1, 2, ...

② Calls `parReduce` with the test data, a block size of 600 and the `+=` builtin operator for integers.

③ Ensures the `sum` has the correct value.

Notice how this example works even though `+=` is not a real `proc` but a builtin thanks to the substitution rules of a `template`.

45.3. parFind

Quite analogous to reduce, a search parFind can be written. The task is to return the minimal index of an element that fulfills some criterion or predicate. The helper proc that runs serially over the slice needs to know the offset so that later on the minimum of the results can be taken:

```
template parFind[T](data: openArray[T]; bulkSize: int;
                    predicate: untyped): int = ①
  proc linearFind[Tx](a: ptr UncheckedArray[Tx];
                      until, offset: int): int = ②
    for i in 0 ..< until:
      if predicate(a[i]): return i + offset ③
    return -1 ④

  var m = createMaster()
  var res = newSeq[int](data.len div bulkSize + 1) ⑤
  var r = 0 ⑥
  m.awaitAll:
    var i = 0
    while i+bulkSize <= data.len: ⑦
      m.spawn linearFind(data@!i, bulkSize, i) -> res[r] ⑧
      r += 1
      i += bulkSize
    if i < data.len: ⑨
      m.spawn linearFind(data@!i, data.len-i, i) -> res[r] ⑩
      r += 1
  var result = -1 ⑪
  for i in 0 ..< r:
    if res[i] >= 0: ⑫
      result = res[i]
      break
  result ⑬
```

① parFind takes an openArray, a bulkSize and the predicate to search for. It returns the smallest index of an element that fulfills predicate. It produces the value -1 if no such element exists.

② linearFind is a helper proc that takes a slice and performs a linear search. The offset parameter is used to adjust the index so that it refers to the real position within the openArray and not within the slice. Note that due to a current compiler limitation the inner generic type cannot be named T and so Tx was used.

③ Returns on a successful search.

④ For an unsuccessful search we return -1.

⑤ For a parallel search we need a helper container that keeps the intermediate results. It is named `res` here. It is very important not to grow the `seq` after construction! An `add` might cause a reallocation of the `seq` which would be disastrous! The binding `-> res[r]` passes the address of `res[r]` to a worker thread and so it must be stable.

⑥ `r` is a helper variable that keeps the currently used length of `res`.

⑦ As long as a slice of `bulkSize` exists...

⑧ ...search `data[i ..< i+bulkSize]`.

⑨ The final slice might have fewer elements than `bulkSize` and needs to be special cased.

⑩ The final slice has length `data.len-i`.

⑪ Keep in mind that a `template` does not have an implicitly declared `result` variable. So we need to declare one here ourselves.

⑫ Iterates over the intermediate results and stops as soon as a valid index was found.

⑬ The final value that is produced by the template is `result`.

This time we pass a *helper template* to `parFind` for our testing purposes:

```
var haystack: seq[int] = @[]
for i in 0 ..< 10_000: haystack.add i ①

template predicate(x): untyped = x == 1000 ②
let idx = parFind(haystack, 600, predicate) ③
assert idx == 1000 ④
```

① Creates test data, an array of 10_000 elements with the values 0, 1, 2, ...

② `predicate` takes the current array element and compares it to the number 1000. This is helper template that is then passed to `parFind`.

③ Calls `parFind` with the test data, a block size of 600 and the `predicate`.

④ Ensures that `idx` has the correct value.

Chapter 46. Final advice

46.1. What to avoid

1. Global variables. Global variables make your routines lack the `noSideEffect` effect that is required for using either `createThread` or `spawn` safely.
2. Don't use `createThread` unless you implement your own thread pool.
3. Don't use `ref`. Prefer `seq` as `seq` lacks the problematic aliasing aspects and can be moved around just as easily as a `ref`.
4. Avoid channels as these do not compose well with *potential* concurrency, where the runtime system is free *not to* exploit the concurrency and to run a task on the same thread instead.

46.2. What to use

Malebolgia's abstractions have been the result of years of research and cover most use cases:

1. Use `parMap`, `parFind`, and `parReduce` which work at a very high level and are easy to use correctly and hard to use incorrectly.
2. Use `spawn` and `awaitAll` or alternatives that enforce structured concurrency.
3. Share memory via a `Locker[T]` wrapper that ensures at compile time that data races cannot happen.

In some sense Malebolgia's abstractions are the most natural extensions that add concurrency to "single-threaded" Nim:

- Function call expressions can be run in parallel via `spawn`. Divergent control flow, as it exists in the single-threaded Nim in the form of `case` or `if` statements, eventually converges again via `awaitAll`.

- Spawned function calls can return values.
- Global variables or shared memory in the form of `var` parameters are enabled with `Locker[T]`.

