一、介绍

本文档是关于nim的宏系统的教程，宏是在编译时执行并将nim语法树转换为不同树的函数。

可以在宏中实现的事情的例子：

- 如果断言失败，则打印比较运算符两侧的断言宏。myAssert (a == b)转换为if a != b : quit (a ”! = ” b)
- 打印符号的值和名称的调试宏。myDebugEcho (a)被转换为echo "a: " , a
- 表达式的符号微分。diff (a * pow (x , 3) + b * pow (x , 2) + c * x + d , x)转换为 3 * a * pow (x , 2) + 2 * b * x + c

1. 宏参数 Macro Arguments

宏参数的类型有两个方面，一面用于重载解析，另一面用于宏体内。例如：如果在表达式foo (x) 中调用宏 foo (arg : int)，则 x必须是与 int 兼容的类型，但在宏的主体中， arg的类型为 NimNode，而不是int！当我们看到具体的例子时，为什么这样做会很明显。

有两种方法可以将参数传递给宏，参数可以是typed或untyped。

2. 无类型参数 Untyped Arguments

未类型化的宏参数在进行语义检查之前被传递给宏。这意味着传递给宏的语法树还不需要对 Nim 有意义，唯一的限制是它需要可解析。通常，宏也不检查参数，而是以某种方式在转换结果中使用它。宏展开的结果总是由编译器检查，所以除了奇怪的错误信息外，不会有任何不好的事情发生。

无类型参数的缺点是它们不能很好地与 Nim 的重载解决方案配合使用。

无类型参数的好处是语法树是相当可预测的，并且与其类型化的对应物相比不那么复杂。

3. 类型参数 Typed Arguments

对于类型化参数，语义检查器在参数上运行并对其进行转换，然后再将其传递给宏。在这里，标识符节点被解析为符号，隐式类型转换在调用时在树中可见，模板被扩展，并且可能最重要的是，节点具有类型信息。类型化参数可以具有在参数列表中键入的类型。但是所有其他类型，例如int、float或MyObjectType也是类型参数，它们作为语法树传递给宏。

4. 静态参数 Static Arguments

静态参数是一种将值作为值而不是作为语法树节点传递给宏的方法。例如对于表达式foo (x)中的宏 foo (arg : static [int])，x需要是一个整型常量，但在宏体中arg就像一个普通的int类型参数。

```
import std/macros

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

myMacro(1 + 2 * 3)
```

5. 代码块参数 code block

可以在带有缩进的单独代码块中传递调用表达式的最后一个参数。例如，以下代码示例是调用 echo 的有效（但不推荐）方式：

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

对于宏，这种调用方式非常有用；任意复杂的语法树都可以用这种表示法传递给宏。

6. 语法树

为了构建 Nim 语法树，需要知道 Nim 源代码是如何表示为语法树的，以及这样一棵树需要是什么样子才能让 Nim 编译器理解它。Nim 语法树的节点记录在宏模块中。但是探索 Nim 语法树的一种更具交互性的方法是使用macros.treeRepr，它将语法树转换为多行字符串以在控制台上打印。它可用于探索参数表达式如何以树形式表示以及用于调试生成语法树的打印。

dumpTree是一个预定义的宏，它只在树表示中打印它的参数，但不做任何其他事情。这是这种树表示的示例：

```
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"
```

7. 自定义语义检查

宏对其参数应该做的第一件事是检查参数的形式是否正确。并非所有类型的错误输入都需要在这里捕获，但是在宏评估期间可能导致崩溃的任何情况都应该被捕获并创建一个很好的错误消息。

macros.expectKind and macros.expectLen are a good start.

如果检查需要更复杂，可以使用macros.error

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

8. 生成代码

有两种方法可以生成代码。通过使用包含大量调用newTree和newLit的表达式创建语法树，或者使用quote do :表达式。第一个选项为语法树生成提供了最好的低级控制，但第二个选项要简洁得多。如果您选择通过调用newTree和newLit宏来创建语法树。dumpAstGen可以帮助您解决冗长的问题。

quote do ： 允许您编写要按字面意义生成的代码。反引号用于将来自NimNode符号的代码插入到生成的表达式中。

```
macro a(i) = quote do: let `i` = 0
a b
```

只要需要反引号，就可以定义自定义前缀运算符。

```
macro a(i) = quote("@") do: assert @i == 0
let b = 0
a b
```

注入的符号在解析为符号时需要引用重音。

```
macro a(i) = quote("@") do: let `@i` == 0
a b
```

确保只将NimNode类型的符号注入到生成的语法树中。您可以使用newLit将任意值转换为NimNode类型的表达式树，以便将它们注入树中是安全的。

```
import std/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")
```

对myMacro 的调用将生成以下代码：

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

9. 建立你的第一个宏

为了提供编写宏的起点，我们现在将展示如何实现前面提到的myDebug宏。首先要做的是构建一个简单的宏用法示例，然后打印参数。这样就可以了解正确的论证应该是什么样子。

```
import std/macros

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

let a = 1
let b = 2

myAssert(a != b)
```

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

从输出中可以看出参数是一个中缀运算符（节点种类是“Infix”），并且两个操作数位于索引 1 和 2 处。有了这些信息，就可以编写实际的宏了。

```
import std/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(AssertionDefect,$`lhs` & `op` & $`rhs`)

let a = 1
let b = 2

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

这是将要生成的代码。要调试宏实际生成的内容，可以在宏的最后一行使用echo result.repr

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

10. 权力伴随责任

宏可以改变表达式的语义，使代码对于任何不确切知道宏用它做什么的人来说都难以理解。因此，只要不需要宏并且可以使用模板或泛型实现相同的逻辑，最好不要使用宏。当宏用于某事时，宏最好有一个编写良好的文档。对于所有声称只编写完美的不言自明代码的人来说：当涉及到宏时，实现不足以记录。

11. 限制

由于宏是在 NimVM 的编译器中求值的，因此宏具有 NimVM 的所有限制。它们必须在纯 Nim 代码中实现。宏可以在 shell 上启动外部进程，但它们不能调用 C 函数，除非是编译器内置的函数。