# Nimrod Tutorial (Part I) 0.9.2

Andreas Rumpf

May 21, 2013

## Contents

# 1 Introduction

This document is a tutorial for the programming language *Nimrod*. This tutorial assumes that you are familiar with basic programming concepts like variables, types or statements but is kept very basic. The manual contains many more examples of the advanced language features.

# 2 The first program

We start the tour with a modified "hello world" program:

```
# This is a comment
echo("What's your name? ")
var name: string = readLine(stdin)
echo("Hi, ", name, "!")
```

Save this code to the file "greetings.nim". Now compile and run it:

```
nimrod compile --run greetings.nim
```

With the `-run` switch Nimrod executes the file automatically after compilation. You can give your program command line arguments by appending them after the filename:

```
nimrod compile --run greetings.nim arg1 arg2
```

Commonly used commands and switches have abbreviations, so you can also use:

```
nimrod c -r greetings.nim
```

To compile a release version use:

```
nimrod c -d:release greetings.nim
```

By default the Nimrod compiler generates a large amount of runtime checks aiming for your debugging pleasure. With `-d:release` these checks are turned off and optimizations are turned on.

Though it should be pretty obvious what the program does, I will explain the syntax: statements which are not indented are executed when the program starts. Indentation is Nimrod's way of grouping statements. Indentation is done with spaces only, tabulators are not allowed.

String literals are enclosed in double quotes. The `var` statement declares a new variable named `name` of type `string` with the value that is returned by the `readLine` procedure. Since the compiler knows that `readLine` returns a string, you can leave out the type in the declaration (this is called local type inference). So this will work too:

```
var name = readLine(stdin)
```

Note that this is basically the only form of type inference that exists in Nimrod: it is a good compromise between brevity and readability.

The "hello world" program contains several identifiers that are already known to the compiler: `echo`, `readLine`, etc. These built-ins are declared in the system module which is implicitly imported by any other module.

# 3 Lexical elements

Let us look at Nimrod's lexical elements in more detail: like other programming languages Nimrod consists of (string) literals, identifiers, keywords, comments, operators, and other punctuation marks.

### 3.1 String and character literals

String literals are enclosed in double quotes; character literals in single quotes. Special characters are escaped with \: \n means newline, \t means tabulator, etc. There are also *raw* string literals:

```
r"C:\program files\nim"
```

In raw literals the backslash is not an escape character.

The third and last way to write string literals are *long string literals*. They are written with three quotes: """ ... """; they can span over multiple lines and the \ is not an escape character either. They are very useful for embedding HTML code templates for example.

### 3.2 Comments

Comments start anywhere outside a string or character literal with the hash character #. Documentation comments start with ##. Multiline comments need to be aligned at the same column:

```
i = 0     # This is a single comment over multiple lines belonging to the
          # assignment statement.
# This is a new comment belonging to the current block, but to no particular
# statement.
i = i + 1 # This a new comment that is NOT
echo(i)   # continued here, because this comment refers to the echo statement
```

The alignment requirement does not hold if the preceding comment piece ends in a backslash:

```
type
  TMyObject {.final, pure, acyclic.} = object  # comment continues: \
    # we have lots of space here to comment 'TMyObject'.
    # This line belongs to the comment as it's properly aligned.
```

Comments are tokens; they are only allowed at certain places in the input file as they belong to the syntax tree! This feature enables perfect source-to-source transformations (such as pretty-printing) and simpler documentation generators. A nice side-effect is that the human reader of the code always knows exactly which code snippet the comment refers to. Since comments are a proper part of the syntax, watch their indentation:

```
echo("Hello!")
# comment has the same indentation as above statement -> fine
echo("Hi!")
  # comment has not the correct indentation level -> syntax error!
```

**Note**: To comment out a large piece of code, it is often better to use a when false: statement.

### 3.3 Numbers

Numerical literals are written as in most other languages. As a special twist, underscores are allowed for better readability: 1_000_000 (one million). A number that contains a dot (or 'e' or 'E') is a floating point literal: 1.0e9 (one million). Hexadecimal literals are prefixed with 0x, binary literals with 0b and octal literals with 0o. A leading zero alone does not produce an octal.

## 4 The var statement

The var statement declares a new local or global variable:

```
var x, y: int # declares x and y to have the type ``int``
```

Indentation can be used after the var keyword to list a whole section of variables:

```
var
  x, y: int
  # a comment can occur here too
  a, b, c: string
```

# 5 The assignment statement

The assignment statement assigns a new value to a variable or more generally to a storage location:

```
var x = "abc" # introduces a new variable `x` and assigns a value to it
x = "xyz"     # assigns a new value to `x`
```

= is the *assignment operator*. The assignment operator cannot be overloaded, overwritten or forbidden, but this might change in a future version of Nimrod.

