Nimrod Tutorial (Part II) 0.9.2

Andreas Rumpf

May 21, 2013

Contents

1 Introduction

"Object-oriented programming is an exceptionally bad idea which could only have originated in California." –Edsger Dijkstra

This document is a tutorial for the advanced constructs of the *Nimrod* programming language. **Note that this document is somewhat obsolete as the ‘manual <manual.html>‘__ contains many more examples of the advanced language features.**

2 Pragmas

Pragmas are Nimrod's method to give the compiler additional information/ commands without introducing a massive number of new keywords. Pragmas are enclosed in the special `{.` and `.}` curly dot brackets. This tutorial does not cover pragmas. See the manual or user guide for a description of the available pragmas.

3 Object Oriented Programming

While Nimrod's support for object oriented programming (OOP) is minimalistic, powerful OOP technics can be used. OOP is seen as *one* way to design a program, not *the only* way. Often a procedural approach leads to simpler and more efficient code. In particular, prefering composition over inheritance is often the better design.

3.1 Objects

Like tuples, objects are a means to pack different values together in a structured way. However, objects provide many features that tuples do not: They provide inheritance and information hiding. Because objects encapsulate data, the `T()` object constructor should only be used internally and the programmer should provide a proc to initialize the object (this is called a *constructor*).

Objects have access to their type at runtime. There is an `of` operator that can be used to check the object's type:

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

  TStudent = object of TPerson # TStudent inherits from TPerson
    id: int                    # with an id field

var
  student: TStudent
  person: TPerson
assert(student of TStudent) # is true
# object construction:
student = TStudent(name: "Anton", age: 5, id: 2)
```

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*. New object types can only be defined within a type section.

Inheritance is done with the `object of` syntax. Multiple inheritance is currently not supported. If an object type has no suitable ancestor, `TObject` can be used as its ancestor, but this is only a convention. Objects that have no ancestor are implicitely `final`. You can use the `inheritable` pragma to introduce new object roots apart from `system.TObject`. (This is used in the GTK wrapper for instance.)

Note: Composition (*has-a* relation) is often preferable to inheritance (*is-a* relation) for simple code reuse. Since objects are value types in Nimrod, composition is as efficient as inheritance.

3.2 Mutually recursive types

Objects, tuples and references can model quite complex data structures which depend on each other; they are *mutually recursive*. In Nimrod these types can only be declared within a single type section. (Anything else would require arbitrary symbol lookahead which slows down compilation.)

Example:

```
type
  PNode = ref TNode # a traced reference to a TNode
  TNode = object
    le, ri: PNode   # left and right subtrees
    sym: ref TSym   # leaves contain a reference to a TSym

  TSym = object     # a symbol
    name: string    # the symbol's name
    line: int       # the line the symbol was declared in
    code: PNode     # the symbol's abstract syntax tree
```

3.3 Type conversions

Nimrod distinguishes between type casts and type conversions. Casts are done with the `cast` operator and force the compiler to interpret a bit pattern to be of another type.

Type conversions are a much more polite way to convert a type into another: They preserve the abstract *value*, not necessarily the *bit-pattern*. If a type conversion is not possible, the compiler complains or an exception is raised.

The syntax for type conversions is `destination_type(expression_to_convert)` (like an ordinary call):

```
proc getID(x: TPerson): int =
  return TStudent(x).id
```

The `EInvalidObjectConversion` exception is raised if `x` is not a `TStudent`.

3.4 Object variants

Often an object hierarchy is overkill in certain situations where simple variant types are needed.

An example:

```
# This is an example how an abstract syntax tree could be modeled in Nimrod
type
  TNodeKind = 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
  PNode = ref TNode
  TNode = object
    case kind: TNodeKind  # the ''kind'' field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: PNode
    of nkIf:
      condition, thenPart, elsePart: PNode

var n = PNode(kind: nkFloat, floatVal: 1.0)
# the following statement raises an 'EInvalidField' exception, because
# n.kind's value does not fit:
n.strVal = ""
```

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

3.5 Methods

In ordinary object oriented languages, procedures (also called *methods*) are bound to a class. This has disadvantages:

- Adding a method to a class the programmer has no control over is impossible or needs ugly workarounds.
- Often it is unclear where the method should belong to: is `join` a string method or an array method?

Nimrod avoids these problems by not assigning methods to a class. All methods in Nimrod are multi-methods. As we will see later, multi-methods are distinguished from procs only for dynamic binding purposes.

3.6 Method call syntax

There is a syntactic sugar for calling routines: The syntax `obj.method(args)` can be used instead of `method(obj, args)`. If there are no remaining arguments, the parentheses can be omitted: `obj.len` (instead of `len(obj)`).

This method call syntax is not restricted to objects, it can be used for any type:

```
echo("abc".len) # is the same as echo(len("abc"))
echo("abc".toUpper())
echo({'a', 'b', 'c'}.card)
stdout.writeln("Hallo") # the same as writeln(stdout, "Hallo")
```

(Another way to look at the method call syntax is that it provides the missing postfix notation.)

So "pure object oriented" code is easy to write:

```
import strutils

stdout.writeln("Give a list of numbers (separated by spaces): ")
stdout.write(stdin.readLine.split.map(parseInt).max.`$`)
stdout.writeln(" is the maximum!")
```

3.7 Properties

As the above example shows, Nimrod 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 a special setter syntax is needed:

```
type
  TSocket* = object of TObject
    FHost: int # cannot be accessed from the outside of the module
               # the 'F' prefix is a convention to avoid clashes since
               # the accessors are named 'host'

proc `host=`*(s: var TSocket, value: int) {.inline.} =
  ## setter of hostAddr
  s.FHost = value

proc host*(s: TSocket): int {.inline.} =
  ## getter of hostAddr
  return s.FHost

var
  s: TSocket
s.host = 34  # same as `host=`(s, 34)
```

(The example also shows `inline` procedures.)

The `[]` array access operator can be overloaded to provide array properties:

```
type
  TVector* = object
    x, y, z: float

proc `[]=`* (v: var TVector, i: int, value: float) =
  # setter
  case i
  of 0: v.x = value
  of 1: v.y = value
  of 2: v.z = value
  else: assert(false)

proc `[]`* (v: TVector, i: int): float =
  # getter
  case i
  of 0: result = v.x
  of 1: result = v.y
  of 2: result = v.z
  else: assert(false)
```

The example is silly, since a vector is better modelled by a tuple which already provides `v[]` access.

3.8 Dynamic dispatch

Procedures always use static dispatch. For dynamic dispatch replace the `proc` keyword by `method`:

```
type
  PExpr = ref object of TObject ## abstract base class for an expression
  PLiteral = ref object of PExpr
    x: int
  PPlusExpr = ref object of PExpr
    a, b: PExpr

# watch out: 'eval' relies on dynamic binding
method eval(e: PExpr): int =
  # override this base method
  quit "to override!"

method eval(e: PLiteral): int = e.x
method eval(e: PPlusExpr): int = eval(e.a) + eval(e.b)

proc newLit(x: int): PLiteral = PLiteral(x: x)
proc newPlus(a, b: PExpr): PPlusExpr = PPlusExpr(a: a, b: b)

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

Note that 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.

In a multi-method all parameters that have an object type are used for the dispatching:

```
type
  TThing = object of TObject
  TUnit = object of TThing
    x: int

method collide(a, b: TThing) {.inline.} =
  quit "to override!"

method collide(a: TThing, b: TUnit) {.inline.} =
  echo "1"

method collide(a: TUnit, b: TThing) {.inline.} =
  echo "2"

var
  a, b: TUnit
collide(a, b) # output: 2
```

As the example demonstrates, invocation of a multi-method cannot be ambiguous: Collide 2 is preferred over collide 1 because the resolution works from left to right. Thus `TUnit, TThing` is preferred over `TThing, TUnit`.

Perfomance note: Nimrod does not produce a virtual method table, but generates dispatch trees. This avoids the expensive indirect branch for method calls and enables inlining. However, other optimizations like compile time evaluation or dead code elimination do not work with methods.

4 Exceptions

In Nimrod exceptions are objects. By convention, exception types are prefixed with an 'E', not 'T'. The system module defines an exception hierarchy that you might want to stick to. Exceptions derive from E_Base, which provides the common interface.

Exceptions have to be allocated on the heap because their lifetime is unknown. The compiler will prevent you from raising an exception created on the stack. All raised exceptions should at least specify the reason for being raised in the `msg` field.

A convention is that exceptions should be raised in *exceptional* cases: For example, if a file cannot be opened, this should not raise an exception since this is quite common (the file may not exist).

4.1 Raise statement

Raising an exception is done with the `raise` statement:

```
var
  e: ref EOS
new(e)
e.msg = "the request to the OS failed"
raise e
```

If the `raise` keyword is not followed by an expression, the last exception is *re-raised.* For the purpose of avoiding repeating this common code pattern, the template `newException` in the `system` module can be used:

```
raise newException(EOS, "the request to the OS failed")
```

4.2 Try statement

The try statement handles exceptions:

```
# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: TFile
if open(f, "numbers.txt"):
  try:
    let a = readLine(f)
    let b = readLine(f)
    echo "sum: ", parseInt(a) + parseInt(b)
  except EOverflow:
    echo "overflow!"
  except EInvalidValue:
    echo "could not convert string to integer"
  except EIO:
    echo "IO error!"
  except:
    echo "Unknown exception!"
    # reraise the unknown exception:
    raise
  finally:
    close(f)
```

The statements after the `try` are executed unless an exception is raised. Then the appropriate `except` part is executed.

