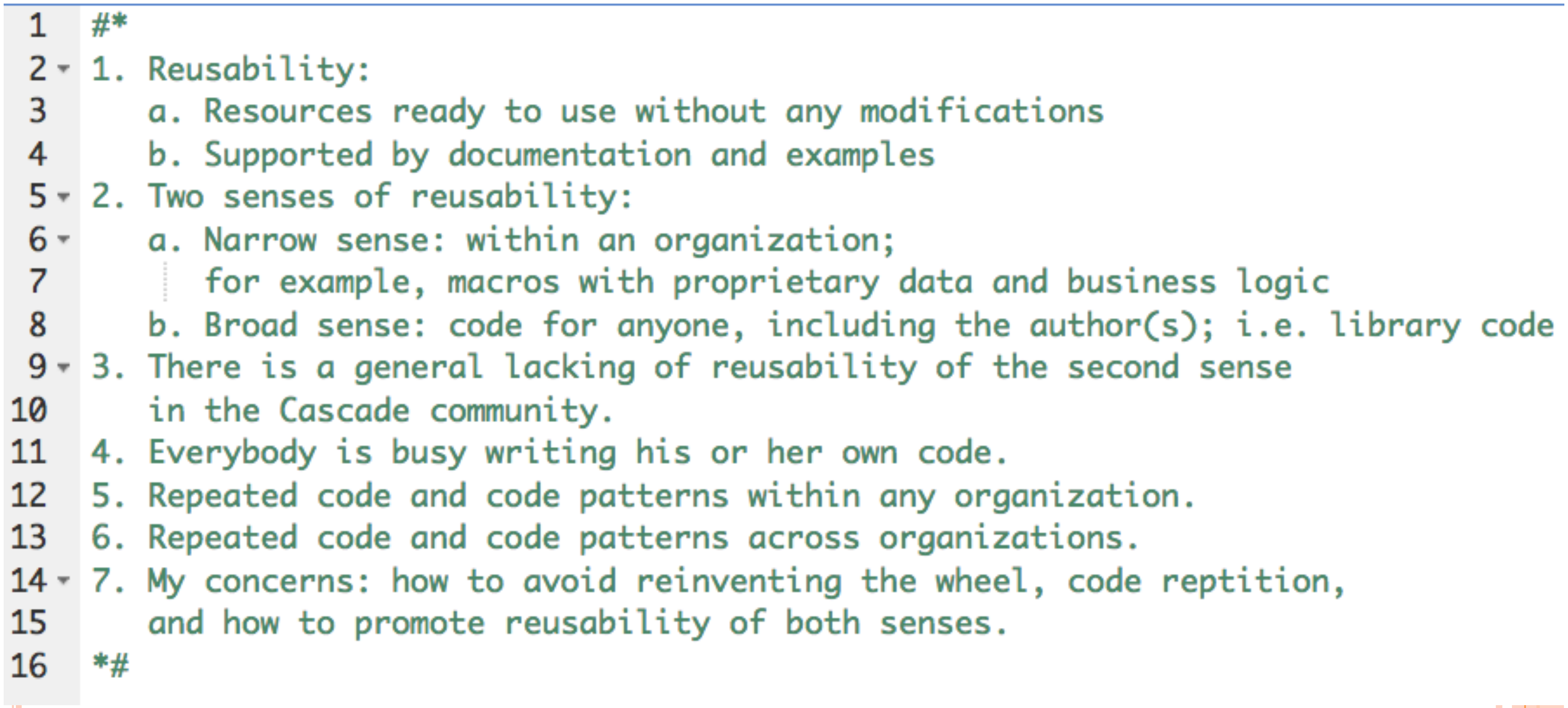
REUSABILITY AND VELOCITY CODE TEMPLATES

#CUC16

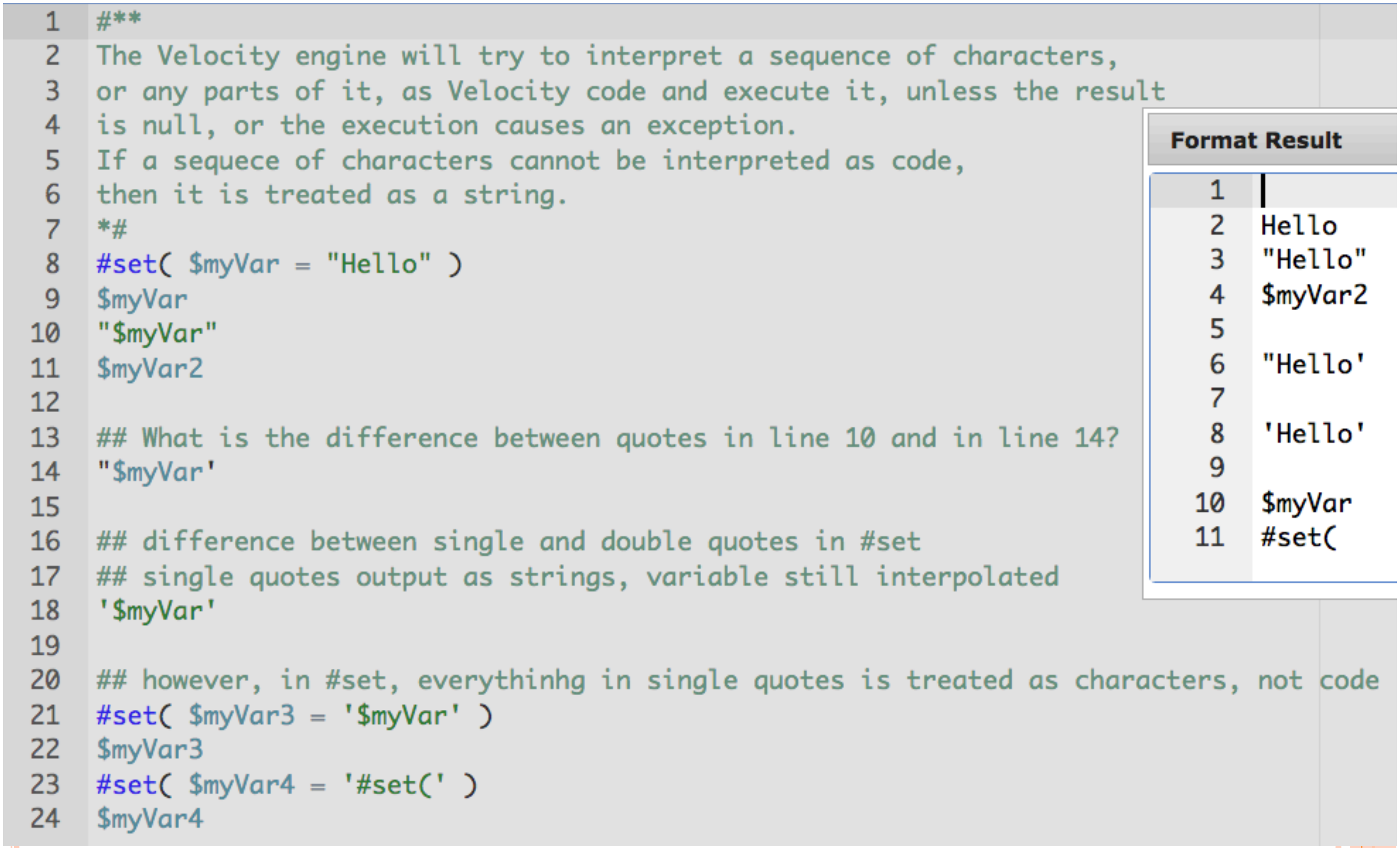
Presented by
Wing Ming Chan, September 19, 2016
Upstate Medical University
chanw@upstate.edu

TOPICS

- Two meanings of reusability
  - Generic library
  - Proprietary reusable code
- Code and code pattern repetition
- What are code templates?
- Why code templates?
- How code templates promote reusability



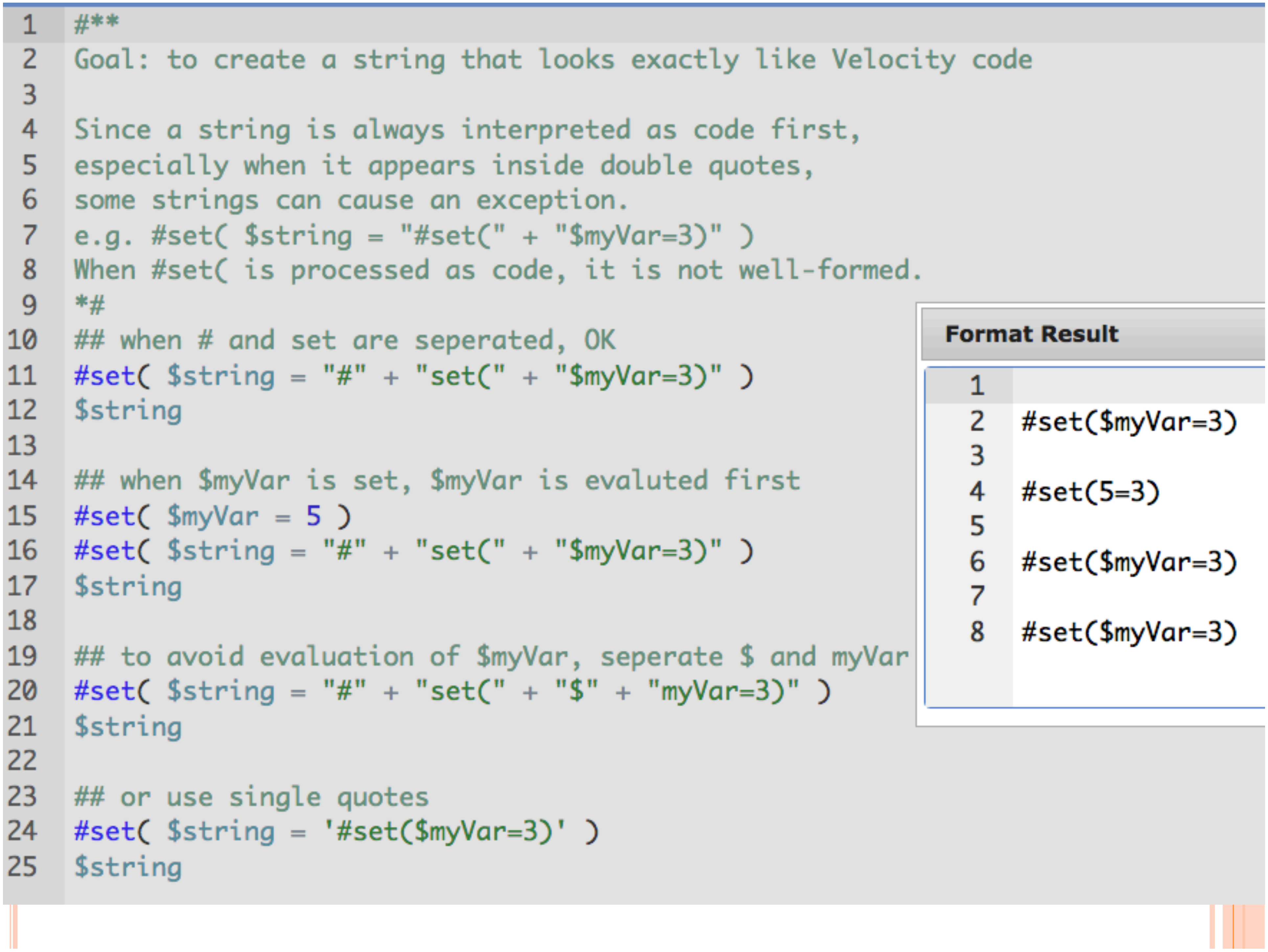
```
#*
1. Reusability:
   a. Resources ready to use without any modifications
   b. Supported by documentation and examples
2. Two senses of reusability:
   a. Narrow sense: within an organization;
      for example, macros with proprietary data and business logic
   b. Broad sense: code for anyone, including the author(s); i.e. library code
3. There is a general lacking of reusability of the second sense
   in the Cascade community.
4. Everybody is busy writing his or her own code.
5. Repeated code and code patterns within any organization.
6. Repeated code and code patterns across organizations.
7. My concerns: how to avoid reinventing the wheel, code reptition,
   and how to promote reusability of both senses.
*#
```



```
1  #**
2  The Velocity engine will try to interpret a sequence of characters,
3  or any parts of it, as Velocity code and execute it, unless the result
4  is null, or the execution causes an exception.
5  If a sequece of characters cannot be interpreted as code,
6  then it is treated as a string.
7  *#
8  #set( $myVar = "Hello" )
9  $myVar
10 "$myVar"
11 $myVar2
12
13 ## What is the difference between quotes in line 10 and in line 14?
14 "$myVar'
15
16 ## difference between single and double quotes in #set
17 ## single quotes output as strings, variable still interpolated
18 '$myVar'
19
20 ## however, in #set, everythinhg in single quotes is treated as characters, not code
21 #set( $myVar3 = '$myVar' )
22 $myVar3
23 #set( $myVar4 = '#set(' )
24 $myVar4
```

