Nim Tutorial (Part III) nimversion

Arne Döring

February 12, 2019

Contents

1 Introduction

"With Great Power Comes Great Responsibility." – Spider Man's Uncle

This document is a tutorial about Nim's macro system. A macro is a function that is executed at compile time and transforms a Nim syntax tree into a different tree.

Examples of things that can be implemented in macros:

- An assert macro that prints both sides of a comparison operator, if

the assertion fails. `myAssert(a == b)` is converted to `if a != b: quit($a " != " $b)`

- A debug macro that prints the value and the name of the symbol.

`myDebugEcho(a)` is converted to `echo "a: ", a`

- Symbolic differentiation of an expression.

`diff(a*pow(x,3) + b*pow(x,2) + c*x + d, x)` is converted to `3*a*pow(x,2) + 2*a*x + c`

1.1 Macro Arguments

The types of macro arguments have two faces. One face is used for the overload resolution, and the other face is used within the macro body. For example, if `macro foo(arg: int)` is called in an expression `foo(x)`, `x` has to be of a type compatible to int, but *within* the macro's body `arg` has the type `NimNode`, not `int`! Why it is done this way will become obvious later, when we have seen concrete examples.

There are two ways to pass arguments to a macro, an argument can be either `typed` or `untyped`.

1.2 Untyped Arguments

Untyped macro arguments are passed to the macro before they are semantically checked. This means the syntax tree that is passed down to the macro does not need to make sense for Nim yet, the only limitation is that it needs to be parseable. Usually the macro does not check the argument either but uses it in the transformation's result somehow. The result of a macro expansion is always checked by the compiler, so apart from weird error messages nothing bad can happen.

The downside for an `untyped` argument is that these do not play well with Nim's overloading resolution.

The upside for untyped arguments is that the syntax tree is quite predictable and less complex compared to its `typed` counterpart.

1.3 Typed Arguments

For typed arguments, the semantic checker runs on the argument and does transformations on it, before it is passed to the macro. Here identifier nodes are resolved as symbols, implicit type conversions are visible in the tree as calls, templates are expanded and probably most importantly, nodes have type information. Typed arguments can have the type `typed` in the arguments list. But all other types, such as `int`, `float` or `MyObjectType` are typed arguments as well, and they are passed to the macro as a syntax tree.

1.4 Static Arguments

Static arguments are a way to pass values as values and not as syntax tree nodes to a macro. For example for `macro foo(arg: static[int])` in the expression `foo(x)`, `x` needs to be an integer constant, but in the macro body `arg` is just like a normal parameter of type `int`.

```
import macros

macro myMacro(arg: static[int]): untyped =
  echo arg # just an int (7), not ``NimNode``

myMacro(1 + 2 * 3)
```

1.5 Code blocks as arguments

It is possible to pass the last argument of a call expression in a separate code block with indentation. For example the following code example is a valid (but not a recommended) way to call `echo`:

```
echo "Hello ":
  let a = "Wor"
  let b = "ld!"
  a & b
```

For macros this way of calling is very useful; syntax trees of arbitrary complexity can be passed to macros with this notation.

1.6 The Syntax Tree

In order to build a Nim syntax tree one needs to know how Nim source code is represented as a syntax tree, and how such a tree needs to look like so that the Nim compiler will understand it. The nodes of the Nim syntax tree are documented in the macros module. But a more interactive way to explore the Nim syntax tree is with `macros.treeRepr`, it converts a syntax tree into a multi line string for printing on the console. It can be used to explore how the argument expressions are represented in tree form and for debug printing of generated syntax tree. `dumpTree` is a predefined macro that just prints its argument in tree representation, but does nothing else. Here is an example of such a tree representation:

```
dumpTree:
  var mt: MyType = MyType(a:123.456, b:"abcdef")

# output:
#   StmtList
#     VarSection
#       IdentDefs
#         Ident "mt"
#         Ident "MyType"
#         ObjConstr
#           Ident "MyType"
#           ExprColonExpr
#             Ident "a"
#             FloatLit 123.456
#           ExprColonExpr
#             Ident "b"
#             StrLit "abcdef"
```

1.7 Custom sematic checking

The first thing that a macro should do with its arguments is to check if the argument is in the correct form. Not every type of wrong input needs to be caught here, but anything that could cause a crash during macro evaluation should be caught and create a nice error message. `macros.expectKind` and `macros.expectLen` are a good start. If the checks need to be more complex, arbitrary error messages can be created with the `macros.error` proc.

```
macro myAssert(arg: untyped): untyped =
  arg.expectKind nnkInfix
```

