Introduction to Nim

Boitumelo Phetla

[1] *General-purpose Programming Language*

May 19, 2019

Nim is unique. It is multi-paradigm, general purpose programming language with syntax like Python. However, the language or the design of this programming language does not emphasize on Object Oriented Programming style/concept. The language follows its own programming styling, it is imperative that its syntax styling is kept as guided by Nim's reference manual. Nim focuses mainly on effiency, expressiveness, and elegance. Nim is much like any other programming language with features such as concurrency, parallelism, user-defined types, the standard library, and more. Nim also has Nim's specific features such as asynchronous input/output, metaprogramming, and the foreign function interface.

1 What is Nim?

Impatience is my vice, I want to know I have the right tools for the job before I commit. So what does this mean?

How to install Nim

```
$bash: brew install nim

==> Pouring nim-0.19.4.mojave.bottle.tar.gz
/usr/local/Cellar/nim/0.19.4: 411 files, 12.8MB
```

Simple Script

Create file hello_world.nim

```
#simple comment line
echo "Hello world!"
```

Compile Script

Compile hello_world.nim

```
$bash: nim compile --run hello_world.nim
Hint: used config file
    '/usr/local/Cellar/nim/0.19.4/nim/config/nim.cfg'
    [Conf]
Hint: system [Processing]
Hint: hello_world [Processing]
CC: hello_world
CC: stdlib_system
Hint: [Link]
Hint: operation successful (12382 lines
    compiled; 0.660 sec total; 16.383MiB
    peakmem; Debug Build) [SuccessX]
Hint:
    /Volumes/pulse/100/Nim/ch01/code/hello_world
    [Exec]
Hello world!
```

Terminal output results

At this stage I try to understand what the terminal is showing and if I cannot make sense of it, I at least try to find out if my output results is as expected.

```
Hint: used config file
    '/usr/local/Cellar/nim/0.19.4/nim/config/nim.cfg'
    [Conf]
     <---- :accessing this directory shows
         |------ nim.cfg
         |------ nimdoc.cfg
         |------ nimdoc.tex.cfg
Hint: system [Processing]
Hint: hello_world [Processing]
```

```
CC: hello_world  <--- looks like Nim is using
    clang
CC: stdlib_system <--- looks like Nim is using
    clang
Hint: [Link]
Hint: operation successful (12382 lines
    compiled; 0.660 sec total; 16.383MiB
    peakmem; Debug Build) [SuccessX] <---
    processed Nim packages
Hint:
    /Volumes/pulse/100/Nim/ch01/code/hello_world
    [Exec] <--- script location
Hello world! <--- Expected output
```

What is in the file/script

```
In the code repo Nim has created some sort of an
    object file hello_world (machine code) that
    Nim (JVM if it is running on it) uses to
    compile and read the script. Languages such
    as Java, C, C++ uses such approach
    (compiled languages). Python does not do
    such (interpreted language).

$bash: tree -L 1
    hello_world
    hello_world.nim

0 directories, 2 files
```

1.1 Beginning to learn Nim

Nim is still a relatively new programming language (first scribbed books —- Nim in Action, Dominik Picheta).

What should you know going in:

- The language is not fully complete
- Nim is a general-purpose programming language
- effiency, expressiveness and elegance are Nim's standardised priority markers and they rank according to the order mentioned
- Nim shares many of Python's characteristics
- Nim is a compiled language (translated to C first)
- Nim is well suited for systems programming (hardware, OSs, IoT, etc)
- Nim is one of the few languages that uses its own language to interpret itself
- type system, execution model
- Applications that perform I/O operations, such as reading files or sending data over a network, are also well supported by Nim.
- Web applications (web frameworks like Jester)
- Nim can compile JavaScript
- Things that you might be familiar with are covered in Nim (procedures, methods, iterators, generics and templates)
- Nim's documentation

1.2 Why should you use Nim

Python, C, C++, ObjC, JavaScript $\longrightarrow$ Nim's design follows that of Python with the capability to perform foreign interfacing (C, C++, ObjectiveC, JavaScript)

Nim project started in 2005 $\longrightarrow$ The language is 14 years old, so you will be some sort of an early adopter

Garbage collector $\longrightarrow$ Garbage collection + manual memory management

Game developers $\longrightarrow$ because of a garbage collector that can be turned on and off this is a useful application for GameDevs

