Nim Tutorial (Part II) 0.17.2

Andreas Rumpf

January 3, 2018

Contents

1 Introduction

"Repetition renders the ridiculous reasonable." – Norman Wildberger

This document is a tutorial for the advanced constructs of the *Nim* programming language. **Note that this document is somewhat obsolete as the** manual **contains many more examples of the advanced language features.**

2 Pragmas

Pragmas are Nim'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 Nim's support for object oriented programming (OOP) is minimalistic, powerful OOP techniques 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, preferring 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
  Person = ref object of RootObj
    name*: string  # the * means that `name` is accessible from other modules
    age: int       # no * means that the field is hidden from other modules

  Student = ref object of Person # Student inherits from Person
    id: int                      # with an id field

var
  student: Student
  person: Person
assert(student of Student) # is true
# object construction:
student = Student(name: "Anton", age: 5, id: 2)
echo student[]
```

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, `RootObj` can be used as its ancestor, but this is only a convention. Objects that have no ancestor are implicitly `final`. You can use the `inheritable` pragma to introduce new object roots apart from `system.RootObj`. (This is used in the GTK wrapper for instance.)

Ref objects should be used whenever inheritance is used. It isn't strictly necessary, but with non-ref objects assignments such as `let person: Person = Student(id: 123)` will truncate subclass fields.

Note: Composition (*has-a* relation) is often preferable to inheritance (*is-a* relation) for simple code reuse. Since objects are value types in Nim, 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 Nim these types can only be declared within a single type section. (Anything else would require arbitrary symbol lookahead which slows down compilation.)

Example:

```
type
  Node = ref NodeObj # a traced reference to a NodeObj
  NodeObj = object
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

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

3.3 Type conversions

Nim 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: Person): int =
  Student(x).id
```

The `InvalidObjectConversionError` exception is raised if `x` is not a `Student`.

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 modelled in Nim
type
  NodeKind = 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
  Node = ref NodeObj
  NodeObj = object
    case kind: NodeKind  # the ''kind'' field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

var n = Node(kind: nkFloat, floatVal: 1.0)
# the following statement raises an `FieldError` 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?

Nim avoids these problems by not assigning methods to a class. All methods in Nim 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.writeLine("Hallo") # the same as writeLine(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, sequtils

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

3.7 Properties

As the above example shows, Nim 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
  Socket* = ref object of RootObj
    h: int # cannot be accessed from the outside of the module due to missing star

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of host address
  s.h = value

proc host*(s: Socket): int {.inline.} =
  ## getter of host address
  s.h

var s: Socket
new s
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
  Vector* = object
    x, y, z: float

proc `[]=`* (v: var Vector, 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: Vector, 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 RootObj ## 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 it makes more sense for them to 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
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

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

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

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

var a, b: Unit
new a
new b
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 `Unit, Thing` is preferred over `Thing, Unit`.

Performance note: Nim 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 Nim exceptions are objects. By convention, exception types are suffixed with 'Error'. The system module defines an exception hierarchy that you might want to stick to. Exceptions derive from `system.Exception`, 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 OSError
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(OSError, "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: File
if open(f, "numbers.txt"):
  try:
    let a = readLine(f)
    let b = readLine(f)
    echo "sum: ", parseInt(a) + parseInt(b)
  except OverflowError:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    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 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 `IOError`, 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: [IOError, ArithmeticError].} =
  ...

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

5 Generics

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

```
type
  BinaryTreeObj[T] = object # BinaryTree is a generic type with
                            # with generic param ``T``
    le, ri: BinaryTree[T]   # left and right subtrees; may be nil
    data: T                 # the data stored in a node
  BinaryTree*[T] = ref BinaryTreeObj[T] # type that is exported

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

proc add*[T](root: var BinaryTree[T], n: BinaryTree[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 BinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: BinaryTree[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[BinaryTree[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: BinaryTree[string] # instantiate a BinaryTree with ''string''
add(root, newNode("hello")) # instantiates ''newNode'' and ''add''
add(root, "world")          # instantiates the second ''add'' proc
for str in preorder(root):
  stdout.writeLine(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 Nim'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: untyped): untyped =
  # 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.writeLine(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.writeLine(msg)

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

The parameters' types can be ordinary types or the meta types `untyped`, `typed`, or `typedesc`. `typedesc` stands for *type description*, and `untyped` means symbol lookups and type resolution is not performed before the expression is passed to the template.