1.8 Generating Code

There are two ways to generate the code. Either by creating the syntax tree with expressions that contain a lot of calls to `newTree` and `newLit`, or with `quote do:` expressions. The first option offers the best low level control for the syntax tree generation, but the second option is much less verbose. If you choose to create the syntax tree with calls to `newTree` and `newLit` the macro `marcos.dumpAstGen` can help you with the verbosity. `quote do:` allows you to write the code that you want to generate literally, backticks are used to insert code from `NimNode` symbols into the generated expression. This means that you can't use backticks within `quote do:` for anything else than injecting symbols. Make sure to inject only symbols of type `NimNode` into the generated syntax tree. You can use `newLit` to convert arbitrary values into expressions trees of type `NimNode` so that it is safe to inject them into the tree.

```
import macros

type
  MyType = object
    a: float
    b: string

macro myMacro(arg: untyped): untyped =
  var mt: MyType = MyType(a:123.456, b:"abcdef")

  # ...

  let mtLit = newLit(mt)

  result = quote do:
    echo `arg`
    echo `mtLit`

myMacro("Hallo")
```

The call to `myMacro` will generate the following code:

```
echo "Hallo"
echo MyType(a: 123.456'f64, b: "abcdef")
```

1.9 Building your first macro

To give a footstart to writing macros we will show now how to implement the `myDebug` macro mentioned earlier. The first thing to do is to build a simple example of the macro usage, and then just print the argument. This way it is possible to get an idea of a correct argument should be look like.

```
import macros

macro myAssert(arg: untyped): untyped =
  echo arg.treeRepr

let a = 1
let b = 2

myAssert(a != b)
```

```
Infix
  Ident "!="
  Ident "a"
  Ident "b"
```

From the output it is possible to see that the information that the argument is an infix operator (node kind is "Infix"), as well as that the two operands are at index 1 and 2. With this information the actual macro can be written.

```
import macros

macro myAssert(arg: untyped): untyped =
  # all node kind identifiers are prefixed with "nnk"
  arg.expectKind nnkInfix
  arg.expectLen 3
  # operator as string literal
  let op  = newLit(" " & arg[0].repr & " ")
  let lhs = arg[1]
  let rhs = arg[2]

  result = quote do:
    if not `arg`:
      raise newException(AssertionError,$`lhs` & `op` & $`rhs`)

let a = 1
let b = 2

myAssert(a != b)
myAssert(a == b)
```

This is the code that will be generated. To debug what the macro actually generated, the statement `echo result.repr` can be used, in the last line of the macro. It is also the statement that has been used to get this output.

```
if not (a != b):
  raise newException(AssertionError, $a & " != " & $b)
```

1.10 With Power Comes Responsibility

Macros are very powerful. A good advice is to use them as little as possible, but as much as necessary. Macros can change the semantics of expressions, making the code incomprehensible for anybody who does not know exactly what the macro does with it. So whenever a macro is not necessary and the same logic can be implemented using templates or generics, it is probably better not to use a macro. And when a macro is used for something, the macro should better have a well written documentation. For all the people who claim to write only perfectly self-explanatory code: when it comes to macros, the implementation is not enough for documentation.

1.11 Limitations

Since macros are evaluated in the compiler in the NimVM, macros share all the limitations of the NimVM. They have to be implemented in pure Nim code. Macros can start external processes on the shell, but they cannot call C functions except from those that are built in the compiler.

2 More Examples

This tutorial can only cover the basics of the macro system. There are macros out there that could be an inspiration for you of what is possible with it.

2.1 Strformat

In the Nim standard library, the `strformat` library provides a macro that parses a string literal at compile time. Parsing a string in a macro like here is generally not recommended. The parsed AST cannot have type information, and parsing implemented on the VM is generally not very fast. Working on AST nodes is almost always the recommended way. But still `strformat` is a good example for a practical use case for a macro that is slightly more complex that the `assert` macro.

Strformat

2.2 Ast Pattern Matching

Ast Pattern Matching is a macro library to aid in writing complex macros. This can be seen as a good example of how to repurpose the Nim syntax tree with new semantics.

Ast Pattern Matching

2.3 OpenGL Sandbox

This project has a working Nim to GLSL compiler written entirely in macros. It scans recursively though all used function symbols to compile them so that cross library functions can be executed on the GPU.

OpenGL Sandbox