Format Result

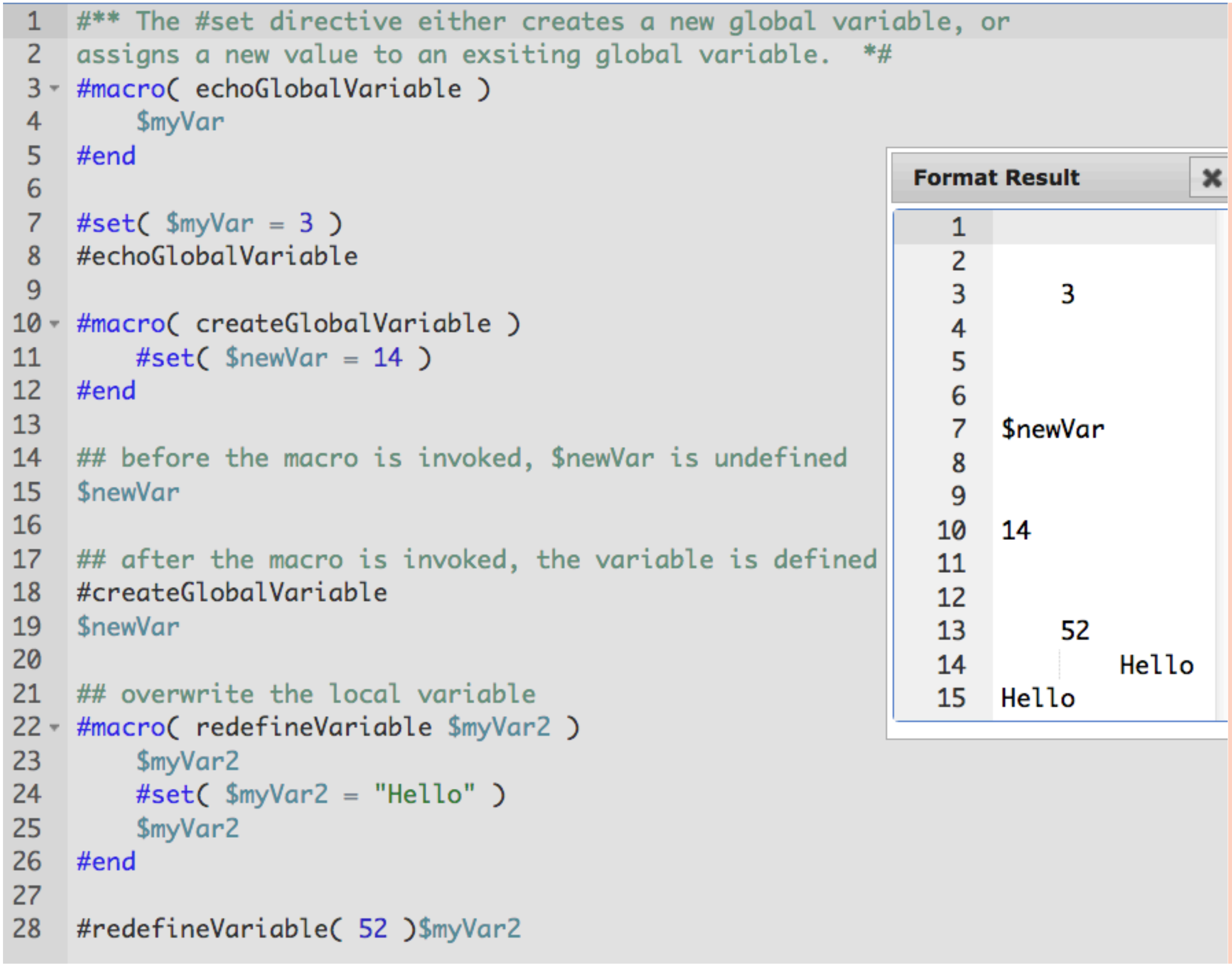
```

Hello
"Hello"
$myVar2

"Hello'

'Hello'

$myVar
#set(
```



```
#**
Goal: to create a string that looks exactly like Velocity code

Since a string is always interpreted as code first,
especially when it appears inside double quotes,
some strings can cause an exception.
e.g. #set( $string = "#set(" + "$myVar=3)" )
When #set( is processed as code, it is not well-formed.
*#
## when # and set are seperated, OK
#set( $string = "#" + "set(" + "$myVar=3)" )
$string

## when $myVar is set, $myVar is evaluted first
#set( $myVar = 5 )
#set( $string = "#" + "set(" + "$myVar=3)" )
$string

## to avoid evaluation of $myVar, seperate $ and myVar
#set( $string = "#" + "set(" + "$" + "myVar=3)" )
$string

## or use single quotes
#set( $string = '#set($myVar=3)' )
$string
```

Format Result

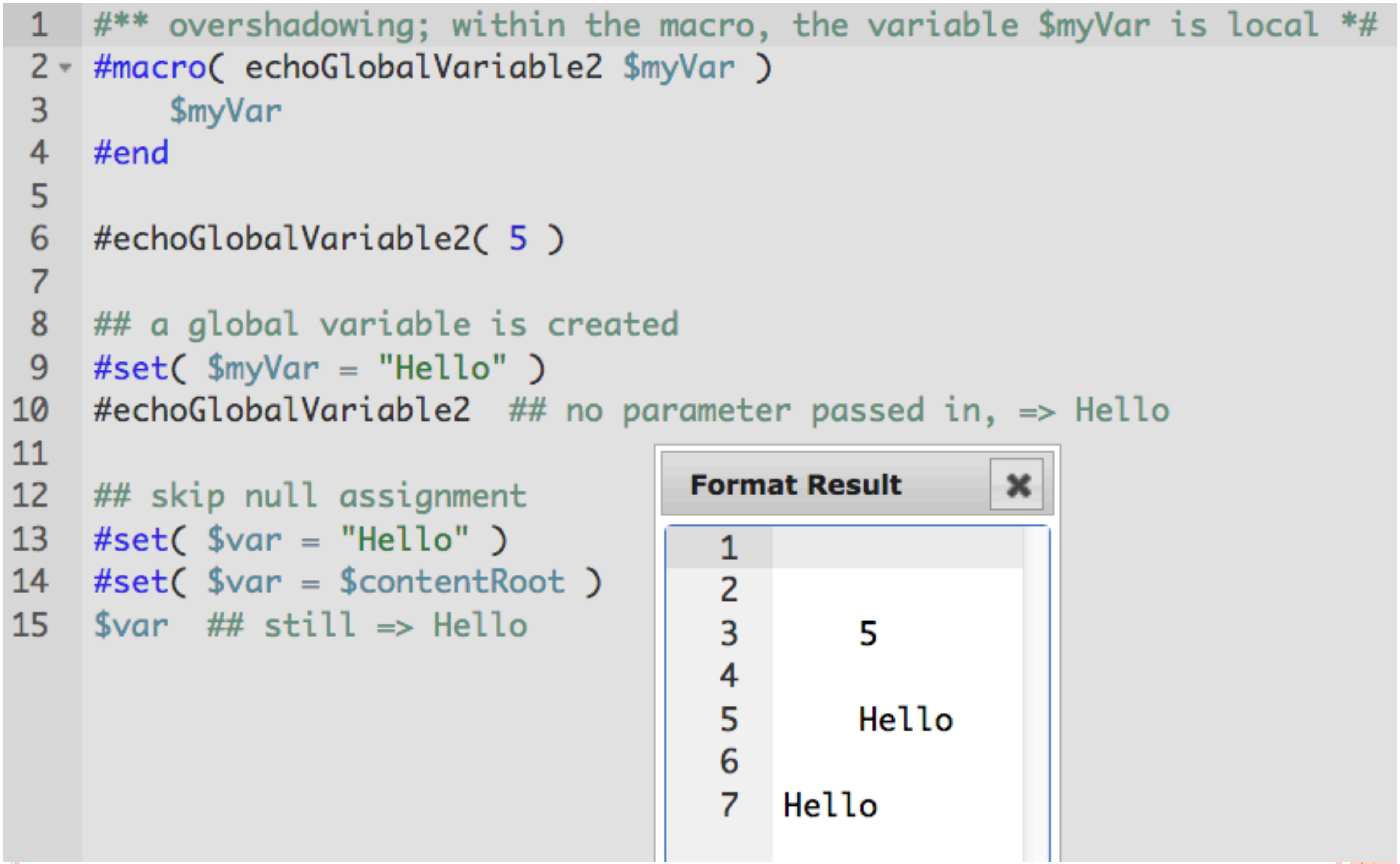
```

#set($myVar=3)

#set(5=3)

#set($myVar=3)

#set($myVar=3)
```



```
#** The #set directive either creates a new global variable, or
assigns a new value to an exsiting global variable.  *#
#macro( echoGlobalVariable )
    $myVar
#end

#set( $myVar = 3 )
#echoGlobalVariable

#macro( createGlobalVariable )
    #set( $newVar = 14 )
#end

## before the macro is invoked, $newVar is undefined
$newVar

## after the macro is invoked, the variable is defined
#createGlobalVariable
$newVar

## overwrite the local variable
#macro( redefineVariable $myVar2 )
    $myVar2
    #set( $myVar2 = "Hello" )
    $myVar2
#end

#redefineVariable( 52 )$myVar2
```