# 6 Constants

Constants are symbols which are bound to a value. The constant's value cannot change. The compiler must be able to evaluate the expression in a constant declaration at compile time:

```
const x = "abc" # the constant x contains the string "abc"
```

Indentation can be used after the `const` keyword to list a whole section of constants:

```
const
  x = 1
  # a comment can occur here too
  y = 2
  z = y + 5 # computations are possible
```

# 7 The let statement

The `let` statement works like the `var` statement but the declared symbols are *single assignment* variables: After the initialization their value cannot change:

```
let x = "abc" # introduces a new variable `x` and binds a value to it
x = "xyz"     # Illegal: assignment to `x`
```

The difference between `let` and `const` is: `let` introduces a variable that can not be re-assigned, `const` means "enforce compile time evaluation and put it into a data section":

```
const input = readline(stdin) # Error: constant expression expected

let input = readline(stdin)   # works
```

# 8 Control flow statements

The greetings program consists of 3 statements that are executed sequentially. Only the most primitive programs can get away with that: branching and looping are needed too.

## 8.1 If statement

The if statement is one way to branch the control flow:

```
let name = readLine(stdin)
if name == "":
  echo("Poor soul, you lost your name?")
elif name == "name":
  echo("Very funny, your name is name.")
else:
  echo("Hi, ", name, "!")
```

There can be zero or more elif parts, and the else part is optional. The keyword `elif` is short for `else if`, and is useful to avoid excessive indentation. (The `""` is the empty string. It contains no characters.)

## 8.2 Case statement

Another way to branch is provided by the case statement. A case statement is a multi-branch:

```
let name = readLine(stdin)
case name
of "":
  echo("Poor soul, you lost your name?")
of "name":
  echo("Very funny, your name is name.")
of "Dave", "Frank":
  echo("Cool name!")
else:
  echo("Hi, ", name, "!")
```

As it can be seen, for an `of` branch a comma separated list of values is also allowed.

The case statement can deal with integers, other ordinal types and strings. (What an ordinal type is will be explained soon.) For integers or other ordinal types value ranges are also possible:

```
# this statement will be explained later:
from strutils import parseInt

echo("A number please: ")
let n = parseInt(readLine(stdin))
case n
of 0..2, 4..7: echo("The number is in the set: {0, 1, 2, 4, 5, 6, 7}")
of 3, 8: echo("The number is 3 or 8")
```

However, the above code does not compile: the reason is that you have to cover every value that `n` may contain, but the code only handles the values `0..8`. Since it is not very practical to list every other possible integer (though it is possible thanks to the range notation), we fix this by telling the compiler that for every other value nothing should be done:

```
...
case n
of 0..2, 4..7: echo("The number is in the set: {0, 1, 2, 4, 5, 6, 7}")
of 3, 8: echo("The number is 3 or 8")
else: nil
```

The `nil` statement is a *do nothing* statement. The compiler knows that a case statement with an else part cannot fail and thus the error disappears. Note that it is impossible to cover all possible string values: that is why there is no such check for string cases.

In general the case statement is used for subrange types or enumerations where it is of great help that the compiler checks that you covered any possible value.

## 8.3 While statement

The while statement is a simple looping construct:

```
echo("What's your name? ")
var name = readLine(stdin)
while name == "":
  echo("Please tell me your name: ")
  name = readLine(stdin)
  # no ``var``, because we do not declare a new variable here
```

The example uses a while loop to keep asking the user for his name, as long as he types in nothing (only presses RETURN).

## 8.4 For statement

The for statement is a construct to loop over any element an *iterator* provides. The example uses the built-in `countup` iterator:

```
echo("Counting to ten: ")
for i in countup(1, 10):
  echo($i)
# --> Outputs 1 2 3 4 5 6 7 8 9 10 on different lines
```

The built-in `$` operator turns an integer (`int`) and many other types into a string. The variable `i` is implicitly declared by the `for` loop and has the type `int`, because that is what `countup` returns. `i` runs through the values 1, 2, .., 10. Each value is `echo`-ed. This code does the same:

```
echo("Counting to 10: ")
var i = 1
while i <= 10:
  echo($i)
  inc(i) # increment i by 1
# --> Outputs 1 2 3 4 5 6 7 8 9 10 on different lines
```

Counting down can be achieved as easily (but is less often needed):

```
echo("Counting down from 10 to 1: ")
for i in countdown(10, 1):
  echo($i)
# --> Outputs 10 9 8 7 6 5 4 3 2 1 on different lines
```

Since counting up occurs so often in programs, Nimrod also has a `..` iterator that does the same:

```
for i in 1..10:
  ...
```

## 8.5 Scopes and the block statement

Control flow statements have a feature not covered yet: they open a new scope. This means that in the following example, `x` is not accessible outside the loop:

```
while false:
  var x = "hi"
echo(x) # does not work
```

A while (for) statement introduces an implicit block. Identifiers are only visible within the block they have been declared. The `block` statement can be used to open a new block explicitly:

```
block myblock:
  var x = "hi"
echo(x) # does not work either
```

The block's *label* (`myblock` in the example) is optional.

