Migrating from n6 to n7

Introduction

The source code of n6 was a big mess. I had no experience in compiler construction when I started working on it (it was actually a calculator program that mutated into a programming language). Many years, and thousands of lines of code later, everything felt jammed. There were language features I wanted to add, but it was impossible because of the mess. Trying to refactor the code was simply out of question. In order to develop the language further I knew I had to start from scratch, and finally I did.

The biggest changes in n7 are the following:

- The language is now dynamically (and weakly) typed
- Arrays have been replaced with generic tables
- Functions may now be assigned to variables

Dynamically typed variables

N6 was strongly typed and had no coercion. A variable could not change type at runtime, and there were no implicit conversions between different types of values - comparing an integer with a float would cause a compile time error. In n7, on the other hand, variables may change type at runtime, and the coercion ... has gone pretty extreme, to be honest:

```
foo = 42
foo = 3.14
foo = ″Hello!″
foo = [14, 5, ″dude″, 137]
if foo[1] = ″burp″  pln ″equal″
else  pln ″not equal″
pln foo[2] + 56.7
pln 15 + ″13″
```

Output:

```
not equal
dude56.7
28
```

You can, of course, still use the *int*, *float* and *string* functions to perform explicit conversions. And when needed, you can use the new *typeof* function to tell the type of a variable.

Tables

Tables have been introduced in n7 and replaced arrays. But that doesn't mean that arrays are gone. A table is a set of key and value pairs, and a table where the keys is a sequence of integers can be seen and treated as an array. You can still create arrays pretty much the same way you did in n6:

```
foo = [42, 13, 7, 153]
```

```
bar = [[3, 1], [13, 11], [2, 0]]
```

In these cases, the table keys (indexes) are automatically generated. And you can access the element the same way you did in n6:

```
pln foo[3]
pln bar[1][1] + bar[2][0]
```

Output:

```
153
13
```

But table keys can also be strings, which introduces some new ways of construction and access:

```
foo = []
foo["x"] = 13
foo["y"] = 5

foo = []
foo.x = 13
foo.y = 5

foo = [x: 13, y: 5]
```

These three groups of code do the exact the same thing. They create a table with the keys ”x” and y” and the values 13 and 5.

Another big difference between n6 and n7 is that tables are handled by reference instead of value in n7. Let's look at some n6 code:

```
foo[] = [42, 13, 7, 153]
bar[] = foo
bar[2] = -13
wln "foo[2]: " + str(foo[2])
wln "bar[2]: " + str(bar[2])
```

Output:

```
foo[2]: 7
bar[2]: -13
```

bar becomes a copy of *foo*, a completely new array with the same values as *foo*. Changing *bar* does not change *foo*. But in n7, assigning *foo* to *bar* makes *bar* reference the same table as *foo*:

```
foo = [42, 13, 7, 153]
bar = foo
bar[2] = -13
pln "foo[2]: " + foo[2]
pln "bar[2]: " + bar[2]
```

Output:

```
foo[2]: -13
bar[2]: -13
```

This is more in line with how arrays/tables/objects are handled in other languages. If you really want to create a (deep) copy of a table in n7, you can use the *copy* function:

```
foo = [42, 13, 7, 153]
bar = copy(foo)
bar[2] = -13
pln "foo[2]: " + foo[2]
pln "bar[2]: " + bar[2]
```

Output:

```
foo[2]: 7
bar[2]: -13
```

Functions

N6 had two types of sub-routines: procedures and functions. This distinction is gone in n7, procedures no longer exists. Instead, a function may or may not return a value that the caller may or may not capture.

```
function PrintHelloWorld()
  pln "Hello world!"
endfunc
function Multiply(x, y)
  pln "Multiply!"
  return x*y
endfunc
PrintHelloWorld()
pln PrintHelloWorld()
pln Multiply(7, 2)
Multiply(7, 2)
```

Output:

```
Hello world!
Hello world!
Unset
Multiply!
14
Multiply!
```

The second time we call *PrintHelloWorld* we capture its return value by printing it with *pln*. That's what causes the output "Unset", which is what an unset variable becomes when converted to a string. The first time we call *Multiply* we capture its output, but the second time we don't.

I like to call these functions "static functions" or "named functions". You can assign these functions to variables and call them through the variables:

```
foo = PrintHelloWorld
bar = Multiply
foo()
pln bar(5, 5)
```

Output:

```
Hello world!
Multiply!
25
```

But you may also assign functions directly to variables:

```
foo = function(name)
  pln ”And the winner is ... ” + name + ”!”
endfunc
foo(”Marcus”)
bar = foo
bar(”Dennis”)
```

Output:

```
And the winner is ... Marcus!
And the winner is ... Dennis!
```

You can even define and call a function in an expression:

```
pln 15 + function(x, y); return x*y; endfunc(13.5, 2) + ” is a number!”
```

Output:

```
42 is a number!
```

Look at the examples

This is just an early beta version of n7. There is no documentation, the runtime is slow and lots of stuff is missing. To get an idea of how things work, I suggest you look at the included examples. The source code of the compiler and runtime is included and always will be from now and on.

The compiler and IDE

The IDE, *ned.exe*, is written in n7, and it's an extremely minimalistic thing. You can associate n7 source files (.n7) with it in order to open them easily.

The compiler, *n7.exe*, can be called from the command line with the syntax:

```
n7.exe <source code filename> [win32] [dbg]
```

If you don't use the *win32*-flag, the resulting executable will be a console application. Don't use the *dbg*-flag; if you do, the garbage collector will print information while your program runs and there'll be some output about the memory when your program terminates. You can activate these compiler flags from the source code too. For example, put:

```
#win32
```

somewhere in your program to make sure it's always compiled into a win32 application.

When you compile a program, let's say *MyProgram.n7*, three files are generated: *MyProgram.n7a*, *MyProgram.n7b* and *MyProgram.exe*. The *n7a* and *n7b* files are not needed for the executable (*exe*) to run, so feel free to delete them. The *n7a* file is a normal textfile, just like an *n7* file. It contains an ”assembler” version of your program. It's generated to help me make optimizations later on (trust me, there are many to be made that will increase the execution speed of n7 programs dramatically). The *n7b* file is, kind of, a binary version of the *n7a*, just like the *sbe* files of n6.