Format Result

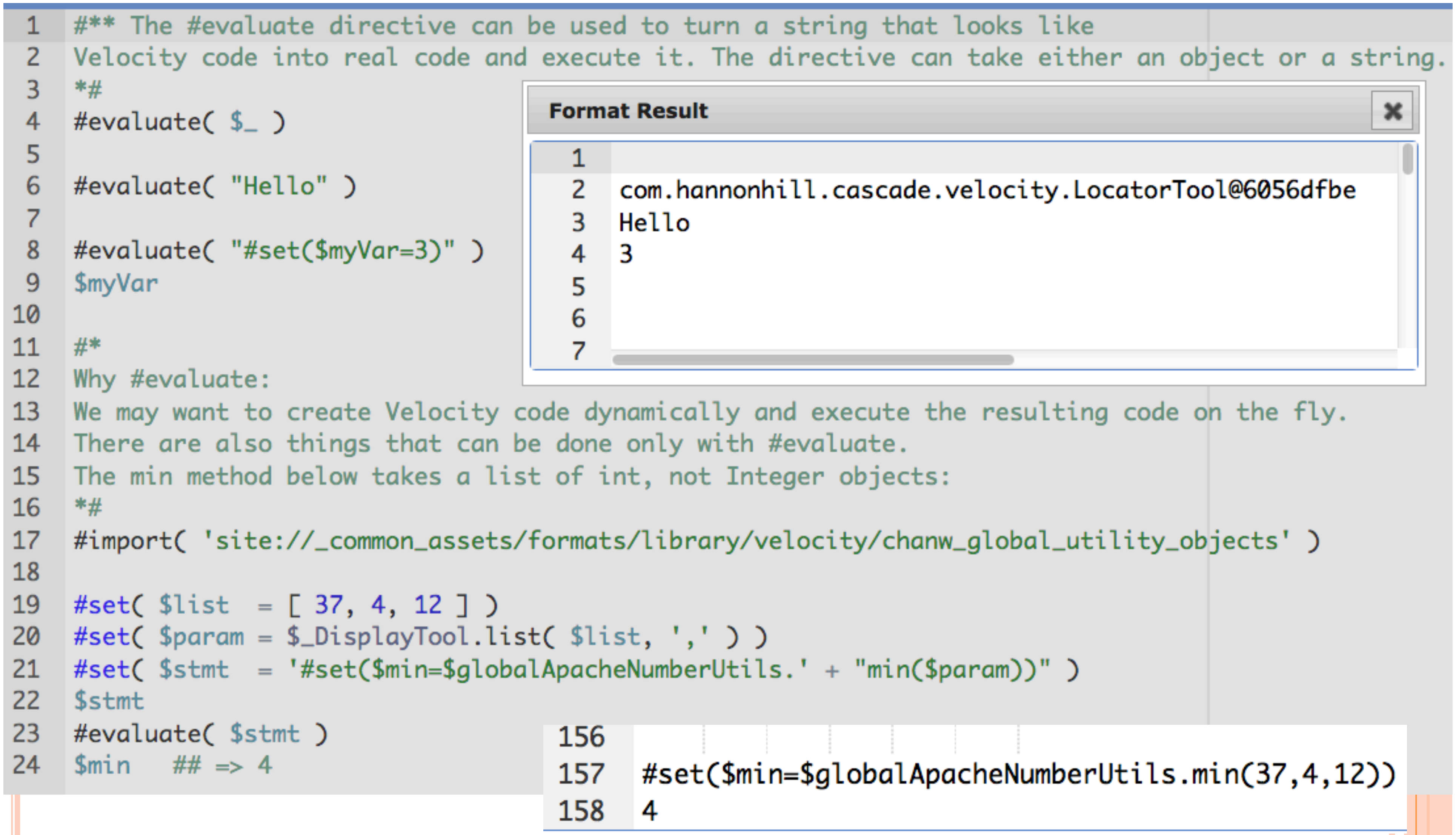
```


    3



$newVar


14


    52
        Hello
Hello
```



```
#** overshadowing; within the macro, the variable $myVar is local *#
#macro( echoGlobalVariable2 $myVar )
    $myVar
#end

#echoGlobalVariable2( 5 )

## a global variable is created
#set( $myVar = "Hello" )
#echoGlobalVariable2  ## no parameter passed in, => Hello

## skip null assignment
#set( $var = "Hello" )
#set( $var = $contentRoot )
$var  ## still => Hello
```

Format Result

```

        5

        Hello

Hello
```

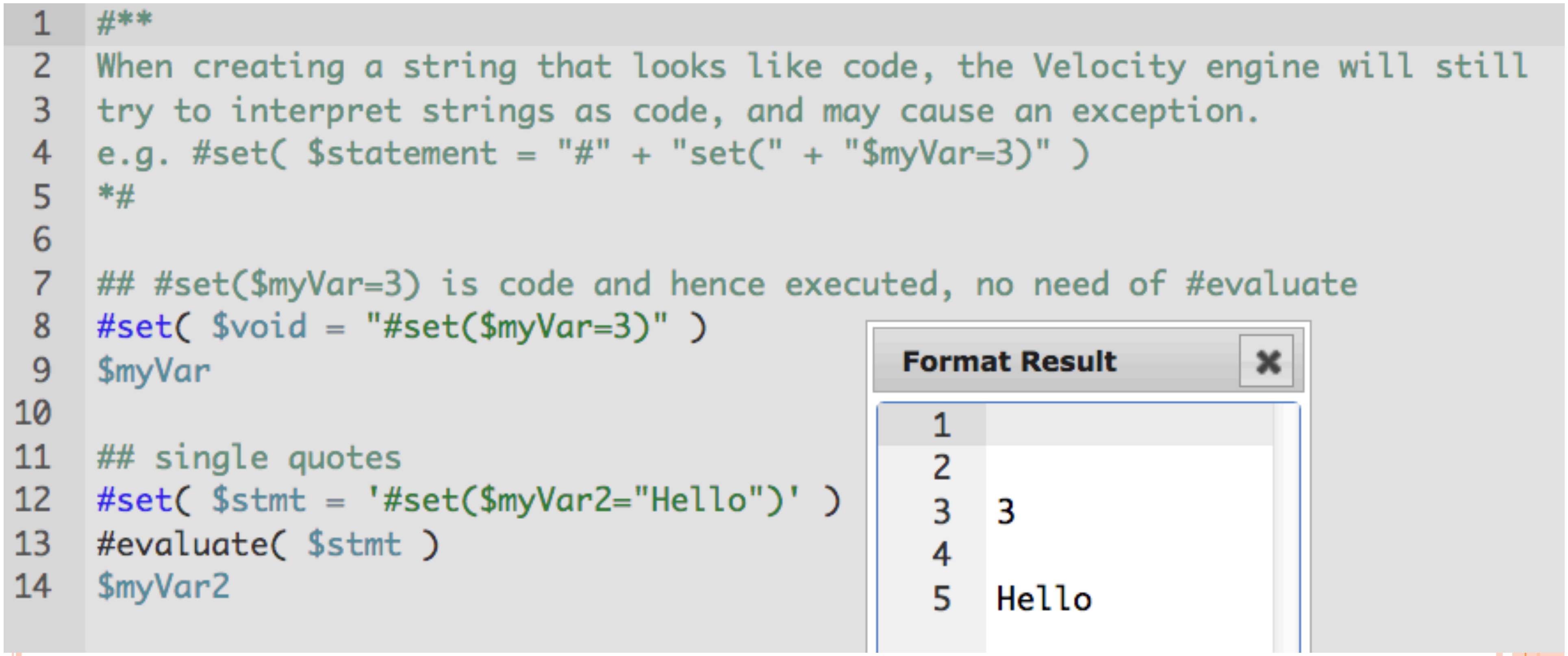


```
#** The #evaluate directive can be used to turn a string that looks like
Velocity code into real code and execute it. The directive can take either an object or a string.
*#
#evaluate( $_ )

#evaluate( "Hello" )

#evaluate( "#set($myVar=3)" )
$myVar

#*
Why #evaluate:
We may want to create Velocity code dynamically and execute the resulting code on the fly.
There are also things that can be done only with #evaluate.
The min method below takes a list of int, not Integer objects:
*#
#import( 'site://_common_assets/formats/library/velocity/chanw_global_utility_objects' )

#set( $list  = [ 37, 4, 12 ] )
#set( $param = $_DisplayTool.list( $list, ',' ) )
#set( $stmt  = '#set($min=$globalApacheNumberUtils.' + "min($param))" )
$stmt
#evaluate( $stmt )
$min   ## => 4
```

Format Result

```

com.hannonhill.cascade.velocity.LocatorTool@6056dfbe
Hello
3



```

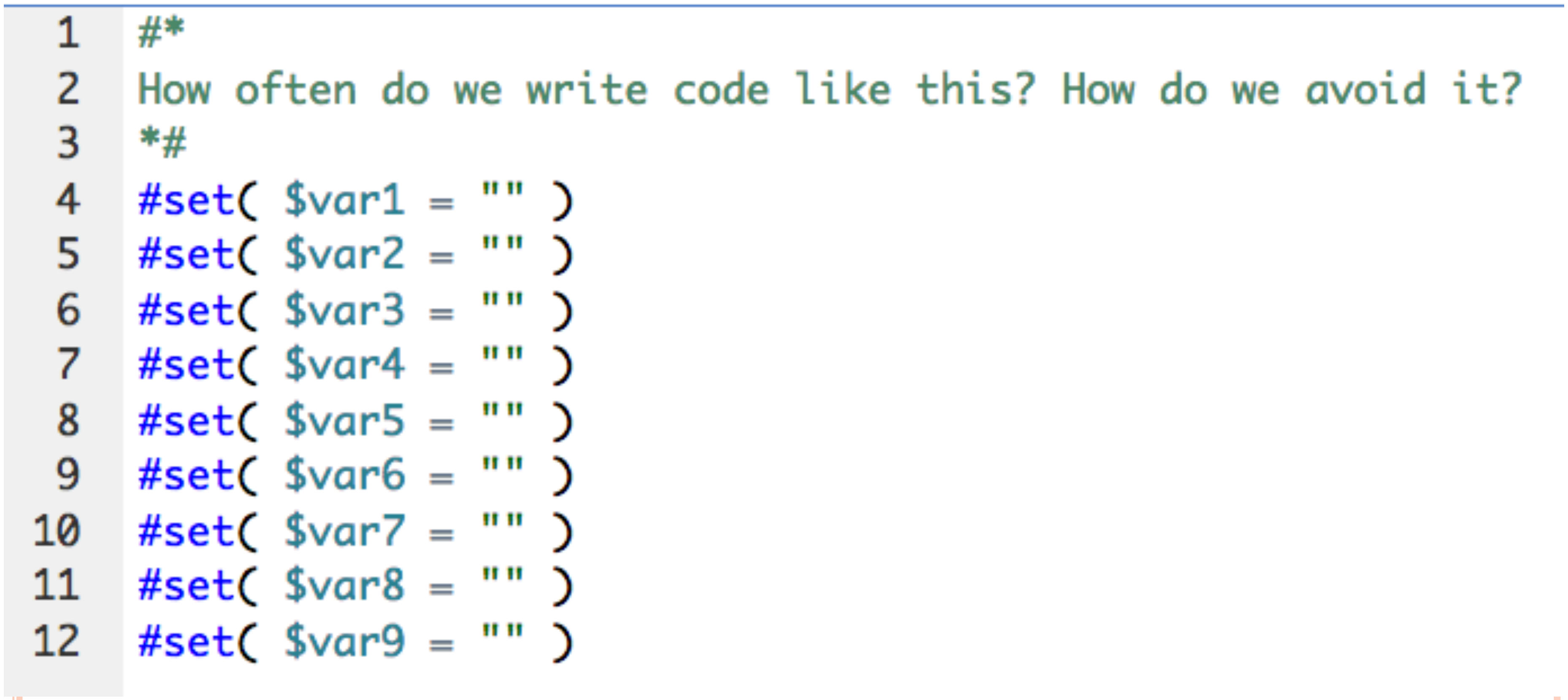
```
156
157 #set($min=$globalApacheNumberUtils.min(37,4,12))
158 4
```