## 8.6 Break statement

A block can be left prematurely with a `break` statement. The break statement can leave a `while`, `for`, or a `block` statement. It leaves the innermost construct, unless a label of a block is given:

```
block myblock:
  echo("entering block")
  while true:
    echo("looping")
    break # leaves the loop, but not the block
  echo("still in block")

block myblock2:
  echo("entering block")
  while true:
    echo("looping")
    break myblock2 # leaves the block (and the loop)
  echo("still in block")
```

### 8.7 Continue statement

Like in many other programming languages, a `continue` statement starts the next iteration immediately:

```
while true:
  let x = readLine(stdin)
  if x == "": continue
  echo(x)
```

### 8.8 When statement

Example:

```
when system.hostOS == "windows":
  echo("running on Windows!")
elif system.hostOS == "linux":
  echo("running on Linux!")
elif system.hostOS == "macosx":
  echo("running on Mac OS X!")
else:
  echo("unknown operating system")
```

The when statement is almost identical to the `if` statement with some differences:

- Each condition has to be a constant expression since it is evaluated by the compiler.
- The statements within a branch do not open a new scope.
- The compiler checks the semantics and produces code *only* for the statements that belong to the first condition that evaluates to `true`.

The `when` statement is useful for writing platform specific code, similar to the `#ifdef` construct in the C programming language.

**Note**: To comment out a large piece of code, it is often better to use a `when false:` statement than to use real comments. This way nesting is possible.

## 9 Statements and indentation

Now that we covered the basic control flow statements, let's return to Nimrod indentation rules.

In Nimrod there is a distinction between *simple statements* and *complex statements*. *Simple statements* cannot contain other statements: Assignment, procedure calls or the `return` statement belong to the simple statements. *Complex statements* like `if`, `when`, `for`, `while` can contain other statements. To avoid ambiguities, complex statements always have to be indented, but single simple statements do not:

```
# no indentation needed for single assignment statement:
if x: x = false

# indentation needed for nested if statement:
if x:
  if y:
    y = false
  else:
    y = true

# indentation needed, because two statements follow the condition:
if x:
  x = false
  y = false
```

*Expressions* are parts of a statement which usually result in a value. The condition in an if statement is an example for an expression. Expressions can contain indentation at certain places for better readability:

```
if thisIsaLongCondition() and
    thisIsAnotherLongCondition(1,
       2, 3, 4):
  x = true
```

As a rule of thumb, indentation within expressions is allowed after operators, an open parenthesis and after commas.

With parenthesis and semicolons `(;)` you can use statements where only an expression is allowed:

```
# computes fac(4) at compile time:
const fac4 = (var x = 1; for i in 1..4: x *= i; x)
```

# 10 Procedures

To define new commands like `echo`, `readline` in the examples, the concept of a *procedure* is needed. (Some languages call them *methods* or *functions*.) In Nimrod new procedures are defined with the `proc` keyword:

```
proc yes(question: string): bool =
  echo(question, " (y/n)")
  while true:
    case readLine(stdin)
    of "y", "Y", "yes", "Yes": return true
    of "n", "N", "no", "No": return false
    else: echo("Please be clear: yes or no")

if yes("Should I delete all your important files?"):
  echo("I'm sorry Dave, I'm afraid I can't do that.")
else:
  echo("I think you know what the problem is just as well as I do.")
```

This example shows a procedure named `yes` that asks the user a `question` and returns true if he answered "yes" (or something similar) and returns false if he answered "no" (or something similar). A `return` statement leaves the procedure (and therefore the while loop) immediately. The `(question: string): bool` syntax describes that the procedure expects a parameter named `question` of type `string` and returns a value of type `bool`. `Bool` is a built-in type: the only valid values for `bool` are `true` and `false`. The conditions in if or while statements should be of the type `bool`.

Some terminology: in the example `question` is called a (formal) *parameter*, `"Should I..."` is called an *argument* that is passed to this parameter.

