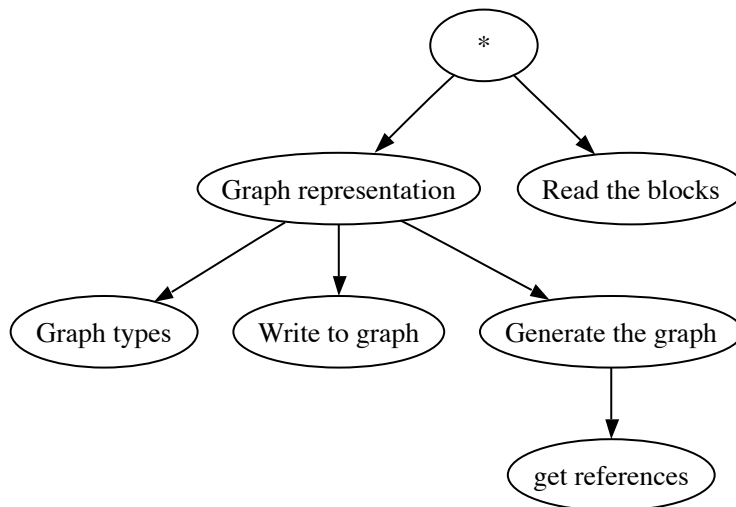


1 Generating a graph from a given literate program

In the following document, I show how to generate a graph that shows which code chunks were referenced from which other chunks. The aim is to visualize the usual structure of a literate program.

For example, this document has the following structure:



1.1 Overview

I will receive input in the block format as described in `markup/blocks.nw`, which I will then convert into the `dot` format¹. Also, some data types will be needed for the intermediary representation:

`<*>` ≡

package visualization

<Read the blocks>

<Graph representation>

¹See <http://www.graphviz.org/doc/info/lang.html>

1.2 Reading the blocks

The reading of the blocks will use the utility functions defined in `util/commandline.nw` for option parsing etc. This is pretty basic stuff:

⟨Read the blocks⟩ ≡

```
object MakeGraph {
  def main(args: Array[String]) = {
    import util.LiterateSettings
    val ls = new LiterateSettings(args)
```

This defined a new literate setting class which takes automatically care of some basic parameters for literate program input. We will use two fields: `chunkCollections` and `output`

⟨Read the blocks⟩ + ≡

```
val graph = new Graph(ls.chunkCollections)
```

First, we generate a new graph class from the chunks that we read in, then we write it to the specified output:

⟨Read the blocks⟩ + ≡

```
    graph.write(ls.output)
  }
}
```

1.3 Graph representation

Our graph will basically be a tree (as no code chunk will be referenced twice). A nice implementation of this is using a set of direct descendants (also note that each chunk has only one parent):

⟨Graph types⟩ ≡

```

import scala.collection.mutable.HashMap
type Node = String
type Graph = HashMap[Node,List[Node]]

```

With this information, we can define the main class. It contains a representation of the graph and a method to write this graph to a dot file (explained in the next section). Also, there is a method to generate the actual graph, which will be invoked in the body

<Graph representation> ≡

```

import tangle.ChunkCollection
class Graph(chunks: List[ChunkCollection]) {
  <Graph types>
  val representation: Graph =
    new HashMap[Node,List[Node]]()
  <Generate the graph>
  <Write to graph>
}

```

Now on how to generate the graph. For this, we will traverse the chunk collection, adding elements as we go along. Then, for each element, we add the links to its other blocks:

<Generate the graph> ≡

```

generate()
def generate(): Unit = {
  chunks foreach {
    cc => cc.cm.keys foreach {
      chunkName =>
        representation(chunkName) =
          getReferences(cc.cm(chunkName), cc.cm)
    }
  }
}
<get references>

```

To get the references, we look at the code in string reference form, adding an edge for each block reference:

$\langle \text{get references} \rangle \equiv$

```
import tangle.CodeChunk
import markup.CodeBlock
def getReferences(chunk: CodeChunk, cm: Map[String,CodeBlock]) = {
  import markup.StringRefs._
```

First, we filter out all the non-reference elements:

$\langle \text{get references} \rangle + \equiv$

```
(chunk.stringRefForm(cm) filter {
  case BlockRef(_)  $\Rightarrow$  true
  case _  $\Rightarrow$  false
```

Then we return the names of these blocks:

$\langle \text{get references} \rangle + \equiv$

```
  } map {
    case BlockRef(r)  $\Rightarrow$  r.blockname
    case _  $\Rightarrow$  error("This should not contain any other elements")
  }).toList
}
```

1.4 Writing the graph

The dot format expects a listing of nodes and edges to build the graph. Because we will use the nodes quite a few times, we give them shorter names (i.e. numbers)

$\langle \text{Write to graph} \rangle \equiv$

```

def write(out: java.io.PrintStream) = {
  val namesToNumbers: Map[Node,Int] = Map() ++
    (representation.keys.toList
     zip
     List.range(1,representation.size + 1))

```

With this in place, we are ready to print the header: $\langle \textit{Write to graph} \rangle + \equiv$

```

out.println("digraph {")

```

Now, we print the list of nodes with their label:

$\langle \textit{Write to graph} \rangle + \equiv$

```

namesToNumbers foreach {
  case (name,number) =>
    out.println("n" + number + " [ label =\"" +
               name
               + "\""]")
}

```

After that, we list all the edges:

$\langle \textit{Write to graph} \rangle + \equiv$

```

namesToNumbers foreach {
  case (name,number) =>
    representation(name) foreach {
      targetName =>
        val targetNumber =
          namesToNumbers(targetName)
        out.println("n" + number +
                   " → n" + targetNumber +
                   ";")
    }
}

```

Finally, we are finished with printing:

$\langle \textit{Write to graph} \rangle + \equiv$

```
        out.println("}")
    }
```

Definitions

- Graph
 - Class definition: 3
 - Method generate: 3
 - Method getReferences: 4
 - Method write: 5
 - Value chunks: 3
 - Value representation : 3
- MakeGraph
 - Object definition: 2
 - Method main: 2