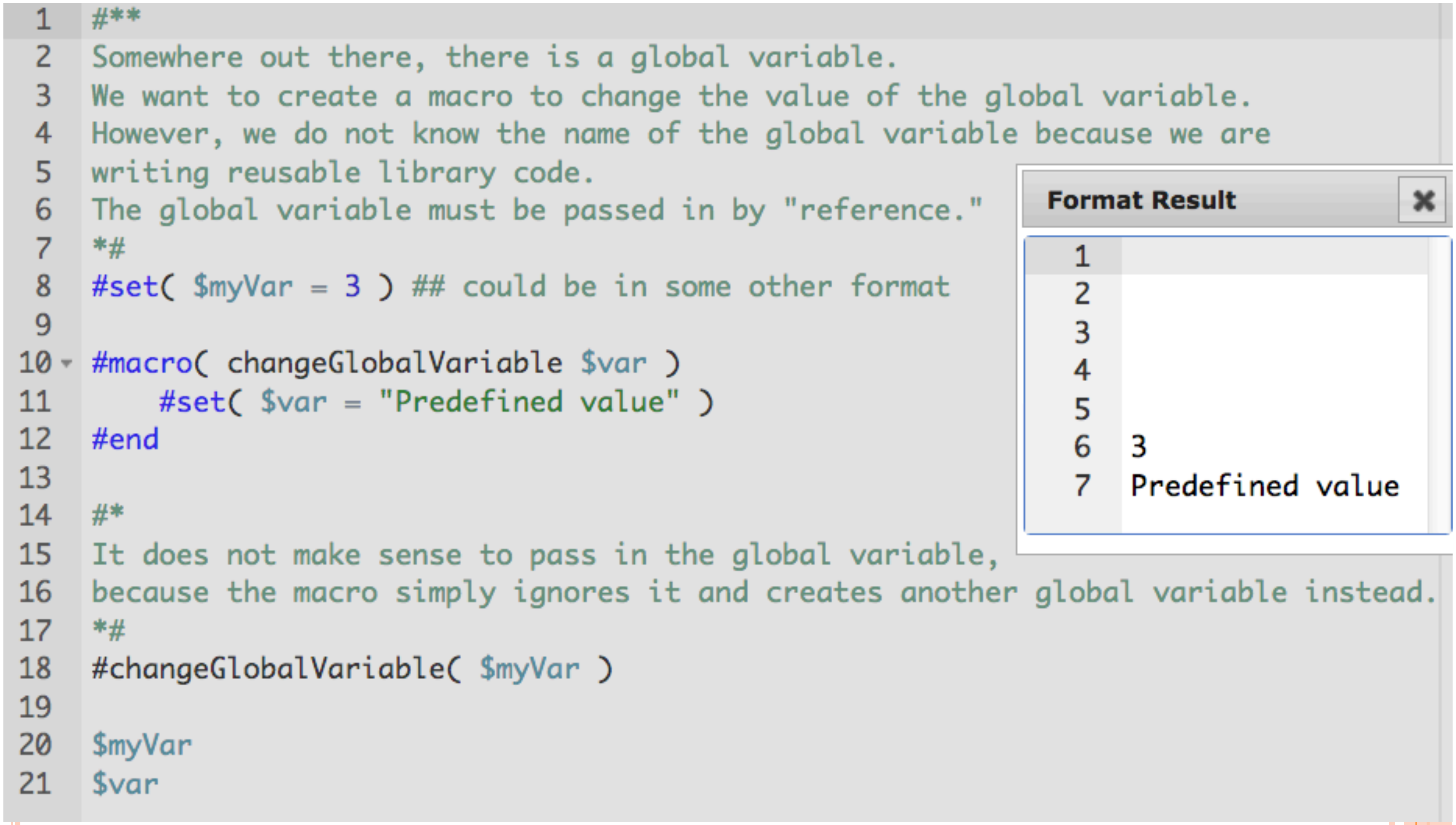
```
#**
When creating a string that looks like code, the Velocity engine will still
try to interpret strings as code, and may cause an exception.
e.g. #set( $statement = "#" + "set(" + "$myVar=3)" )
*#

## #set($myVar=3) is code and hence executed, no need of #evaluate
#set( $void = "#set($myVar=3)" )
$myVar

## single quotes
#set( $stmt = '#set($myVar2="Hello")' )
#evaluate( $stmt )
$myVar2
```

Format Result

```


3

Hello
```



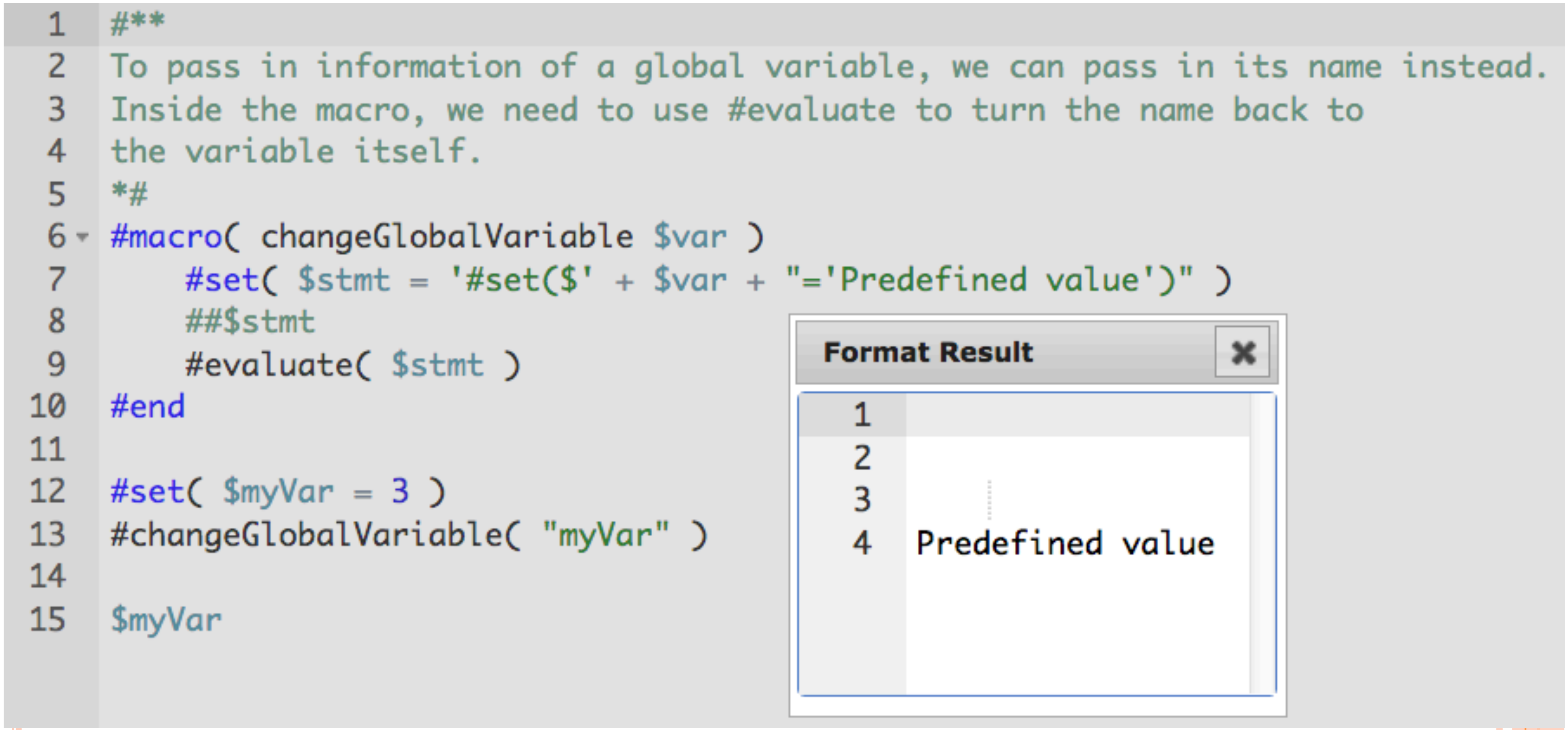
```
#*
How often do we write code like this? How do we avoid it?
*#
#set( $var1 = "" )
#set( $var2 = "" )
#set( $var3 = "" )
#set( $var4 = "" )
#set( $var5 = "" )
#set( $var6 = "" )
#set( $var7 = "" )
#set( $var8 = "" )
#set( $var9 = "" )
```



```
#**
Somewhere out there, there is a global variable.
We want to create a macro to change the value of the global variable.
However, we do not know the name of the global variable because we are
writing reusable library code.
The global variable must be passed in by "reference."
*#
#set( $myVar = 3 ) ## could be in some other format

#macro( changeGlobalVariable $var )
    #set( $var = "Predefined value" )
#end

#*
It does not make sense to pass in the global variable,
because the macro simply ignores it and creates another global variable instead.
*#
#changeGlobalVariable( $myVar )

$myVar
$var
```

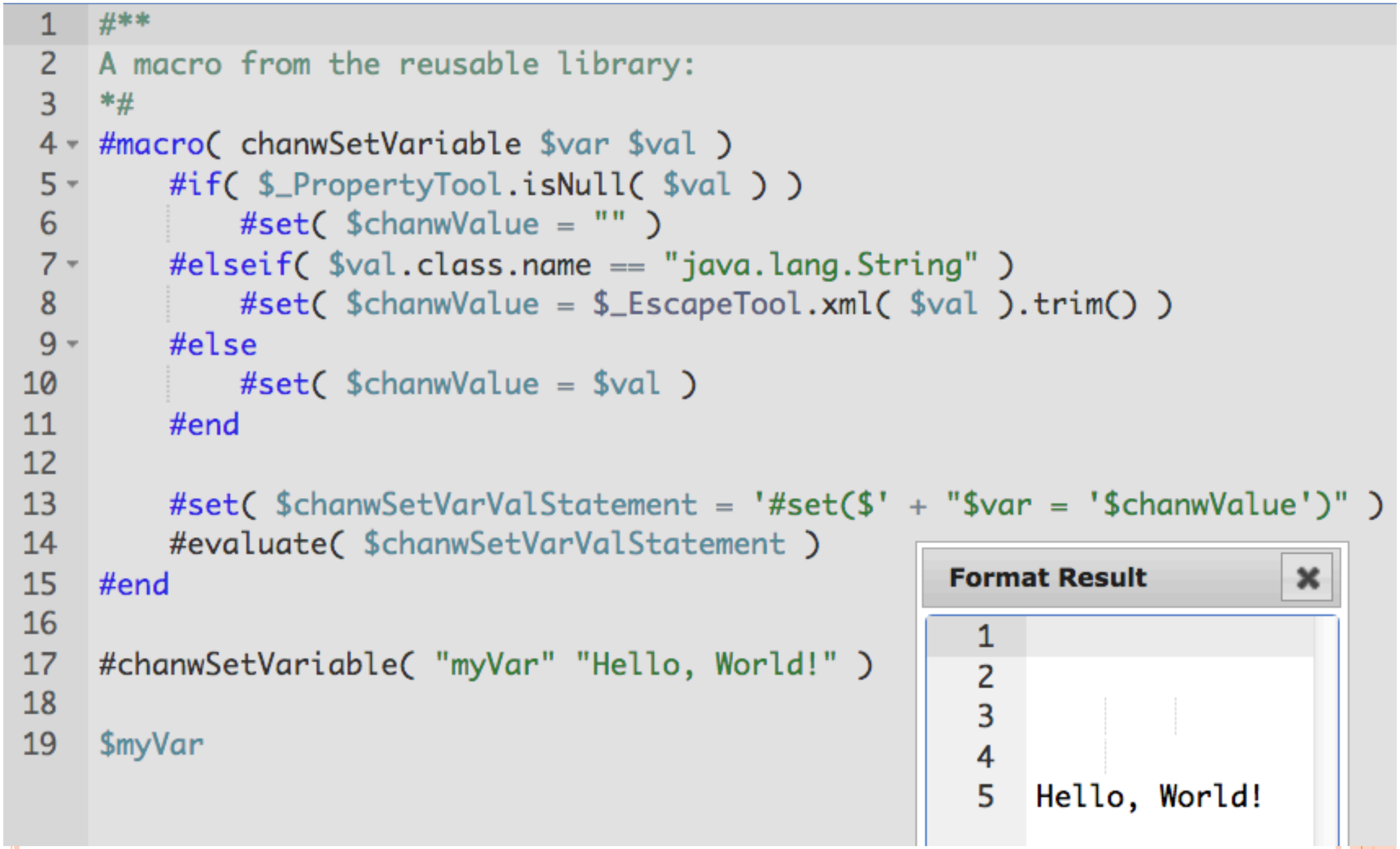
Format Result

```





3
Predefined value
```