## 10.1 Result variable

A procedure that returns a value has an implicit `result` variable that represents the return value. A `return` statement with no expression is a shorthand for `return result`. So all three code snippets are equivalent:

```
return 42

result = 42
return

result = 42
return result
```

## 10.2 Parameters

Parameters are constant in the procedure body. Their value cannot be changed because this allows the compiler to implement parameter passing in the most efficient way. If the procedure needs to modify the argument for the caller, a `var` parameter can be used:

```
proc divmod(a, b: int; res, remainder: var int) =
  res = a div b         # integer division
  remainder = a mod b   # integer modulo operation

var
  x, y: int
divmod(8, 5, x, y) # modifies x and y
echo(x)
echo(y)
```

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. Note that the above example would better make use of a tuple as a return value instead of using var parameters.

## 10.3 Discard statement

To call a procedure that returns a value just for its side effects and ignoring its return value, a discard statement **has** to be used. Nimrod does not allow to silently throw away a return value:

```
discard yes("May I ask a pointless question?")
```

The return value can be ignored implicitely if the called proc/iterator has been declared with the `discardable` pragma:

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

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

## 10.4 Named arguments

Often a procedure has many parameters and it is not clear in which order the parameters appear. This is especially true for procedures that construct a complex data type. Therefore the arguments to a procedure can be named, so that it is clear which argument belongs to which parameter:

```
proc createWindow(x, y, width, height: int; title: string;
                  show: bool): Window =
   ...

var w = createWindow(show = true, title = "My Application",
                     x = 0, y = 0, height = 600, width = 800)
```

Now that we use named arguments to call `createWindow` the argument order does not matter anymore. Mixing named arguments with ordered arguments is also possible, but not very readable:

```
var w = createWindow(0, 0, title = "My Application",
                     height = 600, width = 800, true)
```

The compiler checks that each parameter receives exactly one argument.

## 10.5 Default values

To make the `createWindow` proc easier to use it should provide *default values*, these are values that are used as arguments if the caller does not specify them:

```
proc createWindow(x = 0, y = 0, width = 500, height = 700,
                  title = "unknown",
                  show = true): Window =
   ...

var w = createWindow(title = "My Application", height = 600, width = 800)
```

Now the call to `createWindow` only needs to set the values that differ from the defaults.

Note that type inference works for parameters with default values; there is no need to write `title: string = "unknown"`, for example.

## 10.6 Overloaded procedures

Nimrod provides the ability to overload procedures similar to C++:

```
proc toString(x: int): string = ...
proc toString(x: bool): string =
  if x: return "true"
  else: return "false"

echo(toString(13))   # calls the toString(x: int) proc
echo(toString(true)) # calls the toString(x: bool) proc
```

(Note that `toString` is usually the `$` operator in Nimrod.) The compiler chooses the most appropriate proc for the `toString` calls. How this overloading resolution algorithm works exactly is not discussed here (it will be specified in the manual soon). However, it does not lead to nasty surprises and is based on a quite simple unification algorithm. Ambiguous calls are reported as errors.

## 10.7 Operators

The Nimrod library makes heavy use of overloading - one reason for this is that each operator like + is a just an overloaded proc. The parser lets you use operators in *infix notation* (`a + b`) or *prefix notation* (`+ a`). An infix operator always receives two arguments, a prefix operator always one. Postfix operators are not possible, because this would be ambiguous: does `a @ @ b` mean `(a) @ (@b)` or `(a@) @ (b)`? It always means `(a) @ (@b)`, because there are no postfix operators in Nimrod.

Apart from a few built-in keyword operators such as `and`, `or`, `not`, operators always consist of these characters: `+ - * \ / < > = @ $ ~ & % ! ? ^ . |`

User defined operators are allowed. Nothing stops you from defining your own `@!?+~` operator, but readability can suffer.

The operator's precedence is determined by its first character. The details can be found in the manual.

To define a new operator enclose the operator in "`":

```
proc `$` (x: myDataType): string = ...
# now the $ operator also works with myDataType, overloading resolution
# ensures that $ works for built-in types just like before
```

The "`" notation can also be used to call an operator just like any other procedure:

```
if `==`( `+`(3, 4), 7): echo("True")
```

## 10.8 Forward declarations

Every variable, procedure, etc. needs to be declared before it can be used. (The reason for this is compilation efficiency.) However, this cannot be done for mutually recursive procedures:

```
# forward declaration:
proc even(n: int): bool

proc odd(n: int): bool =
  n == 1 or even(n-1)

proc even(n: int): bool =
  n == 0 or odd(n-1)
```

Here `odd` depends on `even` and vice versa. Thus `even` needs to be introduced to the compiler before it is completely defined. The syntax for such a forward declaration is simple: just omit the = and the procedure's body.