Appendix A: Grammar

The grammar's start symbol is `module`.

```
# This file is generated by compiler/parser.nim.
module = complexOrSimpleStmt ^* (';' / IND{=})
comma = ',' COMMENT?
semicolon = ';' COMMENT?
colon = ':' COMMENT?
colcom = ':' COMMENT?
operator =  OP0 | OP1 | OP2 | OP3 | OP4 | OP5 | OP6 | OP7 | OP8 | OP9
         | 'or' | 'xor' | 'and'
         | 'is' | 'isnot' | 'in' | 'notin' | 'of' | 'as' | 'from'
         | 'div' | 'mod' | 'shl' | 'shr' | 'not' | '..'
prefixOperator = operator
optInd = COMMENT? IND?
optPar = (IND{>} | IND{=})?
simpleExpr = arrowExpr (OP0 optInd arrowExpr)* pragma?
arrowExpr = assignExpr (OP1 optInd assignExpr)*
assignExpr = orExpr (OP2 optInd orExpr)*
orExpr = andExpr (OP3 optInd andExpr)*
andExpr = cmpExpr (OP4 optInd cmpExpr)*
cmpExpr = sliceExpr (OP5 optInd sliceExpr)*
sliceExpr = ampExpr (OP6 optInd ampExpr)*
ampExpr = plusExpr (OP7 optInd plusExpr)*
plusExpr = mulExpr (OP8 optInd mulExpr)*
mulExpr = dollarExpr (OP9 optInd dollarExpr)*
dollarExpr = primary (OP10 optInd primary)*
operatorB = OP0 | OP1 | OP2 | OP3 | OP4 | OP5 | OP6 | OP7 | OP8 | OP9 |
            'div' | 'mod' | 'shl' | 'shr' | 'in' | 'notin' |
            'is' | 'isnot' | 'not' | 'of' | 'as' | 'from' | '..' |
            'and' | 'or' | 'xor'
symbol = '`' (KEYW|IDENT|literal|(operator|'('|')'|'['|']'|'{'|'}'|'=')+)+
'`'
       | IDENT | 'addr' | 'type' | 'static'
symbolOrKeyword = symbol | KEYW
exprColonEqExpr = expr ((':'|'=') expr
                       / doBlock extraPostExprBlock*)?
exprEqExpr = expr ('=' expr
```

```
                  / doBlock extraPostExprBlock*)?
exprList = expr ^+ comma
optionalExprList = expr ^* comma
exprColonEqExprList = exprColonEqExpr (comma exprColonEqExpr)* (comma)?
qualifiedIdent = symbol ('.' optInd symbolOrKeyword)?
setOrTableConstr = '{' ((exprColonEqExpr comma)* | ':' ) '}'
castExpr = 'cast' ('[' optInd typeDesc optPar ']' '(' optInd expr optPar
')') /
parKeyw = 'discard' | 'include' | 'if' | 'while' | 'case' | 'try'
        | 'finally' | 'except' | 'for' | 'block' | 'const' | 'let'
        | 'when' | 'var' | 'mixin'
par = '(' optInd
          ( &parKeyw (ifExpr / complexOrSimpleStmt) ^+ ';'
          | ';' (ifExpr / complexOrSimpleStmt) ^+ ';'
          | pragmaStmt
          | simpleExpr ( (doBlock extraPostExprBlock*)
                       | ('=' expr (';' (ifExpr / complexOrSimpleStmt) ^+
';' )? )
                       | (':' expr (',' exprColonEqExpr     ^+ ',' )? ) ) )
          optPar ')'
literal = | INT_LIT | INT8_LIT | INT16_LIT | INT32_LIT | INT64_LIT
          | UINT_LIT | UINT8_LIT | UINT16_LIT | UINT32_LIT | UINT64_LIT
          | FLOAT_LIT | FLOAT32_LIT | FLOAT64_LIT
          | STR_LIT | RSTR_LIT | TRIPLESTR_LIT
          | CHAR_LIT | CUSTOM_NUMERIC_LIT
          | NIL
generalizedLit = GENERALIZED_STR_LIT | GENERALIZED_TRIPLESTR_LIT
identOrLiteral = generalizedLit | symbol | literal
               | par | arrayConstr | setOrTableConstr | tupleConstr
               | castExpr
tupleConstr = '(' optInd (exprColonEqExpr comma?)* optPar ')'
arrayConstr = '[' optInd (exprColonEqExpr comma?)* optPar ']'
primarySuffix = '(' (exprColonEqExpr comma?)* ')'
      | '.' optInd symbolOrKeyword ('[:' exprList ']' ( '(' exprColonEqExpr
')' )?)? generalizedLit?
      | DOTLIKEOP optInd symbolOrKeyword generalizedLit?
      | '[' optInd exprColonEqExprList optPar ']'
      | '{' optInd exprColonEqExprList optPar '}'
pragma = '{.' optInd (exprColonEqExpr comma?)* optPar ('.}' | '}')
identVis = symbol OPR?  # postfix position
identVisDot = symbol '.' optInd symbolOrKeyword OPR?