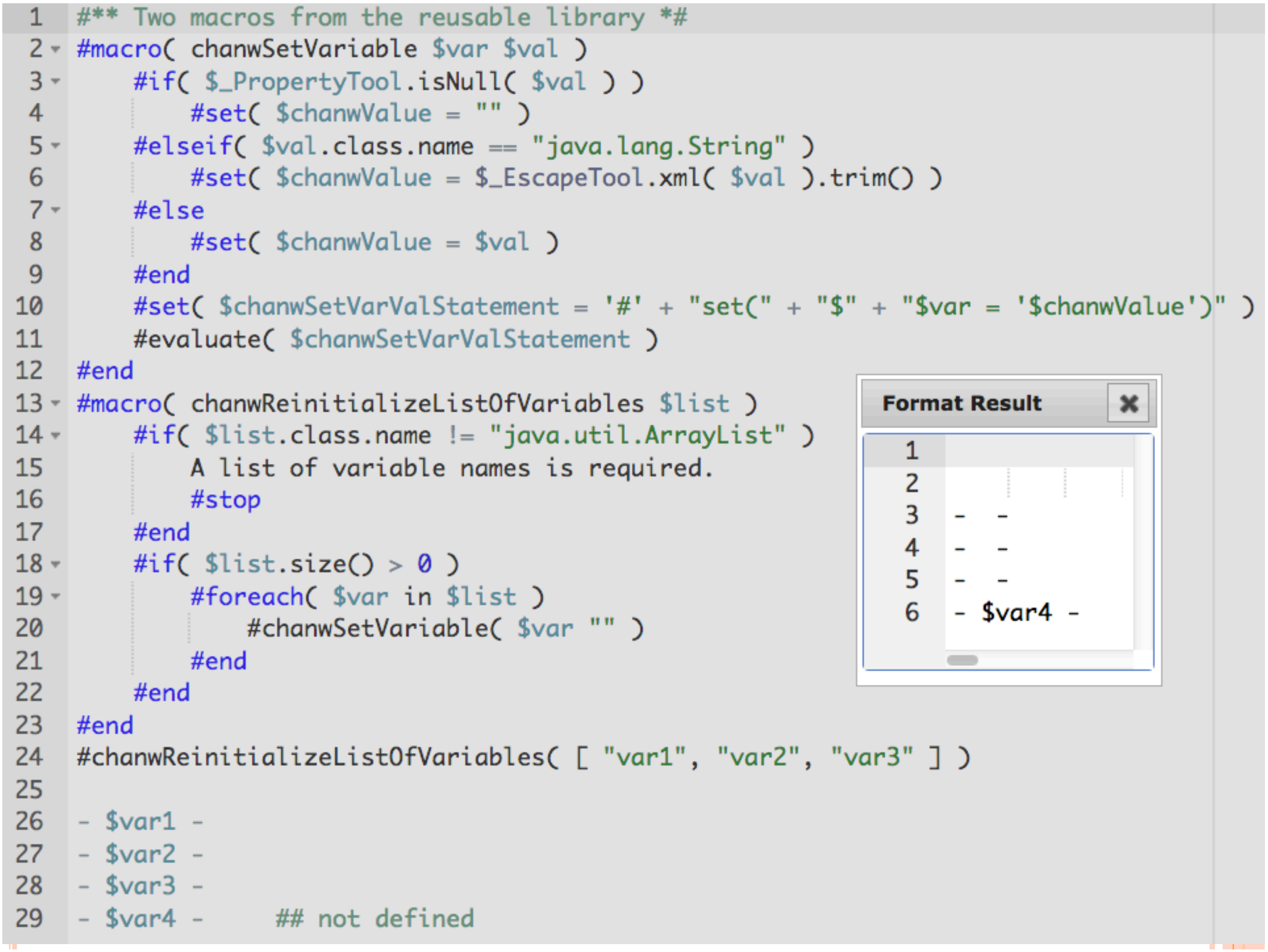
```
#**
To pass in information of a global variable, we can pass in its name instead.
Inside the macro, we need to use #evaluate to turn the name back to
the variable itself.
*#
#macro( changeGlobalVariable $var )
    #set( $stmt = '#set($' + $var + "='Predefined value')" )
    ##$stmt
    #evaluate( $stmt )
#end

#set( $myVar = 3 )
#changeGlobalVariable( "myVar" )

$myVar
```

```



Predefined value
```



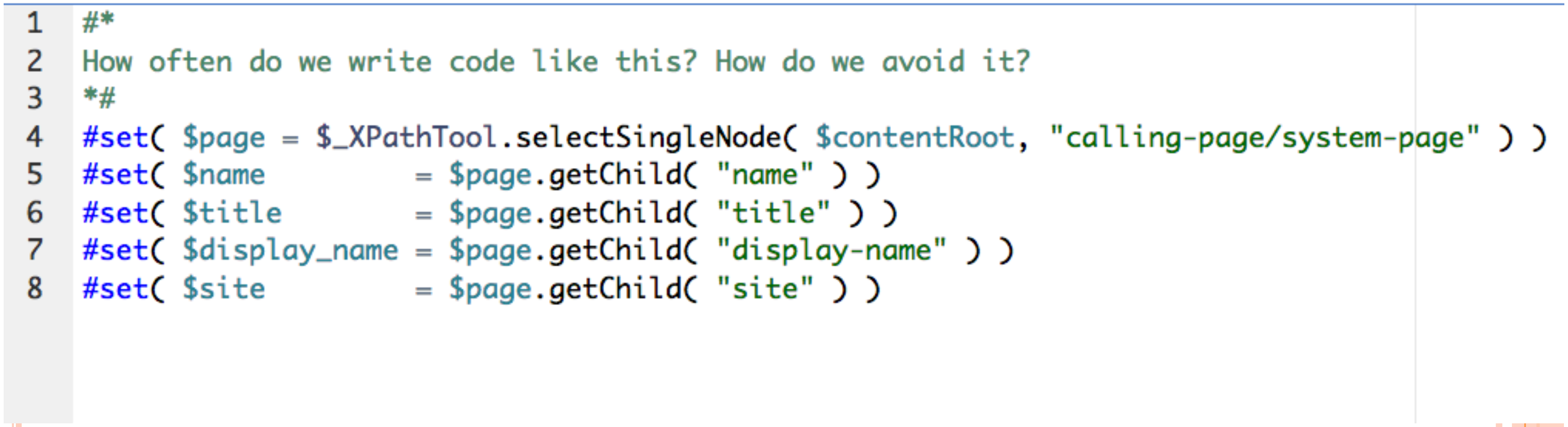
```
#**
A macro from the reusable library:
*#
#macro( chanwSetVariable $var $val )
    #if( $_PropertyTool.isNull( $val ) )
        #set( $chanwValue = "" )
    #elseif( $val.class.name == "java.lang.String" )
        #set( $chanwValue = $_EscapeTool.xml( $val ).trim() )
    #else
        #set( $chanwValue = $val )
    #end

    #set( $chanwSetVarValStatement = '#set($' + "$var = '$chanwValue')" )
    #evaluate( $chanwSetVarValStatement )
#end

#chanwSetVariable( "myVar" "Hello, World!" )

$myVar
```

```




Hello, World!
```

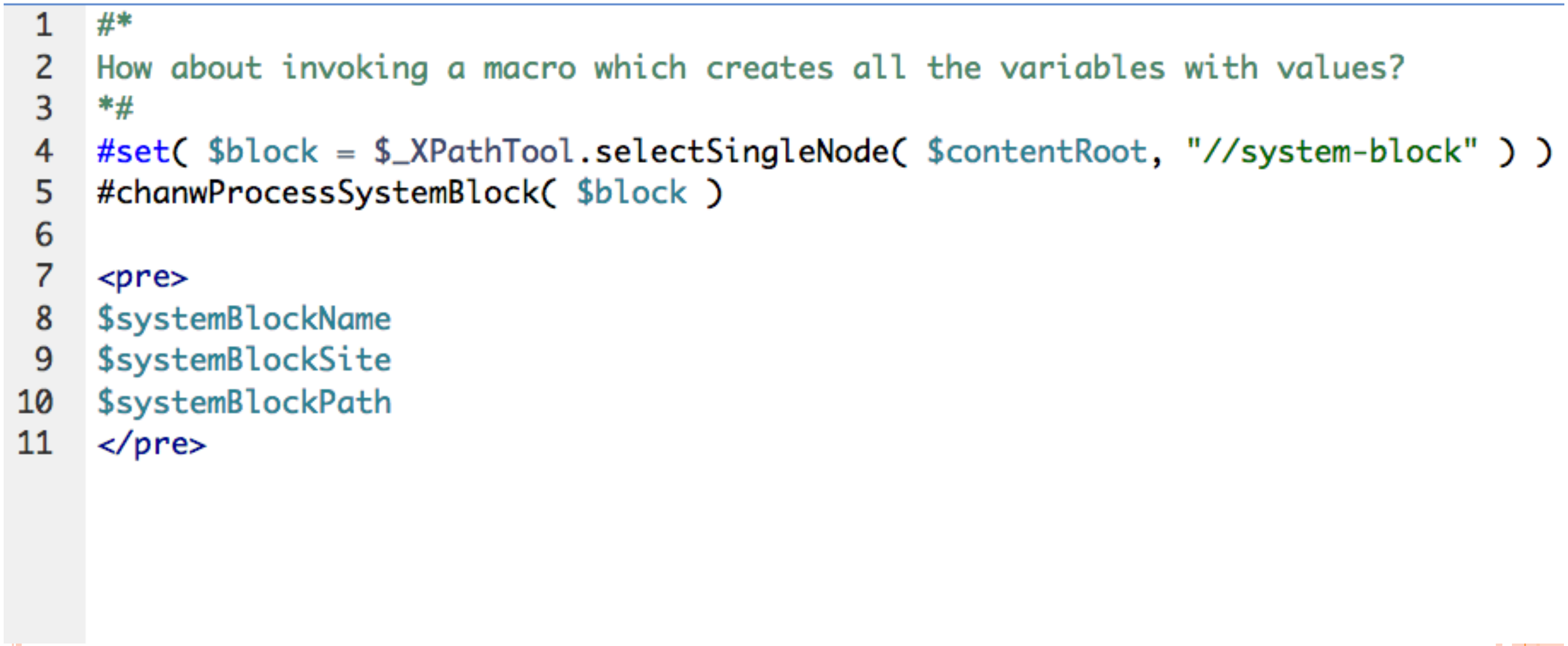


```
#** Two macros from the reusable library *#
#macro( chanwSetVariable $var $val )
    #if( $_PropertyTool.isNull( $val ) )
        #set( $chanwValue = "" )
    #elseif( $val.class.name == "java.lang.String" )
        #set( $chanwValue = $_EscapeTool.xml( $val ).trim() )
    #else
        #set( $chanwValue = $val )
    #end
    #set( $chanwSetVarValStatement = '#' + "set(" + "$" + "$var = '$chanwValue')" )
    #evaluate( $chanwSetVarValStatement )
#end
#macro( chanwReinitializeListOfVariables $list )
    #if( $list.class.name != "java.util.ArrayList" )
        A list of variable names is required.
        #stop
    #end
    #if( $list.size() > 0 )
        #foreach( $var in $list )
            #chanwSetVariable( $var "" )
        #end
    #end
#end
#chanwReinitializeListOfVariables( [ "var1", "var2", "var3" ] )

- $var1 -
- $var2 -
- $var3 -
- $var4 -      ## not defined
```