Later versions of the language may get rid of the need for forward declarations.

The example also shows that a proc's body can consist of a single expression whose value is then returned implicitly.

# 11 Iterators

Let's return to the boring counting example:

```
echo("Counting to ten: ")
for i in countup(1, 10):
  echo($i)
```

Can a `countup` proc be written that supports this loop? Lets try:

```
proc countup(a, b: int): int =
  var res = a
  while res <= b:
    return res
    inc(res)
```

However, this does not work. The problem is that the procedure should not only `return`, but return and **continue** after an iteration has finished. This *return and continue* is called a *yield* statement. Now the only thing left to do is to replace the `proc` keyword by `iterator` and there it is - our first iterator:

```
iterator countup(a, b: int): int =
  var res = a
  while res <= b:
    yield res
    inc(res)
```

Iterators look very similar to procedures, but there are several important differences:

- Iterators can only be called from for loops.
- Iterators cannot contain a `return` statement and procs cannot contain a `yield` statement.
- Iterators have no implicit `result` variable.
- Iterators do not support recursion.
- Iterators cannot be forward declared, because the compiler must be able to inline an iterator. (This restriction will be gone in a future version of the compiler.)

However, you can also use a `closure` iterator to get a different set of restrictions. See first class iterators for details.

# 12 Basic types

This section deals with the basic built-in types and the operations that are available for them in detail.

## 12.1 Booleans

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

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
```

## 12.2 Characters

The *character type* is named `char` in Nimrod. Its size is one byte. Thus it cannot represent an UTF-8 character, but a part of it. The reason for this is efficiency: for the overwhelming majority of use-cases, the resulting programs will still handle UTF-8 properly as UTF-8 was specially designed for this. Character literals are enclosed in single quotes.

Chars can be compared with the ==, <, <=, >, >= operators. The `$` operator converts a `char` to a `string`. Chars cannot be mixed with integers; to get the ordinal value of a `char` use the `ord` proc. Converting from an integer to a `char` is done with the `chr` proc.

## 12.3 Strings

String variables in Nimrod are **mutable**, so appending to a string is quite efficient. Strings in Nimrod are both zero-terminated and have a length field. One can retrieve a string's length with the builtin `len` procedure; the length never counts the terminating zero. Accessing the terminating zero is no error and often leads to simpler code:

```
if s[i] == 'a' and s[i+1] == 'b':
  # no need to check whether ``i < len(s)``!
  ...
```

The assignment operator for strings copies the string. You can use the `&` operator to concatenate strings and `add` to append to a string.

Strings are compared by their lexicographical order. All comparison operators are available. 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 *unichar*.

String variables are initialized with a special value, called `nil`. However, most string operations cannot deal with `nil` (leading to an exception being raised) for performance reasons. One should use empty strings `""` rather than `nil` as the *empty* value. But `""` often creates a string object on the heap, so there is a trade-off to be made here.

## 12.4 Integers

Nimrod has these integer types built-in: `int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64`.

The default integer type is `int`. Integer literals can have a *type suffix* to mark them to be of another integer type:

```
let
  x = 0     # x is of type ``int``
  y = 0'i8  # y is of type ``int8``
  z = 0'i64 # z is of type ``int64``
  u = 0'u   # u is of type ``uint``
```

Most often integers are used for counting objects that reside in memory, so `int` has the same size as a pointer.

The common operators `+ - * div mod < <= == != > >=` are defined for integers. The `and or xor not` operators are defined for integers too and provide *bitwise* operations. Left bit shifting is done with the `shl`, right shifting with the `shr` operator. Bit shifting operators always treat their arguments as *unsigned*. For arithmetic bit shifts ordinary multiplication or division can be used.

Unsigned operations all wrap around; they cannot lead to over- or underflow errors.

Automatic type conversion is performed in expressions where different kinds of integer types are used. However, if the type conversion loses information, the EOutOfRange exception is raised (if the error cannot be detected at compile time).

## 12.5 Floats

Nimrod has these floating point types built-in: `float float32 float64`.

The default float type is `float`. In the current implementation, `float` is always 64 bit wide.

Float literals can have a *type suffix* to mark them to be of another float type:

```
var
  x = 0.0      # x is of type ``float``
  y = 0.0'f32  # y is of type ``float32``
  z = 0.0'f64  # z is of type ``float64``
```

The common operators `+ - * / < <= == != > >=` are defined for floats and follow the IEEE standard.

Automatic type conversion in expressions with different kinds of floating point types is performed: the smaller type is converted to the larger. Integer types are **not** converted to floating point types automatically and vice versa. The `toInt` and `toFloat` procs can be used for these conversions.

# 13 Internal type representation

As mentioned earlier, the built-in `$` (stringify) operator turns any basic type into a string, which you can then print to the screen with the `echo` proc. However, advanced types, or types you may define yourself won't work with the `$` operator until you define one for them. Sometimes you just want to debug the current value of a complex type without having to write its `$` operator. You can use then the `repr` proc which works with any type and even complex data graphs with cycles. The following example shows that even for basic types there is a difference between the `$` and `repr` outputs:

```
var
  myBool = true
  myCharacter = 'n'
  myString = "nimrod"
  myInteger = 42
  myFloat = 3.14
echo($myBool, ":", repr(myBool))
# --> true:true
echo($myCharacter, ":", repr(myCharacter))
# --> n:'n'
echo($myString, ":", repr(myString))
# --> nimrod:0x10fa8c050"nimrod"
echo($myInteger, ":", repr(myInteger))
# --> 42:42
echo($myFloat, ":", repr(myFloat))
# --> 3.1400000000000001e+00:3.1400000000000001e+00
```

# 14 Advanced types

In Nimrod new types can be defined within a `type` statement:

```
type
  biggestInt = int64      # biggest integer type that is available
  biggestFloat = float64  # biggest float type that is available
```

Enumeration and object types cannot be defined on the fly, but only within a `type` statement.

## 14.1 Enumerations

A variable of an enumeration type can only be assigned a value of a limited set. This set consists of ordered symbols. Each symbol is mapped to an integer value internally. The first symbol is represented at runtime by 0, the second by 1 and so on. Example:

```
type
  TDirection = enum
    north, east, south, west

var x = south      # `x` is of type `TDirection`; its value is `south`
echo($x)           # writes "south" to `stdout`
```

(To prefix a new type with the letter `T` is a convention in Nimrod.) All comparison operators can be used with enumeration types.

An enumeration's symbol can be qualified to avoid ambiguities: `TDirection.south`.

The `$` operator can convert any enumeration value to its name, the `ord` proc to its underlying integer value.

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

An explicit ordered enum can have *holes*:

```
type
  TMyEnum = enum
    a = 2, b = 4, c = 89
```

| Operation | Comment |
|---|---|
| `ord(x)` | returns the integer value that is used to represent $x$'s value |
| `inc(x)` | increments $x$ by one |
| `inc(x, n)` | increments $x$ by $n$; $n$ is an integer |
| `dec(x)` | decrements $x$ by one |
| `dec(x, n)` | decrements $x$ by $n$; $n$ is an integer |
| `succ(x)` | returns the successor of $x$ |
| `succ(x, n)` | returns the $n$'th successor of $x$ |
| `prec(x)` | returns the predecessor of $x$ |
| `pred(x, n)` | returns the $n$'th predecessor of $x$ |

## 14.2 Ordinal types

Enumerations without holes, integer types, `char` and `bool` (and subranges) are called ordinal types. Ordinal types have quite a few special operations:

The `inc dec succ pred` operations can fail by raising an *EOutOfRange* or *EOverflow* exception. (If the code has been compiled with the proper runtime checks turned on.)

## 14.3 Subranges

A subrange type is a range of values from an integer or enumeration type (the base type). Example:

```
type
  TSubrange = range[0..5]
```

`TSubrange` is a subrange of `int` which can only hold the values 0 to 5. Assigning any other value to a variable of type `TSubrange` is a compile-time or runtime error. Assignments from the base type to one of its subrange types (and vice versa) are allowed.

The `system` module defines the important `natural` type as `range[0..high(int)]` (`high` returns the maximal value). Other programming languages mandate the usage of unsigned integers for natural numbers. This is often **wrong**: you don't want unsigned arithmetic (which wraps around) just because the numbers cannot be negative. Nimrod's `natural` type helps to avoid this common programming error.

## 14.4 Sets

The set type models the mathematical notion of a set. The set's basetype can only be an ordinal type. The reason is that sets are implemented as high performance bit vectors.

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

These operations are supported by sets:

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

## 14.5 Arrays

An array is a simple fixed length container. Each element in the array has the same type. The array's index type can be any ordinal type.

Arrays can be constructed via `[]`:

| 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` | strong subset relation (A is a real 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 |
| `A -+- B` | symmetric set difference (= (A - B) + (B - A)) |
| `card(A)` | the cardinality of A (number of elements in A) |
| `incl(A, elem)` | same as `A = A + {elem}` |
| `excl(A, elem)` | same as `A = A - {elem}` |

```
type
  TIntArray = array[0..5, int] # an array that is indexed with 0..5
var
  x: TIntArray
x = [1, 2, 3, 4, 5, 6]
for i in low(x)..high(x):
  echo(x[i])
```

The notation `x[i]` is used to access the i-th element of `x`. Array access is always bounds checked (at compile-time or at runtime). These checks can be disabled via pragmas or invoking the compiler with the `-bound_checks:off` command line switch.

Arrays are value types, like any other Nimrod type. The assignment operator copies the whole array contents.

The built-in `len` proc returns the array's length. `low(a)` returns the lowest valid index for the array $a$ and `high(a)` the highest valid index.