identWithPragma = identVis pragma?
identWithPragmaDot = identVisDot pragma?
declColonEquals = identWithPragma (comma identWithPragma)* comma?
                  (':' optInd typeDescExpr)? ('=' optInd expr)?
identColonEquals = IDENT (comma IDENT)* comma?
     (':' optInd typeDescExpr)? ('=' optInd expr)?)
tupleTypeBracket = '[' optInd (identColonEquals (comma/semicolon)?)* optPar
']'
tupleType = 'tuple' tupleTypeBracket
tupleDecl = 'tuple' (tupleTypeBracket /
```

```
    COMMENT? (IND{>} identColonEquals (IND{=} identColonEquals)*)?)
paramList = '(' declColonEquals ^* (comma/semicolon) ')'
paramListArrow = paramList? ('->' optInd typeDesc)?
paramListColon = paramList? (':' optInd typeDesc)?
doBlock = 'do' paramListArrow pragma? colcom stmt
routineExpr = ('proc' | 'func' | 'iterator') paramListColon pragma? ('='
COMMENT? stmt)?
routineType = ('proc' | 'iterator') paramListColon pragma?
forStmt = 'for' ((varTuple / identWithPragma) ^+ comma) 'in' expr colcom
stmt
forExpr = forStmt
expr = (blockExpr
      | ifExpr
      | whenExpr
      | caseStmt
      | forExpr
      | tryExpr)
      / simpleExpr
simplePrimary = SIGILLIKEOP? identOrLiteral primarySuffix*
commandStart = &('`'|IDENT|literal|'cast'|'addr'|'type'|'var'|'out'|
                 'static'|'enum'|'tuple'|'object'|'proc')
primary = simplePrimary (commandStart expr (doBlock extraPostExprBlock*)?)?
        / operatorB primary
        / routineExpr
        / rawTypeDesc
        / prefixOperator primary
rawTypeDesc = (tupleType | routineType | 'enum' | 'object' |
                ('var' | 'out' | 'ref' | 'ptr' | 'distinct') typeDesc?)
                ('not' primary)?
typeDescExpr = (routineType / simpleExpr) ('not' primary)?
typeDesc = rawTypeDesc / typeDescExpr
typeDefValue = ((tupleDecl | enumDecl | objectDecl | conceptDecl |
                 ('ref' | 'ptr' | 'distinct') (tupleDecl | objectDecl))
               / (simpleExpr (exprEqExpr ^+ comma postExprBlocks?)?))
               ('not' primary)?
extraPostExprBlock = ( IND{=} doBlock
                     | IND{=} 'of' exprList ':' stmt
                     | IND{=} 'elif' expr ':' stmt
                     | IND{=} 'except' optionalExprList ':' stmt
                     | IND{=} 'finally' ':' stmt
                     | IND{=} 'else' ':' stmt )
postExprBlocks = (doBlock / ':' (extraPostExprBlock / stmt))
extraPostExprBlock*
exprStmt = simpleExpr postExprBlocks?
         / simplePrimary (exprEqExpr ^+ comma) postExprBlocks?
         / simpleExpr '=' optInd (expr postExprBlocks?)
importStmt = 'import' optInd expr
              ((comma expr)*
              / 'except' optInd (expr ^+ comma))
exportStmt = 'export' optInd expr
              ((comma expr)*
```

```
              / 'except' optInd (expr ^+ comma))
includeStmt = 'include' optInd expr ^+ comma
fromStmt = 'from' expr 'import' optInd expr (comma expr)*
returnStmt = 'return' optInd expr?
raiseStmt = 'raise' optInd expr?
yieldStmt = 'yield' optInd expr?
discardStmt = 'discard' optInd expr?
breakStmt = 'break' optInd expr?
continueStmt = 'continue' optInd expr?
condStmt = expr colcom stmt COMMENT?
           (IND{=} 'elif' expr colcom stmt)*
           (IND{=} 'else' colcom stmt)?
ifStmt = 'if' condStmt
whenStmt = 'when' condStmt
condExpr = expr colcom stmt optInd
        ('elif' expr colcom stmt optInd)*
         'else' colcom stmt
ifExpr = 'if' condExpr
whenExpr = 'when' condExpr
whileStmt = 'while' expr colcom stmt
ofBranch = 'of' exprList colcom stmt
ofBranches = ofBranch (IND{=} ofBranch)*
                      (IND{=} 'elif' expr colcom stmt)*