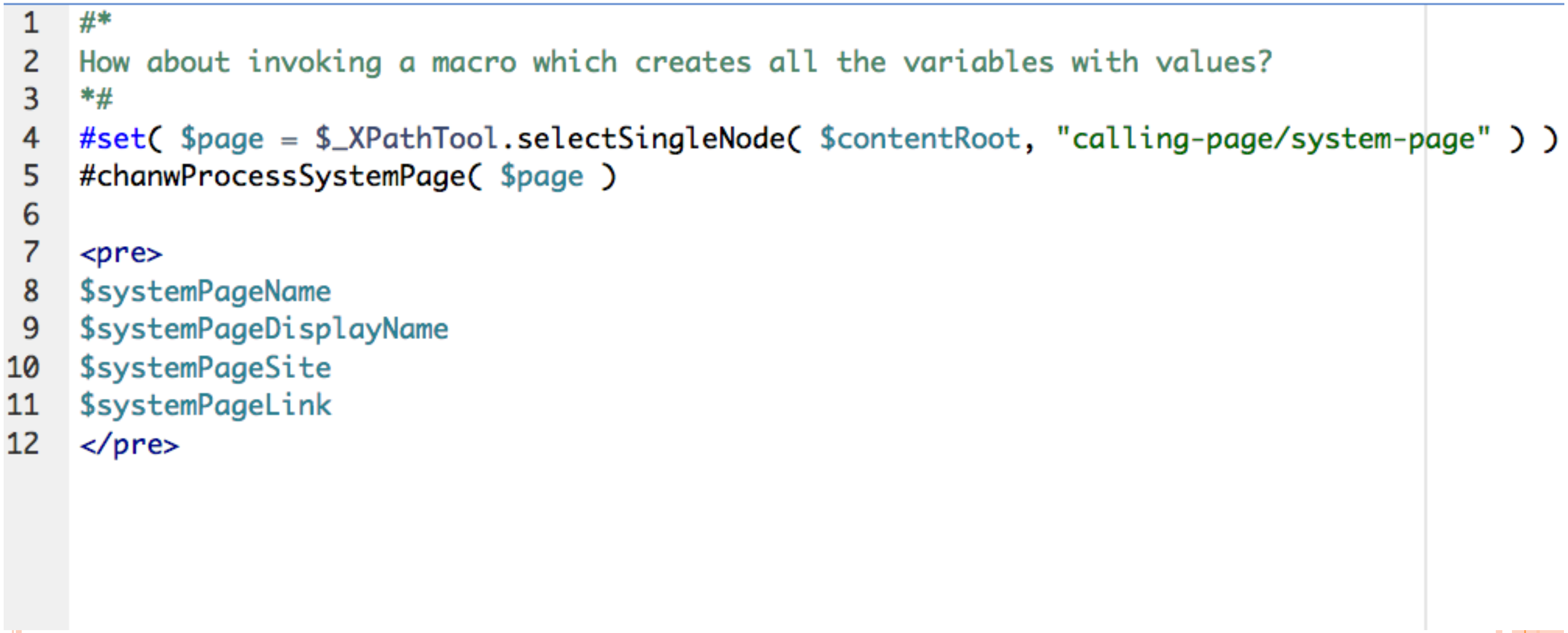
```


-  -
-  -
-  -
- $var4 -
```



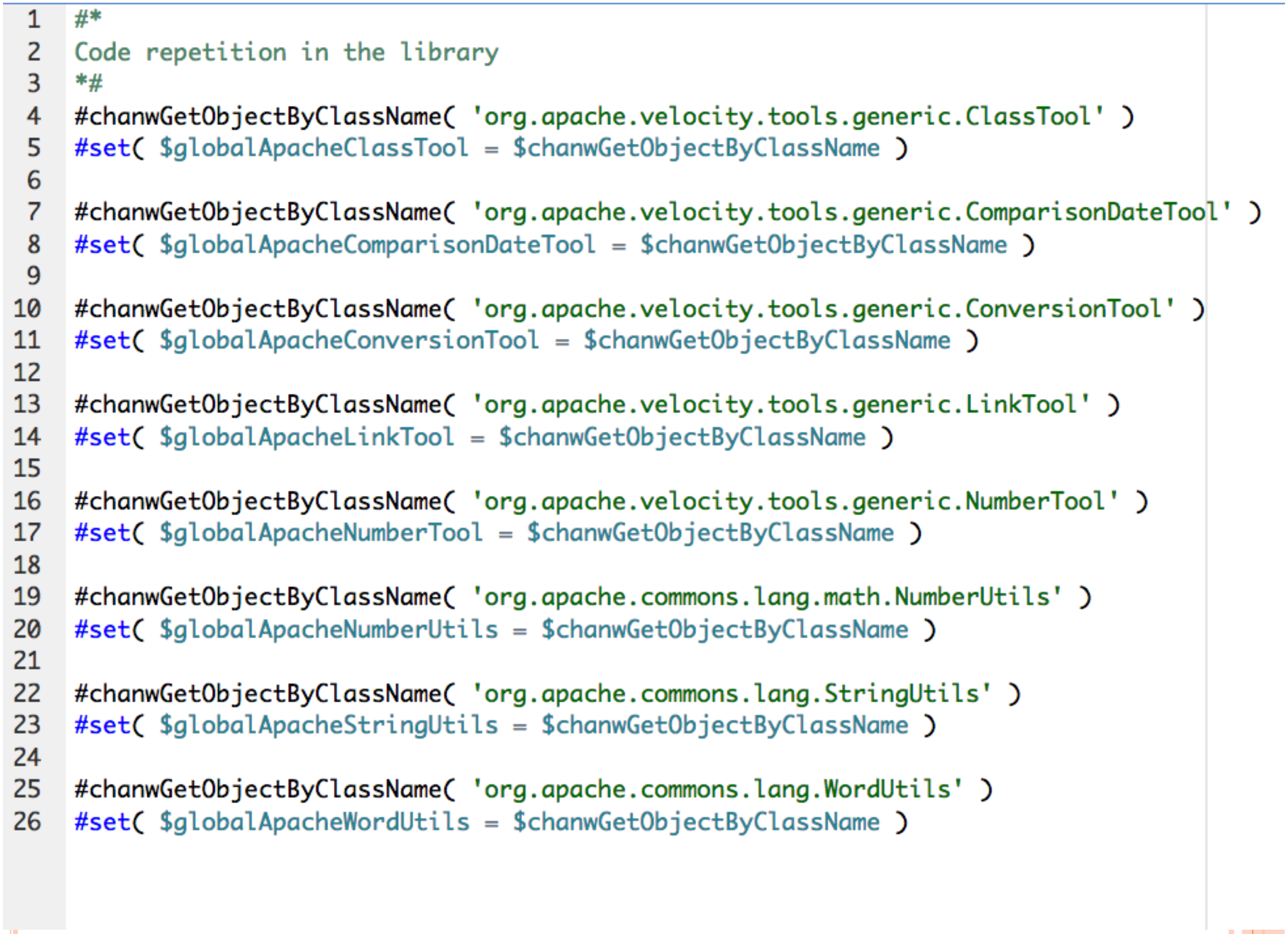
```
#*
How often do we write code like this? How do we avoid it?
*#
#set( $page = $_XPathTool.selectSingleNode( $contentRoot, "calling-page/system-page" ) )
#set( $name         = $page.getChild( "name" ) )
#set( $title        = $page.getChild( "title" ) )
#set( $display_name = $page.getChild( "display-name" ) )
#set( $site         = $page.getChild( "site" ) )
```



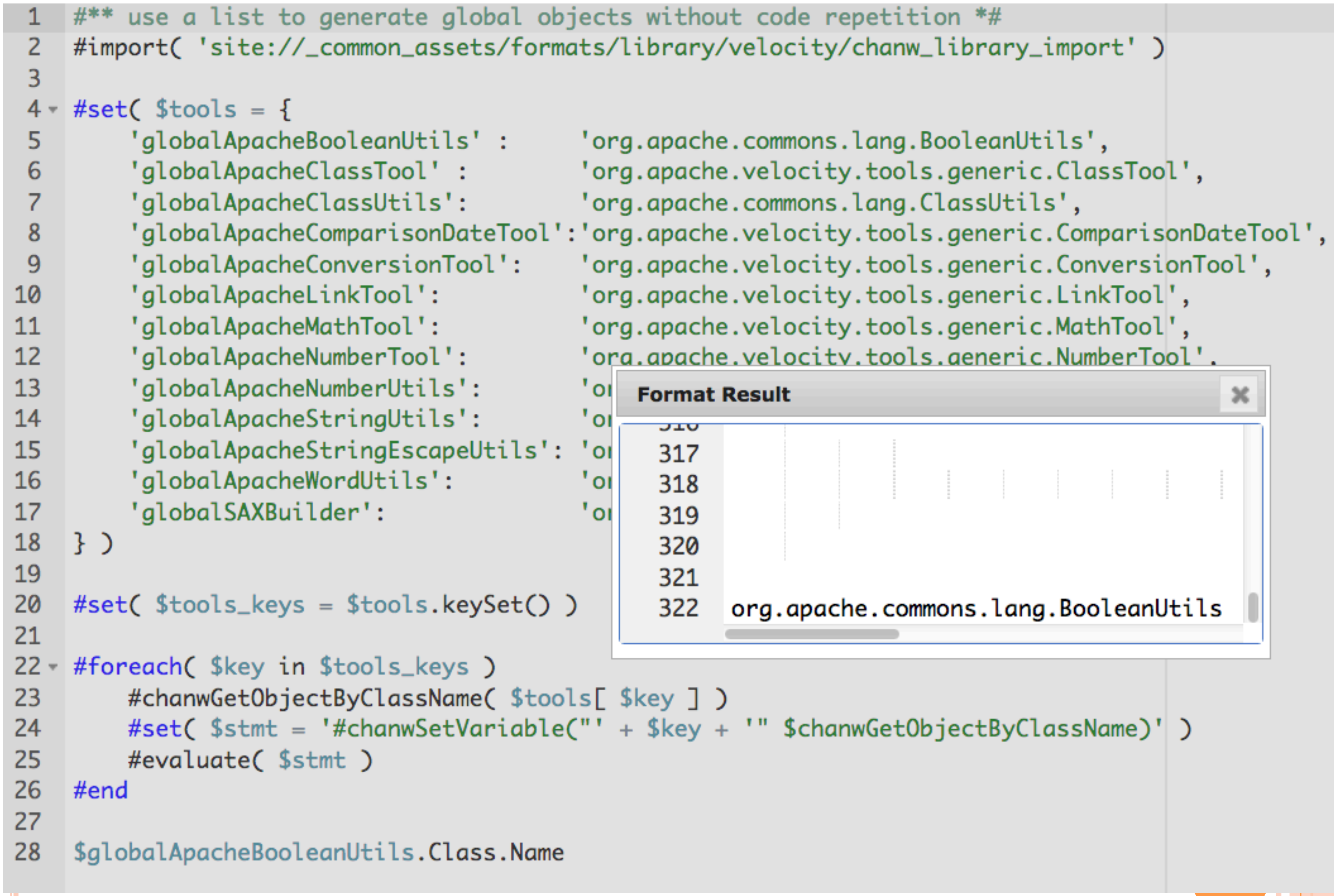
```
#*
How about invoking a macro which creates all the variables with values?
*#
#set( $block = $_XPathTool.selectSingleNode( $contentRoot, "//system-block" ) )
#chanwProcessSystemBlock( $block )

<pre>
$systemBlockName
$systemBlockSite
$systemBlockPath
</pre>
```



```
#*
How about invoking a macro which creates all the variables with values?
*#
#set( $page = $_XPathTool.selectSingleNode( $contentRoot, "calling-page/system-page" ) )
#chanwProcessSystemPage( $page )

<pre>
$systemPageName
$systemPageDisplayName
$systemPageSite
$systemPageLink
</pre>
```



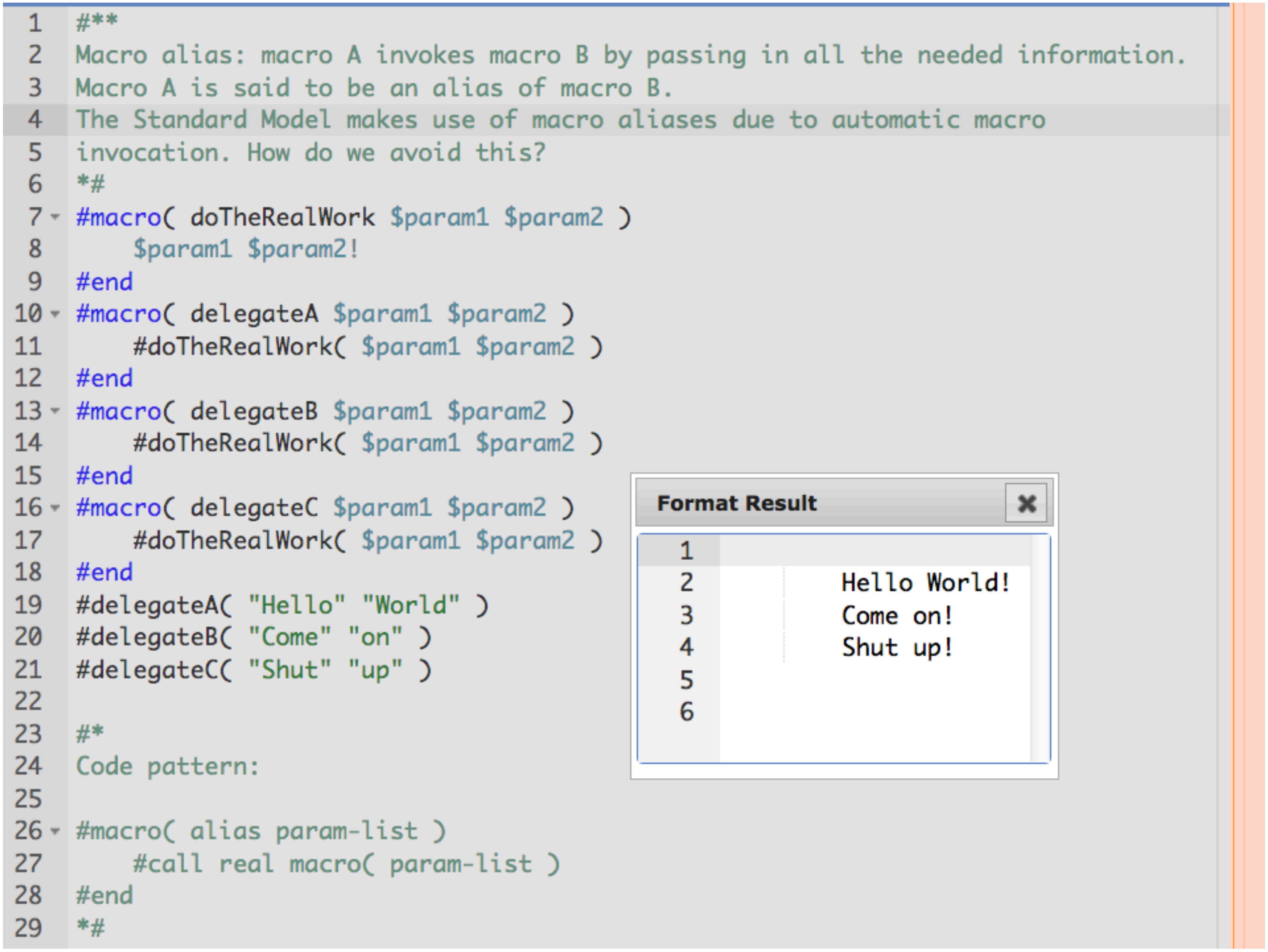
```
#*
Code repetition in the library
*#
#chanwGetObjectByClassName( 'org.apache.velocity.tools.generic.ClassTool' )
#set( $globalApacheClassTool = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.velocity.tools.generic.ComparisonDateTool' )
#set( $globalApacheComparisonDateTool = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.velocity.tools.generic.ConversionTool' )
#set( $globalApacheConversionTool = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.velocity.tools.generic.LinkTool' )
#set( $globalApacheLinkTool = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.velocity.tools.generic.NumberTool' )
#set( $globalApacheNumberTool = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.commons.lang.math.NumberUtils' )
#set( $globalApacheNumberUtils = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.commons.lang.StringUtils' )
#set( $globalApacheStringUtils = $chanwGetObjectByClassName )

#chanwGetObjectByClassName( 'org.apache.commons.lang.WordUtils' )
#set( $globalApacheWordUtils = $chanwGetObjectByClassName )
```