```
type
  TDirection = enum
    north, east, south, west
  TBlinkLights = enum
    off, on, slowBlink, mediumBlink, fastBlink
  TLevelSetting = array[north..west, TBlinkLights]
var
  level : TLevelSetting
level[north] = on
level[south] = slowBlink
level[east] = fastBlink
echo repr(level)  # --> [on, fastBlink, slowBlink, off]
echo low(level)   # --> north
echo len(level)   # --> 4
echo high(level)  # --> west
```

The syntax for nested arrays (multidimensional) in other languages is a matter of appending more brackets because usually each dimension is restricted to the same index type as the others. In nimrod you can have different dimensions with different index types, so the nesting syntax is slightly different. Building on the previous example where a level is defined as an array of enums indexed by yet another enum, we can add the following lines to add a light tower type subdivided in height levels accessed through their integer index:

```
type
  TLightTower = array[1..10, TLevelSetting]
var
  tower: TLightTower
tower[1][north] = slowBlink
tower[1][east] = mediumBlink
echo len(tower)     # --> 10
echo len(tower[1])  # --> 4
```

```
echo repr(tower)    # --> [[slowBlink, mediumBlink, ...more output..
# The following lines don't compile due to type mismatch errors
#tower[north][east] = on
#tower[0][1] = on
```

Note how the built-in `len` proc returns only the array's first dimension length. Another way of defining the `TLightTower` to show better its nested nature would be to omit the previous definition of the `TLevelSetting` type and instead write it embedded directly as the type of the first dimension:

```
type
  TLightTower = array[1..10, array[north..west, TBlinkLights]]
```

## 14.6 Sequences

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Since sequences are resizeable they are always allocated on the heap and garbage collected.

Sequences are always indexed with an `int` starting at position 0. The `len`, `low` and `high` operations are available for sequences too. The notation `x[i]` can be used to access the i-th element of `x`.

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 an openarray parameter.

Example:

```
var
  x: seq[int] # a sequence of integers
x = @[1, 2, 3, 4, 5, 6] # the @ turns the array into a sequence
```

Sequence variables are initialized with `nil`. However, most sequence operations cannot deal with `nil` (leading to an exception being raised) for performance reasons. Thus one should use empty sequences `@[]` rather than `nil` as the *empty* value. But `@[]` creates a sequence object on the heap, so there is a trade-off to be made here.

The `for` statement can be used with one or two variables when used with a sequence. When you use the one variable form, the variable will hold the value provided by the sequence. The `for` statement is looping over the results from the `items()` iterator from the system module. But if you use the two variable form, the first variable will hold the index position and the second variable will hold the value. Here the `for` statement is looping over the results from the `pairs()` iterator from the system module. Examples:

```
for i in @[3, 4, 5]:
  echo($i)
# --> 3
# --> 4
# --> 5

for i, value in @[3, 4, 5]:
  echo("index: ", $i, ", value:", $value)
# --> index: 0, value:3
# --> index: 1, value:4
# --> index: 2, value:5
```

## 14.7 Open arrays

**Note**: Openarrays can only be used for parameters.

Often fixed size arrays turn out to be too inflexible; procedures should be able to deal with arrays of different sizes. The openarray type allows this. 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.

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

## 14.8 Varargs

A `varargs` parameter is like an openarray parameter. However, it is also a means to implement passing a variable number of arguments to a procedure. The compiler converts the list of arguments to an array automatically:

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

myWriteln(stdout, "abc", "def", "xyz")
# is transformed by the compiler 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: TFile, a: varargs[string, `$`]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, 123, "abc", 4.0)
# is transformed by the compiler 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.

## 14.9 Tuples

A tuple type defines various named *fields* and an *order* of the fields. The constructor `()` can be used to construct tuples. The order of the fields in the constructor must match the order in the tuple's definition. Different tuple-types are *equivalent* if they specify the same fields of the same type in the same order.

The assignment operator for tuples copies each component. The notation `t.field` is used to access a tuple's field. Another notation is `t[i]` to access the `i`'th field. Here `i` needs to be a constant integer.