                      (IND{=} 'else' colcom stmt)?
caseStmt = 'case' expr ':'? COMMENT?
            (IND{>} ofBranches DED
            | IND{=} ofBranches)
tryStmt = 'try' colcom stmt &(IND{=}? 'except'|'finally')
           (IND{=}? 'except' optionalExprList colcom stmt)*
           (IND{=}? 'finally' colcom stmt)?
tryExpr = 'try' colcom stmt &(optInd 'except'|'finally')
           (optInd 'except' optionalExprList colcom stmt)*
           (optInd 'finally' colcom stmt)?
blockStmt = 'block' symbol? colcom stmt
blockExpr = 'block' symbol? colcom stmt
staticStmt = 'static' colcom stmt
deferStmt = 'defer' colcom stmt
asmStmt = 'asm' pragma? (STR_LIT | RSTR_LIT | TRIPLESTR_LIT)
genericParam = symbol (comma symbol)* (colon expr)? ('=' optInd expr)?
genericParamList = '[' optInd
  genericParam ^* (comma/semicolon) optPar ']'
pattern = '{' stmt '}'
indAndComment = (IND{>} COMMENT)? | COMMENT?
routine = optInd identVis pattern? genericParamList?
  paramListColon pragma? ('=' COMMENT? stmt)? indAndComment
commentStmt = COMMENT
section(RULE) = COMMENT? RULE / (IND{>} (RULE / COMMENT)^+IND{=} DED)
enumDecl = 'enum' optInd (symbol pragma? optInd ('=' optInd expr COMMENT?)?
comma?)+
objectWhen = 'when' expr colcom objectPart COMMENT?
            ('elif' expr colcom objectPart COMMENT?)*
```

```
            ('else' colcom objectPart COMMENT?)?
objectBranch = 'of' exprList colcom objectPart
objectBranches = objectBranch (IND{=} objectBranch)*
                      (IND{=} 'elif' expr colcom objectPart)*
                      (IND{=} 'else' colcom objectPart)?
objectCase = 'case' declColonEquals ':'? COMMENT?
            (IND{>} objectBranches DED
            | IND{=} objectBranches)
objectPart = IND{>} objectPart^+IND{=} DED
           / objectWhen / objectCase / 'nil' / 'discard' / declColonEquals
objectDecl = 'object' ('of' typeDesc)? COMMENT? objectPart
conceptParam = ('var' | 'out')? symbol
conceptDecl = 'concept' conceptParam ^* ',' (pragma)? ('of' typeDesc ^*
',')?
              &IND{>} stmt
typeDef = identVisDot genericParamList? pragma '=' optInd typeDefValue
            indAndComment?
varTupleLhs = '(' optInd (identWithPragma / varTupleLhs) ^+ comma optPar ')'
varTuple = varTupleLhs '=' optInd expr
colonBody = colcom stmt postExprBlocks?
variable = (varTuple / identColonEquals) colonBody? indAndComment
constant = (varTuple / identWithPragma) (colon typeDesc)? '=' optInd expr
indAndComment
bindStmt = 'bind' optInd qualifiedIdent ^+ comma
mixinStmt = 'mixin' optInd qualifiedIdent ^+ comma
pragmaStmt = pragma (':' COMMENT? stmt)?
simpleStmt = ((returnStmt | raiseStmt | yieldStmt | discardStmt | breakStmt
           | continueStmt | pragmaStmt | importStmt | exportStmt | fromStmt
           | includeStmt | commentStmt) / exprStmt) COMMENT?
complexOrSimpleStmt = (ifStmt | whenStmt | whileStmt
                    | tryStmt | forStmt
                    | blockStmt | staticStmt | deferStmt | asmStmt
                    | 'proc' routine
                    | 'method' routine
                    | 'func' routine
                    | 'iterator' routine
                    | 'macro' routine
                    | 'template' routine
                    | 'converter' routine
                    | 'type' section(typeDef)
                    | 'const' section(constant)
                    | ('let' | 'var' | 'using') section(variable)
                    | bindStmt | mixinStmt)
                    / simpleStmt
stmt = (IND{>} complexOrSimpleStmt^+(IND{=} / ';') DED)
     / simpleStmt ^+ ';'
```