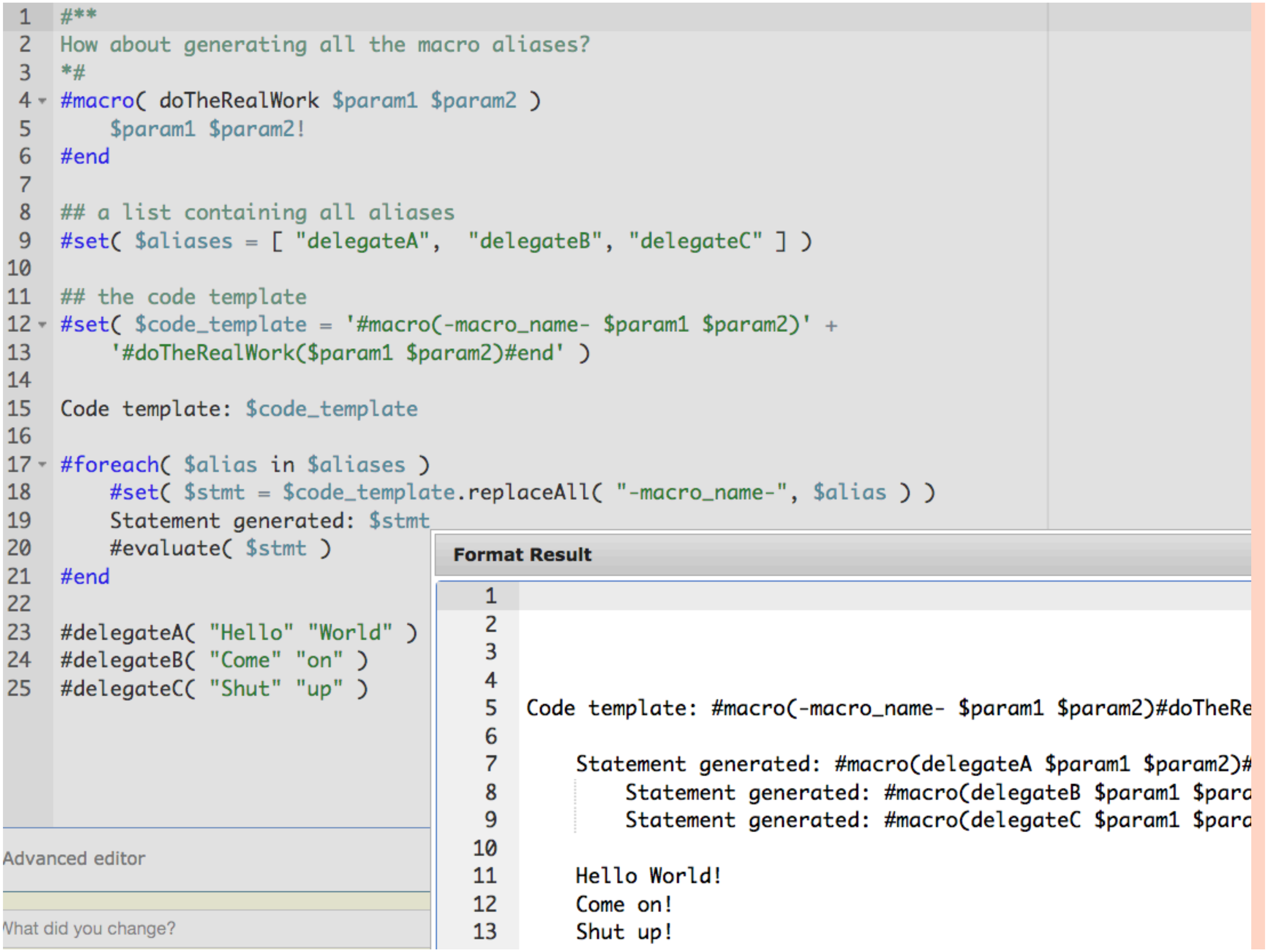
```
#** use a list to generate global objects without code repetition *#
#import( 'site://_common_assets/formats/library/velocity/chanw_library_import' )

#set( $tools = {
    'globalApacheBooleanUtils' :     'org.apache.commons.lang.BooleanUtils',
    'globalApacheClassTool' :        'org.apache.velocity.tools.generic.ClassTool',
    'globalApacheClassUtils':        'org.apache.commons.lang.ClassUtils',
    'globalApacheComparisonDateTool':'org.apache.velocity.tools.generic.ComparisonDateTool',
    'globalApacheConversionTool':    'org.apache.velocity.tools.generic.ConversionTool',
    'globalApacheLinkTool':          'org.apache.velocity.tools.generic.LinkTool',
    'globalApacheMathTool':          'org.apache.velocity.tools.generic.MathTool',
    'globalApacheNumberTool':        'org.apache.velocity.tools.generic.NumberTool',
    'globalApacheNumberUtils':       'or
    'globalApacheStringUtils':       'or
    'globalApacheStringEscapeUtils': 'or
    'globalApacheWordUtils':         'or
    'globalSAXBuilder':              'or
} )

#set( $tools_keys = $tools.keySet() )

#foreach( $key in $tools_keys )
    #chanwGetObjectByClassName( $tools[ $key ] )
    #set( $stmt = '#chanwSetVariable("' + $key + '" $chanwGetObjectByClassName)' )
    #evaluate( $stmt )
#end

$globalApacheBooleanUtils.Class.Name
```

Format Result

```
317
318
319
320
321
322  org.apache.commons.lang.BooleanUtils
```


```
#**
Macro alias: macro A invokes macro B by passing in all the needed information.
Macro A is said to be an alias of macro B.
The Standard Model makes use of macro aliases due to automatic macro
invocation. How do we avoid this?
*#
#macro( doTheRealWork $param1 $param2 )
    $param1 $param2!
#end
#macro( delegateA $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end
#macro( delegateB $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end
#macro( delegateC $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end
#delegateA( "Hello" "World" )
#delegateB( "Come" "on" )
#delegateC( "Shut" "up" )

#*
Code pattern:

#macro( alias param-list )
    #call real macro( param-list )
#end
*#
```

```

        Hello World!
        Come on!
        Shut up!


```



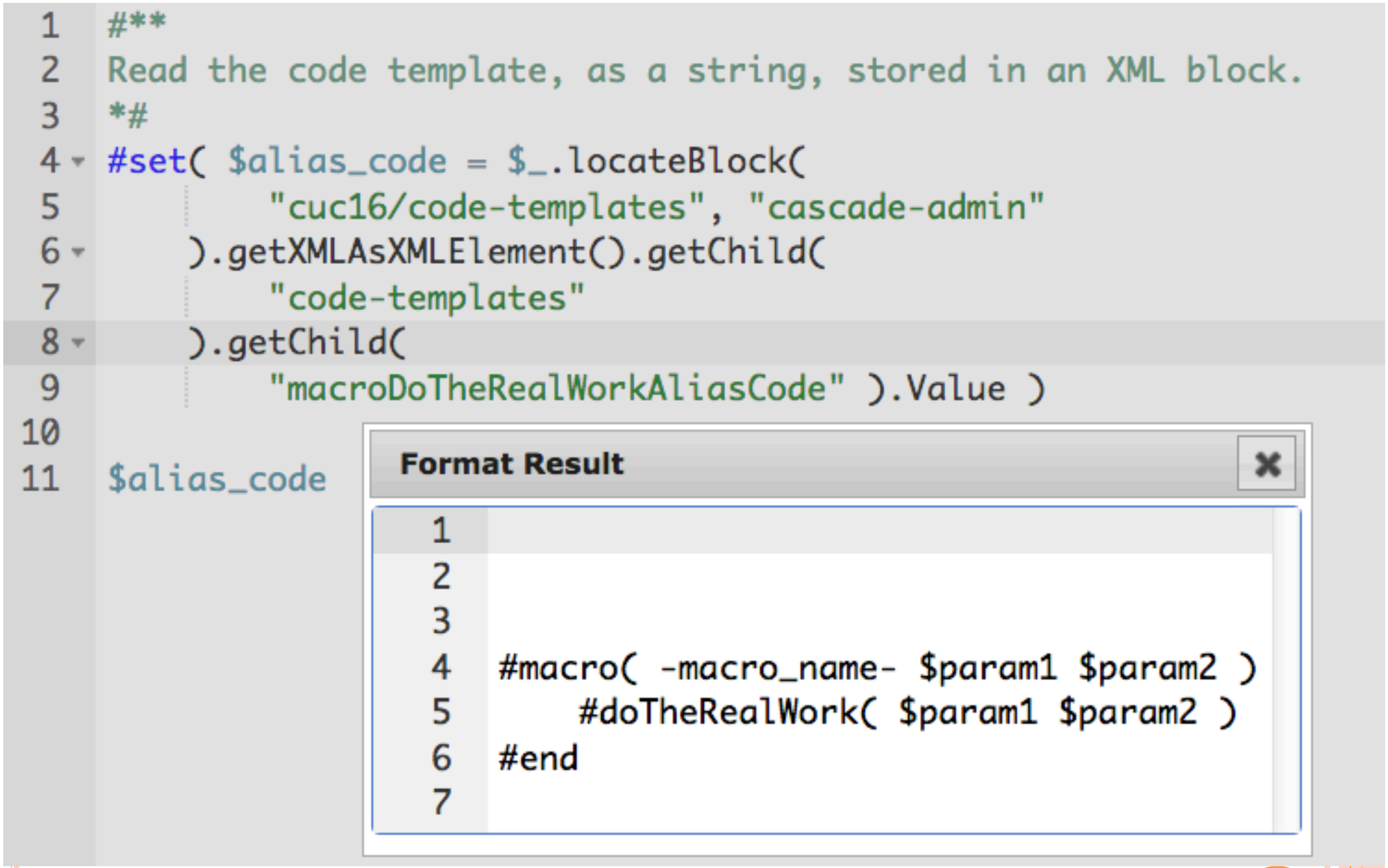
```
#**
How about generating all the macro aliases?
*#
#macro( doTheRealWork $param1 $param2 )
    $param1 $param2!
#end

## a list containing all aliases
#set( $aliases = [ "delegateA",  "delegateB", "delegateC" ] )

## the code template
#set( $code_template = '#macro(-macro_name- $param1 $param2)' +
    '#doTheRealWork($param1 $param2)#end' )

Code template: $code_template

#foreach( $alias in $aliases )
    #set( $stmt = $code_template.replaceAll( "-macro_name-", $alias ) )
    Statement generated: $stmt
    #evaluate( $stmt )
#end

#delegateA( "Hello" "World" )
#delegateB( "Come" "on" )
#delegateC( "Shut" "up" )
```

Advanced editor

What did you change?