The empty `except` part is executed if there is an exception that is not explicitly listed. It is similar to an `else` part in `if` statements.

If there is a `finally` part, it is always executed after the exception handlers.

The exception is *consumed* in an `except` part. If an exception is not handled, it is propagated through the call stack. This means that often the rest of the procedure - that is not within a `finally` clause - is not executed (if an exception occurs).

If you need to *access* the actual exception object or message inside an `except` branch you can use the getCurrentException() and getCurrentExceptionMsg() procs from the system module. Example:

```
try:
  doSomethingHere()
except:
  let
    e = getCurrentException()
    msg = getCurrentExceptionMsg()
  echo "Got exception ", repr(e), " with message ", msg
```

4.3 Exception hierarchy

If you want to create your own exceptions you can inherit from E_Base, but you can also inherit from one of the existing exceptions if they fit your purpose. The exception tree is:

```
* E_Base
  * EAsynch
    * EControlC
  * ESynch
    * ESystem
      * EIO
      * EOS
        * EInvalidLibrary
    * EResourceExhausted
    * EOutOfMemory
    * EStackOverflow
  * EArithmetic
    * EDivByZero
    * EOverflow
  * EAccessViolation
  * EAssertionFailed
  * EInvalidValue
    * EInvalidKey
  * EInvalidIndex
  * EInvalidField
  * EOutOfRange
  * ENoExceptionToReraise
  * EInvalidObjectAssignment
  * EInvalidObjectConversion
  * EFloatingPoint
    * EFloatInvalidOp
    * EFloatDivByZero
    * EFloatOverflow
    * EFloatUnderflow
    * EFloatInexact
  * EDeadThread
```

See the system module for a description of each exception.

4.4 Annotating procs with raised exceptions

Through the use of the optional `{.raises.}` pragma you can specify that a proc is meant to raise a specific set of exceptions, or none at all. If the `{.raises.}` pragma is used, the compiler will verify that this is true. For instance, if you specify that a proc raises `EIO`, and at some point it (or one of the procs it calls) starts raising a new exception the compiler will prevent that proc from compiling. Usage example:

```
proc complexProc() {.raises: [EIO, EArithmetic].} =
  ...

proc simpleProc() {.raises: [].} =
  ...
```

Once you have code like this in place, if the list of raised exception changes the compiler will stop with an error specifying the line of the proc which stopped validating the pragma and the raised exception not being caught, along with the file and line where the uncaught exception is being raised, which may help you locate the offending code which has changed.

If you want to add the `{.raises.}` pragma to existing code, the compiler can also help you. You can add the `{.effect.}` pragma statement to your proc and the compiler will output all inferred effects up to that point (exception tracking is part of Nimrod's effect system). Another more roundabout way to find out the list of exceptions raised by a proc is to use the Nimrod `doc2` command which generates documentation for a whole module and decorates all procs with the list of raised exceptions. You can read more about Nimrod's effect system and related pragmas in the manual.

5 Generics

Generics are Nimrod's means to parametrize procs, iterators or types with type parameters. They are most useful for efficient type safe containers:

```
type
  TBinaryTree[T] = object      # TBinaryTree is a generic type with
                               # with generic param ''T''
    le, ri: ref TBinaryTree[T] # left and right subtrees; may be nil
    data: T                    # the data stored in a node
  PBinaryTree*[T] = ref TBinaryTree[T] # type that is exported

proc newNode*[T](data: T): PBinaryTree[T] =
  # constructor for a node
  new(result)
  result.dat = data

proc add*[T](root: var PBinaryTree[T], n: PBinaryTree[T]) =
  # insert a node into the tree
  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 PBinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: PBinaryTree[T]): T =
  # Preorder traversal of a binary tree.
  # Since recursive iterators are not yet implemented,
  # this uses an explicit stack (which is more efficient anyway):
  var stack: seq[PBinaryTree[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: PBinaryTree[string] # instantiate a PBinaryTree with ''string''
add(root, newNode("hallo")) # instantiates ''newNode'' and ''add''
add(root, "world")          # instantiates the second ''add'' proc
for str in preorder(root):
  stdout.writeln(str)
```

The example shows a generic binary tree. Depending on context, the brackets are used either to introduce type parameters or to instantiate a generic proc, iterator or type. As the example shows, generics work with overloading: the best match of `add` is used. The built-in `add` procedure for sequences is not hidden and is used in the `preorder` iterator.

6 Templates

Templates are a simple substitution mechanism that operates on Nimrod's abstract syntax trees. Templates are processed in the semantic pass of the compiler. They integrate well with the rest of the language and share none of C's preprocessor macros flaws.

To *invoke* a template, call it like a procedure.