Appendix B: Nim standard library cheat sheet

B.1. Integers

Integer functionality is available via the automatically imported `system` module.

Table 11. Operations on integers

Operation	Example
`div`: Integer division (without a remainder).	`13 div 5 == 2`
`mod`: Integer modulo (remainder).	`13 mod 5 == 3`
`shl`: Shift left.	`1 shl 3 == 8`
`shr`: Shift right.	`8 shr 1 == 4`
`and`: Bitwise and.	`0b0011 and 0b0101 == 0b0001`
`or`: Bitwise or.	`0b0011 or 0b0101 == 0b0111`
`xor`: Bitwise xor.	`0b0011 xor 0b0101 == 0b0110`
`toInt`: Converts a floating point number into an integer.	`toInt(2.49) == 2` `toInt(2.5) == 3`

`inc`: Increments the value of an ordinal variable.	`var x = 5`
	`inc x`
	`assert x == 6`
	`inc x, 3`
	`assert x == 9`

`dec`: Decrements the value of an ordinal variable.	`var x = 5`
	`dec x`
	`assert x == 4`
	`dec x, 3`
	`assert x == 1`

B.2. Strings

To use some of the available functionality on strings, you need to `import std/strutils`.

Table 12. String-related functionality

Operation	Example
`$`: Converts a type into a string.	`$123 == "123"`
`add`: Add a character to a string.	`"abc".add 'd' == "abcd"`
`&`: Concatenation of two strings.	`"ab" & "cd" == "abcd"`
`join`: Concatenation with a string between each element.	`["ab", "cd", "ef"].join("-x-") == "ab-x-cd-x-ef"`
`split`: Splits a string on whitespace characters.	`split("ab cd") == @["ab", "cd"]`
`split`: Splits a string on a given character.	`"abxcd".split('x') == @["ab", "cd"]`
`find`: Searches for a character inside of a string.	`"abcd".find('c') == 2`
`find`: Searches for a substring inside of a string.	`"abcd".find("bc") == 1`
`replace`: Replaces every occurrence of a given character with a new one.	`"acdc".replace('c', 'x') == "axdx"`

B.3. Sequences

To use some of the available functionality on sequences, you need to `import std/sequtils`.

Table 13. Seq-related functionality

Operation	Example
`toSeq`: Converts an iterable into a sequence.	`toSeq(1..3) == @[1, 2, 3]`
`@`: Converts arrays and strings to a sequence.	`@"abc" == @['a', 'b', 'c']`
`&`: Concatenation of two sequences.	`@[1, 2] & @[3, 4] == @[1, 2, 3, 4]`

`map`: Applies a proc to every item in a sequence.	`proc double(x: int): int =`
	`  2*x`
	`var a = @[1, 3, 5]`
	`var b = a.map(double)`
	`assert b == @[2, 6, 10]`

`filter`: Returns a new sequence with values that satisfy a predicate.	`proc small(x: int): bool =`
	`  x < 4`
	`var a = @[1, 3, 5]`
	`var b = a.filter(small)`
	`assert b == @[1, 3]`

B.4. Bit sets

Bit sets are built-in, i.e. available via the automatically imported `system` module.

Table 14. Bit sets operations

Operation	Example

`incl`: Include an element in the set.	`var a = {5 .. 8}`
	`a.incl 3`
	`assert a == {3, 5, 6, 7, 8}`

`excl`: Exclude an element from the set.	`var a = {5 .. 8}`
	`a.excl 5`
	`assert a == {6, 7, 8}`

`*`: Intersection between two sets.	`assert {1, 2} * {2, 3} == {2}`
`+`: Union between two sets.	`assert {1, 2} * {2, 3} == {1, 2, 3}`
`-`: Difference between two sets.	`assert {1, 2} - {2, 3} == {1}`
`<`: A proper subset.	`assert {1, 2} < {1, 2, 3}` `assert not({1, 2} < {1, 2})`

B.5. Hashes

To implement hashing for custom types, one can import the `std/hashes` module.

Table 15. Hashing operations

Operation	Example

`!&`: Mixing a hash value.	`var h: Hash = 0`
	`for element in mySeq:`
	`  h = h !& hash(element)`

`!$`: Finishing the hash value.	`var h: Hash = 0`
	`for element in mySeq:`
	`  h = h !& hash(element)`
	`h = !$h`

B.6. Hash sets

Hash sets are available with `import std/sets`.