If the template has no explicit return type, `void` is used for consistency with procs and methods.

To pass a block of statements to a template, use 'untyped' for the last parameter:

```
template withFile(f: untyped, filename: string, mode: FileMode,
                  body: untyped): typed =
  let fn = filename
  var f: File
  if open(f, fn, mode):
    try:
      body
    finally:
      close(f)
  else:
    quit("cannot open: " & fn)

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

In the example the two `writeLine` 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 Nim's syntax. However, this is no real restriction because Nim's syntax is flexible enough anyway. Macros have to be implemented in pure Nim code if the foreign function interface (FFI) is not enabled in the compiler, but other than that restriction (which at some point in the future will go away) you can write any kind of Nim code and the compiler will run it at compile time.

There are two ways to write a macro, either *generating* Nim source code and letting the compiler parse it, or creating manually an abstract syntax tree (AST) which you feed to the compiler. In order to build the AST one needs to know how the Nim concrete syntax is converted to an abstract syntax tree (AST). The AST is documented in the macros module.

Once your macro is finished, there are two ways to invoke it:

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 Nim syntax trees, we need an API that is defined in the
# ``macros`` module:
import macros

macro debug(n: varargs[untyped]): typed =
  # `n` is a Nim AST that contains a list of expressions;
  # this macro returns a list of statements (n is passed for proper line
  # information):
  result = newNimNode(nnkStmtList, n)
  # iterate over any argument that is passed to this macro:
  for x in n:
    # 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(x)))
    # 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("writeLine", newIdentNode("stdout"), x))

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

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

write(stdout, "x")
write(stdout, ": ")
writeLine(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: varargs[untyped]): typed =
  # creates a lexical analyzer from regular expressions
  # ... (implementation is an exercise for the reader :-)
  discard

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 Building your first macro

To give a footstart to writing macros we will show now how to turn your typical dynamic code into something that compiles statically. For the exercise we will use the following snippet of code as the starting point:

```
import strutils, tables

proc readCfgAtRuntime(cfgFilename: string): Table[string, string] =
  let
    inputString = readFile(cfgFilename)
  var
    source = ""

  result = initTable[string, string]()
  for line in inputString.splitLines:
    # Ignore empty lines
    if line.len < 1: continue
    var chunks = split(line, ',')
    if chunks.len != 2:
      quit("Input needs comma split values, got: " & line)
    result[chunks[0]] = chunks[1]

  if result.len < 1: quit("Input file empty!")

let info = readCfgAtRuntime("data.cfg")

when isMainModule:
  echo info["licenseOwner"]
  echo info["licenseKey"]
  echo info["version"]
```

Presumably this snippet of code could be used in a commercial software, reading a configuration file to display information about the person who bought the software. This external file would be generated by an online web shopping cart to be included along the program containing the license information:

```
version,1.1
licenseOwner,Hyori Lee
licenseKey,M1Tl3PjBWO2CC48m
```

The `readCfgAtRuntime` proc will open the given filename and return a `Table` from the tables module. The parsing of the file is done (without much care for handling invalid data or corner cases) using the splitLines proc from the strutils module. There are many things which can fail; mind the purpose is explaining how to make this run at compile time, not how to properly implement a DRM scheme.

The reimplementation of this code as a compile time proc will allow us to get rid of the `data.cfg` file we would need to distribute along the binary, plus if the information is really constant, it doesn't make from a logical point of view to have it *mutable* in a global variable, it would be better if it was a constant. Finally, and likely the most valuable feature, we can implement some verification at compile time. You could think of this as a *better unit testing*, since it is impossible to obtain a binary unless everything is correct, preventing you to ship to users a broken program which won't start because a small critical file is missing or its contents changed by mistake to something invalid.

7.3.1 Generating source code

Our first attempt will start by modifying the program to generate a compile time string with the *generated source code*, which we then pass to the `parseStmt` proc from the macros module. Here is the modified source code implementing the macro:

```
import macros, strutils

macro readCfgAndBuildSource(cfgFilename: string): typed =
  let
    inputString = slurp(cfgFilename.strVal)
  var
    source = ""

  for line in inputString.splitLines:
    # Ignore empty lines
    if line.len < 1: continue
    var chunks = split(line, ',')
```

```
    if chunks.len != 2:
      error("Input needs comma split values, got: " & line)
    source &= "const cfg" & chunks[0] & "= \"" & chunks[1] & "\"\n"

  if source.len < 1: error("Input file empty!")
  result = parseStmt(source)

readCfgAndBuildSource("data.cfg")

when isMainModule:
  echo cfglicenseOwner
  echo cfglicenseKey
  echo cfgversion
```

The good news is not much has changed! First, we need to change the handling of the input parameter (line 3). In the dynamic version the `readCfgAtRuntime` proc receives a string parameter. However, in the macro version it is also declared as string, but this is the *outside* interface of the macro. When the macro is run, it actually gets a `PNimNode` object instead of a string, and we have to call the strVal proc (line 5) from the macros module to obtain the string being passed in to the macro.

Second, we cannot use the readFile proc from the system module due to FFI restriction at compile time. If we try to use this proc, or any other which depends on FFI, the compiler will error with the message `cannot evaluate` and a dump of the macro's source code, along with a stack trace where the compiler reached before bailing out. We can get around this limitation by using the slurp proc from the system module, which was precisely made for compilation time (just like gorge which executes an external program and captures its output).

The interesting thing is that our macro does not return a runtime Table object. Instead, it builds up Nim source code into the `source` variable. For each line of the configuration file a `const` variable will be generated (line 15). To avoid conflicts we prefix these variables with `cfg`. In essence, what the compiler is doing is replacing the line calling the macro with the following snippet of code:

```
const cfgversion= "1.1"
const cfglicenseOwner= "Hyori Lee"
const cfglicenseKey= "M1Tl3PjBWO2CC48m"
```

You can verify this yourself adding the line `echo source` somewhere at the end of the macro and compiling the program. Another difference is that instead of calling the usual quit proc to abort (which we could still call) this version calls the error proc (line 14). The `error` proc has the same behavior as `quit` but will dump also the source and file line information where the error happened, making it easier for the programmer to find where compilation failed. In this situation it would point to the line invoking the macro, but **not** the line of `data.cfg` we are processing, that's something the macro itself would need to control.

7.3.2 Generating AST by hand

To generate an AST we would need to intimately know the structures used by the Nim compiler exposed in the macros module, which at first look seems a daunting task. But we can use as helper shortcut the dumpTree macro, which is used as a statement macro instead of an expression macro. Since we know that we want to generate a bunch of `const` symbols we can create the following source file and compile it to see what the compiler *expects* from us:

```
import macros

dumpTree:
  const cfgversion: string = "1.1"
  const cfglicenseOwner= "Hyori Lee"
  const cfglicenseKey= "M1Tl3PjBWO2CC48m"
```

During compilation of the source code we should see the following lines in the output (again, since this is a macro, compilation is enough, you don't have to run any binary):

```
StmtList
  ConstSection
```

```
    ConstDef
      Ident !"cfgversion"
      Ident !"string"
      StrLit 1.1
  ConstSection
    ConstDef
      Ident !"cfglicenseOwner"
      Empty
      StrLit Hyori Lee
  ConstSection
    ConstDef
      Ident !"cfglicenseKey"
      Empty
      StrLit M1Tl3PjBWO2CC48m
```

With this output we have a better idea of what kind of input the compiler expects. We need to generate a list of statements. For each constant the source code generates a `ConstSection` and a `ConstDef`. If we were to move all the constants to a single `const` block we would see only a single `ConstSection` with three children.

Maybe you didn't notice, but in the `dumpTree` example the first constant explicitly specifies the type of the constant. That's why in the tree output the two last constants have their second child `Empty` but the first has a string identifier. So basically a `const` definition is made up from an identifier, optionally a type (can be an *empty* node) and the value. Armed with this knowledge, let's look at the finished version of the AST building macro:

```
import macros, strutils

macro readCfgAndBuildAST(cfgFilename: string): typed =
  let
    inputString = slurp(cfgFilename.strVal)

  result = newNimNode(nnkStmtList)
  for line in inputString.splitLines:
    # Ignore empty lines
    if line.len < 1: continue
    var chunks = split(line, ',')
    if chunks.len != 2:
      error("Input needs comma split values, got: " & line)
    var
      section = newNimNode(nnkConstSection)
      constDef = newNimNode(nnkConstDef)
    constDef.add(newIdentNode("cfg" & chunks[0]))
    constDef.add(newEmptyNode())
    constDef.add(newStrLitNode(chunks[1]))
    section.add(constDef)
    result.add(section)

  if result.len < 1: error("Input file empty!")

readCfgAndBuildAST("data.cfg")

when isMainModule:
  echo cfglicenseOwner
  echo cfglicenseKey
  echo cfgversion
```

Since we are building on the previous example generating source code, we will only mention the differences to it. Instead of creating a temporary `string` variable and writing into it source code as if it were written *by hand*, we use the `result` variable directly and create a statement list node (`nnkStmtList`) which will hold our children (line 7).

For each input line we have to create a constant definition (`nnkConstDef`) and wrap it inside a constant section (`nnkConstSection`). Once these variables are created, we fill them hierarchichally (line 17) like the previous AST dump tree showed: the constant definition is a child of the section definition, and the constant definition has an identifier node, an empty node (we let the compiler figure out the type), and a string literal with the value.

A last tip when writing a macro: if you are not sure the AST you are building looks ok, you may be tempted to use the `dumpTree` macro. But you can't use it *inside* the macro you are writting/debugging.

Instead `echo` the string generated by treeRepr. If at the end of the this example you add `echo treeRepr(result)` you should get the same output as using the `dumpTree` macro, but of course you can call that at any point of the macro where you might be having troubles.

8 Example Templates and Macros

8.0.1 Lifting Procs

```
import math

template liftScalarProc(fname) =
  ## Lift a proc taking one scalar parameter and returning a
  ## scalar value (eg ``proc sssss[T](x: T): float``),
  ## to provide templated procs that can handle a single
  ## parameter of seq[T] or nested seq[seq[]] or the same type
  ##
  ## .. code-block:: Nim
  ##  liftScalarProc(abs)
  ##  # now abs(@[@[1,-2], @[-2,-3]]) == @[@[1,2], @[2,3]]
  proc fname[T](x: openarray[T]): auto =
    var temp: T
    type outType = type(fname(temp))
    result = newSeq[outType](x.len)
    for i in 0..<x.len:
      result[i] = fname(x[i])

liftScalarProc(sqrt)   # make sqrt() work for sequences
echo sqrt(@[4.0, 16.0, 25.0, 36.0])   # => @[2.0, 4.0, 5.0, 6.0]
```

8.0.2 Identifier Mangling

```
proc echoHW() =
  echo "Hello world"
proc echoHW0() =
  echo "Hello world 0"
proc echoHW1() =
  echo "Hello world 1"

template joinSymbols(a, b: untyped): untyped =
  `a b`()

joinSymbols(echo, HW)

macro str2Call(s1, s2): typed =
  result = newNimNode(nnkStmtList)
  for i in 0..1:
    # combines s1, s2 and an integer into an proc identifier
    # that is called in a statement list
    result.add(newCall(!($s1 & $s2 & $i)))

str2Call("echo", "HW")

# Output:
#   Hello world
#   Hello world 0
#   Hello world 1
```

9 Compilation to JavaScript

Nim code can be compiled to JavaScript. However in order to write JavaScript-compatible code you should remember the following:

- `addr` and `ptr` have slightly different semantic meaning in JavaScript. It is recommended to avoid those if you're not sure how they are translated to JavaScript.

- `cast[T](x)` in JavaScript is translated to `(x)`, except for casting between signed/unsigned ints, in which case it behaves as static cast in C language.
- `cstring` in JavaScript means JavaScript string. It is a good practice to use `cstring` only when it is semantically appropriate. E.g. don't use `cstring` as a binary data buffer.