Scientific computing $\longrightarrow$ Data Scientists

Scripting $\longrightarrow$ Clue codes

Operating Systems $\longrightarrow$ Supports Windows, Linux, Unix

effiency $\longrightarrow$ Nim focues on compile-time mechanisms (runtime becomes effient)

Package manager $\longrightarrow$ Nimble

Environments to use Nim $\longrightarrow$ Web applications, Kernel

Andreas Rumpf $\longrightarrow$ Andreas Rumpf is the designer of Nimrod programming language, which he develops in his spare time. He is a software engineer working at a top secret company and constantly attempts to create his own start-up which he will allow himself to program in Nimrod fulltime. He has programmed in various programming languages over the years (including quite obscure ones) without being satisfied with any of them. Andreas Rumpf holds a degree in Computer Science which he obtained from the University of Kaiserslautern.

The compiler, standard library, and related tools are all open source and written in Nim.

1.3 Core features of Nim

1. Metaprogramming $\longrightarrow$ Read, Generate, Analyze and Transform source code
2. Style-insensitive $\longrightarrow$ camelCase or snake_case
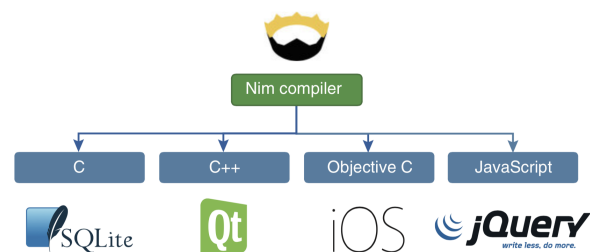3. Compilation to C $\longrightarrow$ enhances the language's performance

Figure 1: *Compilation backends*

1.4 Metaprogramming

Metaprogramming is the writing of computer programs with the ability to treat programs as their data. It means that a program could be designed to read, generate, analyse and/or transform other programs, and even modify itself while running. *Sounds like stuff from the movies*.

[HookRace blog](#)

- Normal procs and inline iterators
- Generic procs and closure iterators
- Templates
- Macros

With Nim's metaprogramming capabilities you are able to write domain-specific languages (DSL) shown below in simple few lines.

```
echo("<html>")
echo(" <body>")
echo("   <p>Hello World</p>")
echo(" </body>")
echo("</html>")

output:
<html>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

1.4.1 Normal procs (Functions)

```
#declare function
proc stationName(station: string) =
        echo "Analyzing " , station, " station"

#call function
stationName("Westergloor")   # f(a)
"Azaadville".stationName     # a.f = f(a)
stationName "Mohlakeng"      # f a

output:
Analyzing Westergloor station
Analyzing Azaadville station
Analyzing Mohlakeng station
```

1.4.2 What does style insensitive mean?

Notice the use of method to_upper and toUpper executes the same thing, but adopts snake_case and camelCase.

```
#style insensitive
import strutils

proc nameFormat(fm: string, name: string) =
  if fm == "Y":
```

```
    echo "called toUpper() : " , name.toUpper()
  else:
    echo "called to_upper(): " ,name.to_upper()

nameFormat("Y", "john")
nameFormat("N", "john")

output:
called toUpper() : JOHN
called to_upper(): JOHN
```

Nim is a static type system programming language with some dynamic type system interfacing (or supporting) to enable both free-error-proned language and less hindrance to development/writing code. Languages that are dynamically typed are languages like python. Basically, the meaning is that your variables, functions and data structures are not primitively declared or typed to a certain primitive data type (float, int, string, etc). The data assigned to your function will dynamically assign a primitive data type to your function or variable.

```
def array(argument, *args):
   #**args, argument <--not assigned any type
   flag = False
   if argument in args:
      print("Item found")
      flag = True
   else:
      for item in args:
         print(item)
   return flag <--returns a boolean

if __name__ == "__main__":
   #search 1 in 2, 3, 4, 5, 6
   #return false if not found or true if found
   flag = array(1, 2, 3, 4, 5, 6)
   print(flag)
```

The code above shows how powerful dynamically typed languages are. However, Nim likes to argue that their static type structure is as powerful unless you employ *ptrs from C that are not so friendly on memory saving. How does Nim take care of its performance?

- Arrays are bounds-checked at compile time, or at runtime when compile-time checks aren't possible, preventing both buffer overflows and buffer overreads.
- Pointer arithmetic isn't possible for reference types as they're entirely managed by Nim's garbage collector; this prevents issues such as dangling pointers and other memory issues related to managing memory manually.
- Variables are always initialized by Nim to their default values, which prevents variables containing unexpected and corrupt data.

1.5   Generic Programming

This is a feature that is rather unique and interesting, the ability to generically chose whether a typed variable can be used as one type or the other.