Table 16. Hash sets operations

Operation	Example
`toHashSet`: Converts a collection to a hash set.	`assert toHashSet("acdc") == ['a', 'c', 'd'].toHashSet`

`*`, `intersection`: Intersection between two sets.	`var a = [1, 2].toHashSet`
	`var b = [2, 3].toHashSet`
	`assert a * b == [2].toHashSet`

`+`, `union`: Union between two sets.	`var a = [1, 2].toHashSet`
	`var b = [2, 3].toHashSet`
	`assert a + b == [1, 2, 3].toHashSet`

`-`, `difference`: Difference between two sets.	`var a = [1, 2].toHashSet`
	`var b = [2, 3].toHashSet`
	`assert a - b == [1].toHashSet`
	`assert b - a == [3].toHashSet`

`<`: A proper subset.	`var a = [1, 2].toHashSet`
	`var c = [1, 2, 3].toHashSet`
	`assert a < c`
	`assert not(a < a)`

`<=`: A subset.	`var a = [1, 2].toHashSet`
	`var c = [1, 2, 3].toHashSet`
	`assert a <= c`
	`assert a <= a`

`card`, `len`: Number of elements in a set.	`assert card([1, 2].toHashSet) == 2`

`pop`: Removes and returns a random element from a set.	`var s = [1, 2, 3].toHashSet`
	`let x = s.pop`
	`assert card(s) == 2`

Operation	Example

`containsOrIncl`: Adds an element to the set, and returns `true` if the key already existed.	`var s = [1, 2].toHashSet`
	`assert s.containsOrIncl(1)`
	`assert not s.containsOrIncl(9)`
	`assert s == [1, 2, 9].toHashSet`

B.7. Hash tables

Hash tables are available with `import std/tables`.

Table 17. Tables-related functionality

Operation	Example
`initTable`: Initializes an empty hash table.	`var a = initTable[int, string]()`
`toTable`: Creates a table from a container of pairs.	`var a = toTable([(5, "ab"), (7, "cd")])`
`[]=`: Inserts a key-value pair into a table.	`var a = toTable([(5, "ab"), (7, "cd")])` `a[9] = "ef"`

`[]`: Retrieves a value from a given key. Raises an exception if a key doesn't exist.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`echo a[5] # => "ab"`
	`echo a[9] # => raises KeyError`

`getOrDefault`: Retrieves a value from a given key. Returns a default (or provided) value if a key doesn't exist.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`echo a.getOrDefault(5) # => "ab"`
	`echo a.getOrDefault(9) # => ""`
	`echo a.getOrDefault(5, "ef") # => "ab"`
	`echo a.getOrDefault(9, "ef") # => "ef"`

`hasKey`: Checks if a given key is in the table.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`assert a.hasKey(5)`
	`assert not a.hasKey(9)`

`hasKeyOrPut`: Returns `true` if a given key is in the table. Otherwise inserts a value.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`if a.hasKeyOrPut(7, "ef"):`
	`  a[5].add 'z'`
	`if a.hasKeyOrPut(9, "gh"):`
	`  a[5].add 'y'`
	`assert a == {5: "abz", 7: "cd", 9: "gh"}.toTable`

`mgetOrPut`: Gets a value of a given key, or puts a new value if the key doesn't exist. Returns a modifiable value.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`a.mgetOrPut(5, "xy").add 'z'`
	`a.mgetOrPut(9, "ef").add 'y'`
	`assert a == {5: "abz", 7: "cd", 9: "efy"}.toTable`

Operation	Example

`del`: Deletes a key from the table. Does nothing if the key is not in the table.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`a.del(5)`
	`a.del(9)`
	
	`assert a == {7: "cd"}.toTable`

`pop`: Deletes a key from the table. Returns `true` if the key existed, and sets the given variable to the value of the key. Returns `false` if the key didn't exist, and the variable is unchanged.	`var a = toTable([(5, "ab"), (7, "cd")])`
	`var s = ""`
	`assert a.pop(5, s)`
	`assert s == "ab"`
	`s = ""`
	`assert not a.pop(9, s)`
	`assert s == ""`

B.8. Optionals

Types which encapsulate optional value are available with `import std/options`.

Table 18. Optionals-related functionality

Operation	Example
`Option[T]`: Type of an optional variable.	`var a: Option[int]`
`some`: Returns a value of an `Option`.	`var a: Option[int]` `a = some(31)`
`none`: Returns an `Option` that has no value.	`var a: Option[int]` `a = none(int)`
`isSome`: Checks if an `Option` contains a value.	`assert some(31).isSome` `assert not none(int).isSome`
`isNone`: Checks if an `Option` is empty.	`assert not some(31).isNone` `assert none(int).isNone`
`get`: Return a value of an `Option` or a default value if there is no value.	`assert some(31).get(-1) == 31` `assert none(int).get(-1) == -1`

`filter`: Applies a function to the value of an `Option`.	`proc isOdd(x: int): bool =`
	`  x mod 2 == 1`
	
	`assert some(31).filter(isOdd) == some(31)`
	`assert some(32).filter(isOdd) == none(int)`
	`assert none(int).filter(isOdd) == none(int)`

`map`: Applies a function to the value of an `Option` and returns a new `Option`.	`proc isOdd(x: int): bool =`
	`  x mod 2 == 1`
	
	`assert some(31).map(isOdd) == some(true)`
	`assert some(32).map(isOdd) == none(bool)`
	`assert none(int).map(isOdd) == none(bool)`

B.9. String formatting

String formatting and interpolation is available with `import std/strformat`.