Example:

```
template `!=` (a, b: expr): expr =
  # this definition exists in the System module
  not (a == b)

assert(5 != 6) # the compiler rewrites that to: assert(not (5 == 6))
```

The `!=`, `>`, `>=`, `in`, `notin`, `isnot` operators are in fact templates: this has the benefit that if you overload the `==` operator, the `!=` operator is available automatically and does the right thing. (Except for IEEE floating point numbers - NaN breaks basic boolean logic.)

`a > b` is transformed into `b < a`. `a in b` is transformed into `contains(b, a)`. `notin` and `isnot` have the obvious meanings.

Templates are especially useful for lazy evaluation purposes. Consider a simple proc for logging:

```
const
  debug = True

proc log(msg: string) {.inline.} =
  if debug: stdout.writeln(msg)

var
  x = 4
log("x has the value: " & $x)
```

This code has a shortcoming: if `debug` is set to false someday, the quite expensive `$` and `&` operations are still performed! (The argument evaluation for procedures is *eager*).

Turning the `log` proc into a template solves this problem:

```
const
  debug = True

template log(msg: string) =
  if debug: stdout.writeln(msg)

var
  x = 4
log("x has the value: " & $x)
```

The parameters' types can be ordinary types or the meta types `expr` (stands for *expression*), `stmt` (stands for *statement*) or `typedesc` (stands for *type description*). If the template has no explicit return type, `stmt` is used for consistency with procs and methods.

The template body does not open a new scope. To open a new scope use a `block` statement:

```
template declareInScope(x: expr, t: typeDesc): stmt {.immediate.} =
  var x: t

template declareInNewScope(x: expr, t: typeDesc): stmt {.immediate.} =
  # open a new scope:
  block:
    var x: t

declareInScope(a, int)
a = 42  # works, `a` is known here

declareInNewScope(b, int)
b = 42  # does not work, `b` is unknown
```

(The manual explains why the `immediate` pragma is needed for these templates.)

If there is a `stmt` parameter it should be the last in the template declaration. The reason is that statements can be passed to a template via a special `:` syntax:

```
template withFile(f: expr, filename: string, mode: TFileMode,
                  body: stmt): stmt {.immediate.} =
  block:
    let fn = filename
    var f: TFile
    if open(f, fn, mode):
      try:
        body
      finally:
        close(f)
    else:
      quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeln("line 1")
  txt.writeln("line 2")
```

In the example the two `writeln` statements are bound to the `body` parameter. The `withFile` template contains boilerplate code and helps to avoid a common bug: to forget to close the file. Note how the `let fn = filename` statement ensures that `filename` is evaluated only once.

7 Macros

Macros enable advanced compile-time code transformations, but they cannot change Nimrod's syntax. However, this is no real restriction because Nimrod's syntax is flexible enough anyway.

To write a macro, one needs to know how the Nimrod concrete syntax is converted to an abstract syntax tree (AST). The AST is documented in the macros module.

There are two ways to invoke a macro:

1. invoking a macro like a procedure call (expression macros)
2. invoking a macro with the special `macrostmt` syntax (statement macros)

7.1 Expression Macros

The following example implements a powerful `debug` command that accepts a variable number of arguments:

```
# to work with Nimrod syntax trees, we need an API that is defined in the
# ``macros`` module:
import macros

macro debug(n: varargs[expr]): stmt =
  # `n` is a Nimrod AST that contains a list of expressions;
  # this macro returns a list of statements:
  result = newNimNode(nnkStmtList, n)
  # iterate over any argument that is passed to this macro:
```

```
  for i in 0..n.len-1:
    # add a call to the statement list that writes the expression;
    # `toStrLit` converts an AST to its string representation:
    result.add(newCall("write", newIdentNode("stdout"), toStrLit(n[i])))
    # add a call to the statement list that writes ": "
    result.add(newCall("write", newIdentNode("stdout"), newStrLitNode(": ")))
    # add a call to the statement list that writes the expressions value:
    result.add(newCall("writeln", newIdentNode("stdout"), n[i]))

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, ": ")
writeln(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeln(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeln(stdout, x)
```

7.2 Statement Macros

Statement macros are defined just as expression macros. However, they are invoked by an expression following a colon.

The following example outlines a macro that generates a lexical analyzer from regular expressions:

```
macro case_token(n: stmt): stmt =
  # creates a lexical analyzer from regular expressions
  # ... (implementation is an exercise for the reader :-)
  nil

case_token: # this colon tells the parser it is a macro statement
of r"[A-Za-z_]+[A-Za-z_0-9]*":
  return tkIdentifier
of r"0-9+":
  return tkInteger
of r"[\+\-\*\?]+":
  return tkOperator
else:
  return tkUnknown
```

7.3 Term rewriting macros

Term rewriting macros can be used to enhance the compilation process with user defined optimizations; see this document for further information.