```
proc intFloat(number: int | float) =
    echo(number)

intFloat(2)    #f(a)
intFloat 2.2   #f a
3.14.intFloat  #a.f
```

1.6   Foreign language interface

1.6.1   nim compile to JavaScript

```
nim js -d:nodejs [name of script].nim
```

Compiling the code creeates a nimcache directory within the current working directory and generates the JavaScript script in it.

```
tree nimcache
nimcache/
        |_____ fn1.js
```

1.6.2   nim compile to C

```
nim -c -d:release c fn1
```

The converted C code is stored in the home directory under a hidden file (.cache/nim). It is usually assigned the same name as the nim script name with an suffix underscore some letter.

```
bash$: cd $HOME/.cache/nim
tree
 |_____ fn1_r/
            |_____ fn1.c
            |_____ fn1.json
            |_____ stdlib_system.c
1 directory, 3 files
```

1.6.3   nim compile to ObjectiveC

```
nim objc --cpu:amd64 --os:macosx --compile_only
    --gen_script [script name]
```

The converted objective C code is stored in the home directory under a hidden file (.cache/nim). It is usually assigned the same name as the nim script name with an suffix underscore some letter.

```
fn1_d/
    |_____compile_fn1.sh
    |_____fn1.deps
```

```
    |_____fn1.json
    |_____fn1.m
    |_____nimbase.h
    |_____stdlib_system.m
```

1.7   Why does Nim compile to C

The C programming language is very well established as a systems programming language and has been in use for over 40 years. C is one of the most portable programming languages, with multiple implementations for Windows, Linux, Mac OS, x86, AMD64, ARM, and many other, more obscure OSs and platforms. C compilers sup- port everything from supercomputers to microcontrollers. They're also very mature and implement many powerful optimizations, which makes C very efficient.

Nim takes advantage of these aspects of C, including its portability, widespread use, and efficiency. Compiling to C also makes it easy to use existing C and C++ libraries—all you need to do is write some simple wrapper code. You can write this code much faster by using a tool called **c2nim**. This tool converts C and C++ header files to Nim code, which wraps those files. This is of great benefit because many popular libraries are written in C and C++.

1.8   Nim's Garbage collector

Switching between garbage collectors is easy. You just need to specify the –gc:<gc_name> flag during compilation and replace <gc_name> with markandsweep, boehm, or none.

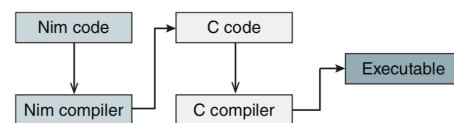## 1.9   Nim's compilation lifecycle

Figure 2: *Compilation lifecycle*

1.10   Nim benefits

1.10.1   Efficiency

1. Nim has performance similar to C.
2. Nim results in software that's more reliable than software written in C.
3. Nim features an improved type system.
4. Nim supports generics.
5. Nim implements an advanced form of metaprogramming.

1.10.2   Where and why I should use Nim

Picking Nim as a C replacement makes a lot of sense.

1.10.3   Nim is clean and readable

Nim class

```nim
#class declaration
type
    Dog = object

#class method
proc bark(self: Dog) = echo("Woof!!")

#instantiation
let dog = Dog()
dog.bark()
```

Python class

```python
class Dog(object): #class
    def bark(self): #class method
        print("Woof!!")

if __name__ == "__main__":
    dog = Dog()  #instantiation
    dog.bark()   #calling class method
```

1.11   Nim simple code snippets

```nim
#simple for loop
let list = ["pineapple", "banana", "apple",
    "orange", "lemon"]

for i in list:
  echo(i)
```

```nim
import strutils

let names = @["John Doe", "John X", "John C"]

var count = 0
for name in names:
    count += 1
    echo("[", count, "] ", name.split[1])

Output:
[1] Doe
[2] X
[3] C
```