One can use either `fmt` or `&` for formatting. Note that the string in `fmt"{expr}"` is a generalized raw string literal, i.e. `\n` will *not* be interpreted as a newline (the `\` will be escaped). The `&` will interpret `\n` as a new line:

```
import std/strformat

let msg = "hello"
assert fmt"{msg}\n" == "hello\\n"
assert &"{msg}\n" == "hello\n"
```

Table 19. String format functionality

Operation	Example

`<`, `^`, `>`: Left (default for strings), center, right (default for numbers) alignment.	`let s = "nim"`
	`let x = 987.12`
	
	`assert fmt"{s:5}" == "nim  "`
	`assert fmt"{s:<5}" == "nim  "`
	`assert fmt"{s:>5}" == "  nim"`
	`assert fmt"{s:^5}" == " nim "`
	
	`assert fmt"{x:8.2f}" == "  987.12"`
	`assert fmt"{x:8.4f}" == "987.1200"`
	`assert fmt"{x:<8.1f}" == "987.1   "`

`fmt"{expr=}"`: This expands to `fmt"expr={expr}"`, which is useful for debugging.	`let s = "nim"`
	
	`assert fmt"{s=}" == "s=nim"`
	`assert fmt"{s = }" == "s = nim"`

B.10. Algorithms

Some common algorithms on arrays and sequences are available via `import std/algorithm`.

Table 20. Algorithm functionality

Operation	Example
`binarySearch`: Assumes the container is sorted and binary searches for an element.	`var a = [50, 60, 70, 80]` `assert a.binarySearch(70) == 2`

`fill`: Fills a slice of a container with a value. If no range is specified, it assigns a value to all elements.	`var a: array[5, int]`
	`a.fill(2, 4, 99)`
	`assert a == [0, 0, 99, 99, 99]`
	`a.fill(88)`
	`assert a == [88, 88, 88, 88, 88]`

`reverse`: Reverses a slice of a container. If no range is specified, it reverses the whole container.	`var a = [10, 20, 30, 40, 50, 60]`
	`a.reverse(1, 3)`
	`assert a == [10, 40, 30, 20, 50, 60]`
	`a.reverse()`
	`assert a == [60, 50, 20, 30, 40, 10]`

`nextPermutation`: Modifies a container, changing it to the next lexicographic permutation. Returns `true` if a permutation happened (the last-ordered permutation was not reached).	`var a = [10, 20, 30, 40]`
	`assert a.nextPermutation() == true`
	`assert a == [10, 20, 40, 30]`
	
	`a = [40, 30, 20, 10]`
	`assert a.nextPermutation() == false`
	`assert a == [40, 30, 20, 10]`

`prevPermutation`: Modifies a container, changing it to the previous lexicographic permutation. Returns `true` if a permutation happened (the first-ordered permutation was not reached).	`var a = [10, 20, 30, 40]`
	`assert a.prevPermutation() == false`
	`assert a == [10, 20, 30, 40]`
	
	`a = [40, 30, 10, 20]`
	`assert a.prevPermutation() == true`
	`assert a == [40, 20, 30, 10]`

Operation	Example

`product`: Cartesian product.	`var a = @[10, 20, 30]`
	`var b = @[99, 88]`
	
	`assert product([a, b]) == @[`
	`  @[30, 88], @[20, 88], @[10, 88],`
	`  @[30, 99], @[20, 99], @[10, 99],`
	`]`

`rotateLeft`, `rotatedLeft`: Left rotation of a container. For right rotation use negative distance. `rotateLeft` is an in-place version of `rotatedLeft`.	`var a = [10, 20, 30, 40, 50]`
	`a.rotateLeft(1)`
	`assert a == [20, 30, 40, 50, 10]`
	
	`assert a.rotatedLeft(-2) == @[50, 10, 20, 30, 40]`

`sort`, `sorted`: Merge sort of a container. `sort` is an in-place version of `sorted`.	`var a = [20, 40, 50, 10, 30]`
	
	`assert sorted(a) == @[10, 20, 30, 40, 50]`
	
	`a.sort(Descending)`
	`assert a == [50, 40, 30, 20, 10]`

B.11. OS

Basic operating system facilities are available with `import std/os`.