```
type
  TPerson = tuple[name: string, age: int] # type representing a person:
                                          # a person consists of a name
                                          # and an age
var
  person: TPerson
person = (name: "Peter", age: 30)
# the same, but less readable:
person = ("Peter", 30)

echo(person.name) # "Peter"
echo(person.age)  # 30

echo(person[0]) # "Peter"
echo(person[1]) # 30

# You don't need to declare tuples in a separate type section.
var building: tuple[street: string, number: int]
building = ("Rue del Percebe", 13)
echo(building.street)

# The following line does not compile, they are different tuples!
#person = building
# --> Error: type mismatch: got (tuple[street: string, number: int])
#     but expected 'TPerson'

# The following works because the field names and types are the same.
var teacher: tuple[name: string, age: int] = ("Mark", 42)
person = teacher
```

Even though you don't need to declare a type for a tuple to use it, tuples created with different field names will be considered different objects despite having the same field types.

### 14.10 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.

Nimrod 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 to 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.

The empty `[]` subscript notation can be used to *derefer* a reference, meaning to retrieve the item the reference points to. The `.` (access a tuple/object field operator) and `[]` (array/string/sequence index operator) operators perform implicit dereferencing operations for reference types:

```
type
  PNode = ref TNode
  TNode = tuple[le, ri: PNode, data: int]
var
  n: PNode
new(n)
n.data = 9
# no need to write n[].data; in fact n[].data is highly discouraged!
```

(As a convention, reference types use a 'P' prefix.)

To allocate a new traced object, the built-in procedure `new` has to be used. To deal with untraced memory, the procedures `alloc`, `dealloc` and `realloc` can be used. The documentation of the system module contains further information.

If a reference points to *nothing*, it has the value `nil`.

### 14.11 Procedural type

A procedural type is a (somewhat abstract) pointer to a procedure. `nil` is an allowed value for a variable of a procedural type. Nimrod uses procedural types to achieve functional programming techniques.

Example:

```
type
  TCallback = proc (x: int)

proc echoItem(x: Int) = echo(x)

proc forEach(callback: TCallback) =
  const
    data = [2, 3, 5, 7, 11]
  for d in items(data):
    callback(d)

forEach(echoItem)
```

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. The different calling conventions are listed in the manual.

## 15 Modules

Nimrod supports splitting a program into pieces with a module concept. Each module is in its own file. Modules enable information hiding and separate compilation. A module may gain access to symbols of another module by the import statement. Only top-level symbols that are marked with an asterisk (`*`) are exported:

```
# Module A
var
  x*, y: int

proc `*` *(a, b: seq[int]): seq[int] =
  # allocate a new sequence:
  newSeq(result, len(a))
  # multiply two int sequences:
  for i in 0..len(a)-1: result[i] = a[i] * b[i]

when isMainModule:
  # test the new ``*`` operator for sequences:
  assert(@[1, 2, 3] * @[1, 2, 3] == @[1, 4, 9])
```

The above module exports `x` and `*`, but not `y`.

The top-level statements of a module are executed at the start of the program. This can be used to initialize complex data structures for example.

Each module has a special magic constant `isMainModule` that is true if the module is compiled as the main file. This is very useful to embed tests within the module as shown by the above example.

Modules that depend on each other are possible, but strongly discouraged, because then one module cannot be reused without the other.

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); if an unknown identifier occurs then abort.

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 has been 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 the compiler has already
  # added T1 to A's interface symbol table
  return x + 1
```

A symbol of a module *can* be *qualified* with the `module.symbol` syntax. If the symbol is ambiguous, it even *has* to be qualified. A symbol is ambiguous if it is defined in two (or more) different modules and both modules are imported by a third one:

```
# Module A
var x*: string

# Module B
var x*: int

# 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
```

But this rule does not apply to procedures or iterators. Here the overloading rules apply:

```
# Module A
proc x*(a: int): string = return $a

# Module B
proc x*(a: string): string = return $a

# Module C
import A, B
write(stdout, x(3))   # no error: A.x is called
write(stdout, x(""))  # no error: B.x is called

proc x*(a: int): string = nil
write(stdout, x(3))   # ambiguous: which `x` is to call?
```

### 15.1 From statement

We have already seen the simple `import` statement that just imports all exported symbols. An alternative that only imports listed symbols is the `from import` statement:

```
from mymodule import x, y, z
```

### 15.2 Include statement

The include statement does something fundametally different than importing a module: it merely includes the contents of a file. The `include` statement is useful to split up a large module into several files:

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

**Note**: The documentation generator currently does not follow `include` statements, so exported symbols in an include file will not show up in the generated documentation.

## 16 Part 2

So, now that we are done with the basics, let's see what Nimrod offers apart from a nice syntax for procedural programming: Part II