Table 21. OS functionality

Operation	Example
`/`, `joinPath`: Joins two directory names to one.	`"foo" / "bar" == "foo/bar"`
`addFileExt`: Adds an extension to a filename without one.	`addFileExt("foo", "bar") == "foo.bar"` `addFileExt("foo.exe", "bar") == "foo.exe"`

`changeFileExt`: Changes an extension of a filename. Pass `""` to remove an existing extension.	`changeFileExt("foo", "bar") == "foo.bar"`
	`changeFileExt("foo.exe", "bar") == "foo.bar"`
	`changeFileExt("foo.exe", "") == "foo"`

`execShellCmd`: Executes a shell command and returns its error code.	`assert execShellCmd("ls -la") == 0`
`extractFilename`: Extracts the filename of a given path.	`extractFilename("foo/bar/baz.exe") == "baz.exe"` `extractFilename("foo/bar/") == ""`
`parentDir`: Returns the parent directory of a path.	`parentDir("foo/bar/baz.exe") == "foo/bar"` `parentDir("foo/bar/") == "foo"`
`splitFile`: Splits a filename into a tuple containing directory, filename, and extension.	`splitFile("foo/bar/baz.exe") == ("foo/bar", "baz", ".exe")` `splitFile("foo/bar/") == ("foo/bar", "", "")`

<table>
<tr><th>Operation</th><th>Example</th></tr>
<tr><td><code>paramCount</code>: Returns the number of command line arguments given to the application.</td><td>myfile.nim
<pre>import os

echo paramCount()</pre>
<pre>> ./myfile
0

> ./myfile foo bar
2</pre></td></tr>
<tr><td><code>paramStr</code>: Returns n-th command line argument given to the application</td><td>myfile.nim
<pre>import os

echo paramStr(1)</pre>
<pre>> ./myfile foo bar
foo</pre></td></tr>
</table>

B.12. JSON

Basic JSON support is available via `import std/json`.

Table 22. JSON functionality

Operation	Example

`parseJson`: Creates a JSON tree from a string.	`let jsonNode = parseJson("""{"key": 3.14}""")`
	`assert jsonNode.kind == JObject`
	`assert jsonNode["key"].kind == JFloat`

`pretty`: Produces a pretty string representation for the provided JSON tree.	`let a = parseJson("""{"key": 3.14}""")` `echo pretty(a)`
`getInt`: Retrieves the int value of a `JInt JsonNode`.	`var a = %60` `assert a.getInt == 60`
`getFloat`: Retrieves the float value of a `JFloat JsonNode`.	`var a = %60.0` `assert a.getFloat == 60.0`
`getStr`: Retrieves the string value of a `JString JsonNode`.	`var a = %"abc"` `assert a.getStr == "abc"`
`getBool`: Retrieves the bool value of a `JBool JsonNode`.	`var a = %true` `assert a.getBool == true`

`%`: Generic constructor for JSON data. Can construct atoms and composed arrays and objects.	`let s = %"abc"`
	`let i = %5`
	`let f = %5.5`
	`let b = %false`
	`let a = %[%5, %5.0, %"x", %true,`
	`          %{"key": %8, "keyB": %9}]`
	
	`assert $a ==`
	`  """[5,5.0,"x",true,{"key":8,"keyB":9}]"""`

`%*`: Generic constructor for JSON data and does so recursively. Can construct atoms and composed arrays and objects.	`let a = %*[5, 5.0, "x", true,`
	`           {"key": 8, "keyB": 9}]`
	
	`assert $a ==`
	`  """[5,5.0,"x",true,{"key":8,"keyB":9}]"""`

B.13. Unicode

Basic Unicode support is available via `import std/unicode`. A Unicode code point is called a `Rune`.

Table 23. Unicode functionality

Operation	Example

`runeLen`: Returns the number of runes of a string.	`let a = "añyóng"`
	`assert a.runeLen == 6`
	`assert a.len == 8`

`runeAt`: Returns the rune of the given string at the given *byte index*.	`let a = "añyóng"`
	`assert a.runeAt(1) == "ñ".runeAt(0)`
	`assert a.runeAt(2) == "ñ".runeAt(1)`
	`assert a.runeAt(3) == "y".runeAt(0)`

`validateUtf8`: Returns the position of the first invalid byte that does not hold valid UTF-8 data. If every byte is valid `-1` is returned.	`assert validateUtf8("añyóng") == -1`
`runes`: Iterates over any rune of a string.	`for r in runes("añyóng"):` `  echo r == Rune('g')`
`cmpRunesIgnoreCase`: Compares two UTF-8 strings and ignores the case. Returns: `0` if `a == b` and a value `< 0` if `a < b` and a value `> 0`` if `a > b`.	`assert cmpRunesIgnoreCase("añyóng",` `"anyong") > 0`
`toUTF8`, `$`: Converts a rune into its UTF-8 representation. `$` is an alias for `toUTF8`.	`assert toUTF8("añyóng".runeAt(1)) == "ñ"`

Index

C

D

N

O

P

Q

R

S

T