Format Result

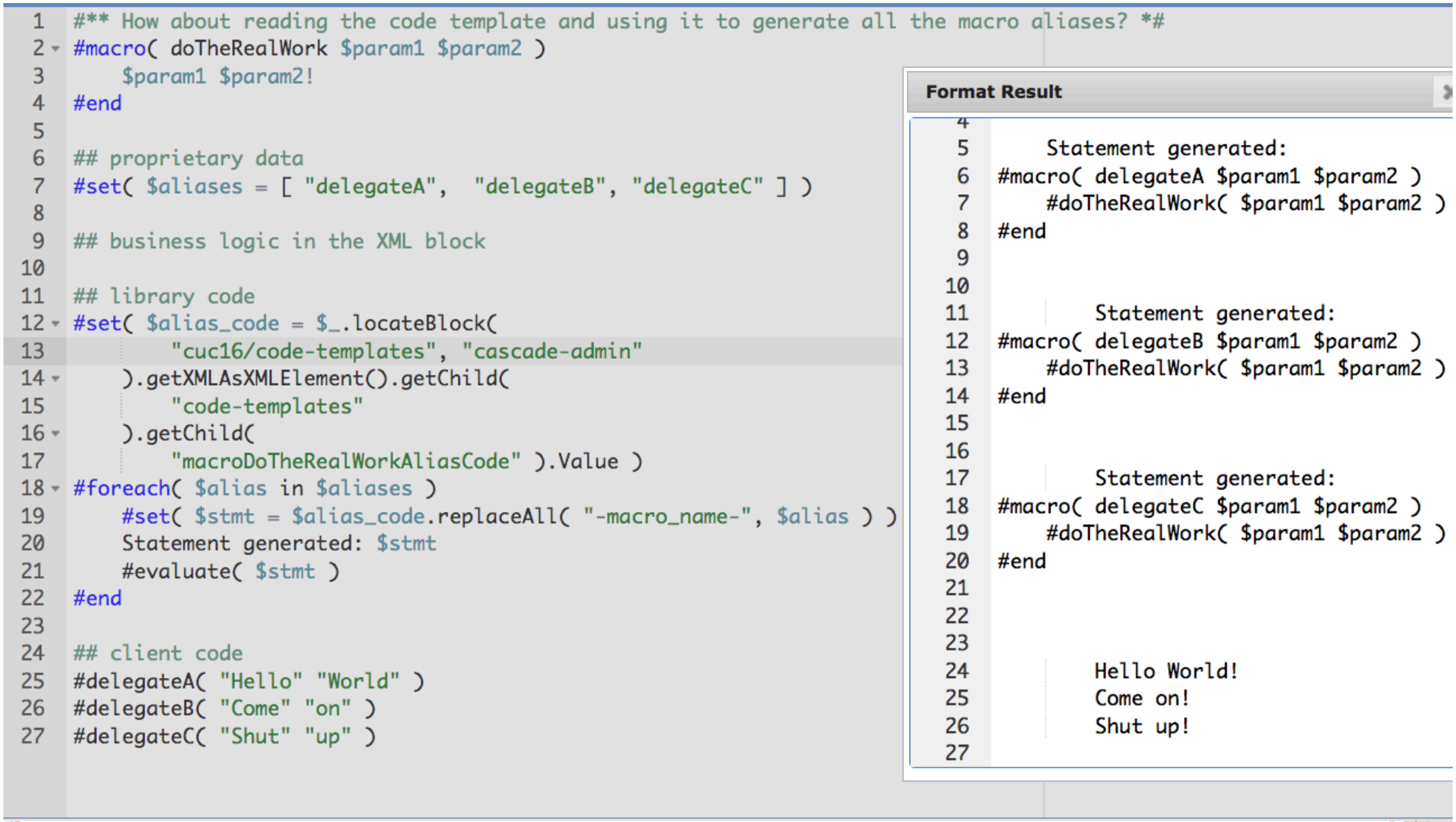
```




Code template: #macro(-macro_name- $param1 $param2)#doTheRe

    Statement generated: #macro(delegateA $param1 $param2)#
        Statement generated: #macro(delegateB $param1 $para
        Statement generated: #macro(delegateC $param1 $para

    Hello World!
    Come on!
    Shut up!
```



```
#**
Read the code template, as a string, stored in an XML block.
*#
#set( $alias_code = $_.locateBlock(
        "cuc16/code-templates", "cascade-admin"
    ).getXMLAsXMLElement().getChild(
        "code-templates"
    ).getChild(
        "macroDoTheRealWorkAliasCode" ).Value )

$alias_code
```

Format Result

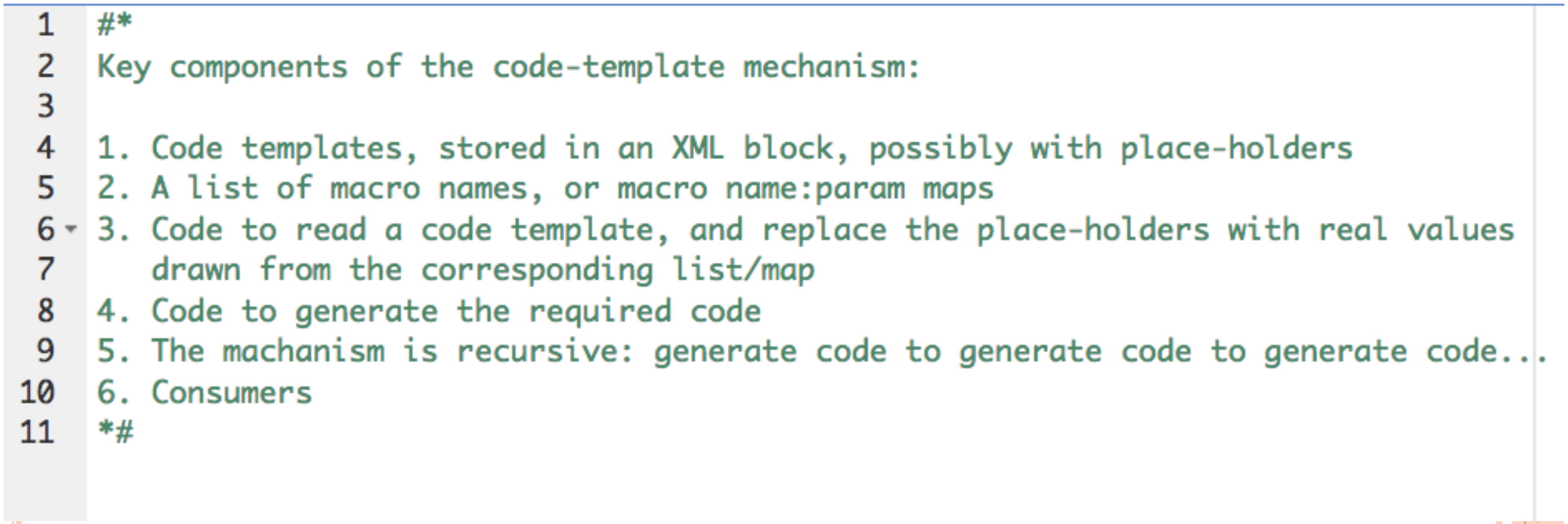
```



#macro( -macro_name- $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end

```



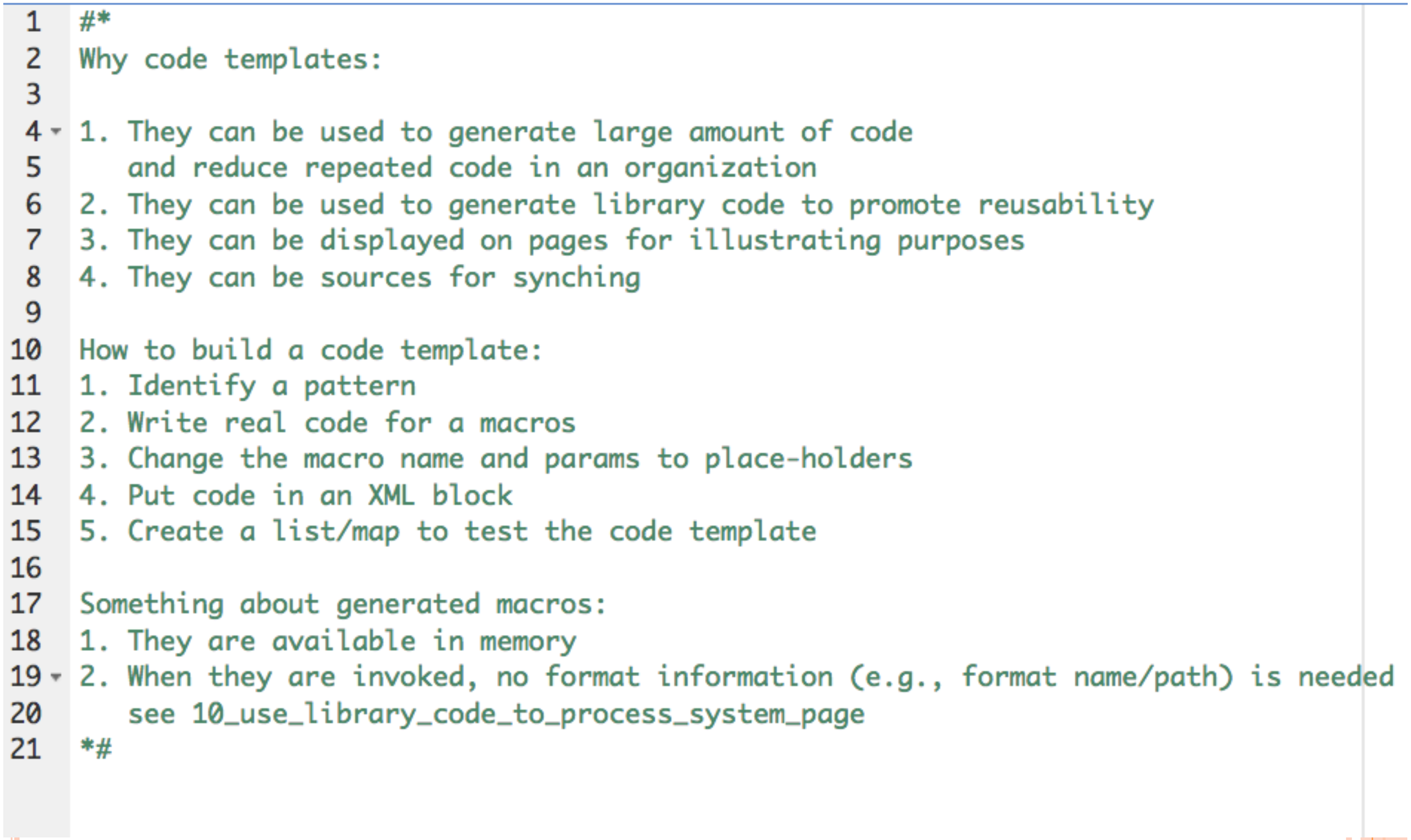
```
#** How about reading the code template and using it to generate all the macro aliases? *#
#macro( doTheRealWork $param1 $param2 )
    $param1 $param2!
#end

## proprietary data
#set( $aliases = [ "delegateA",  "delegateB", "delegateC" ] )

## business logic in the XML block

## library code
#set( $alias_code = $_.locateBlock(
        "cuc16/code-templates", "cascade-admin"
    ).getXMLAsXMLElement().getChild(
        "code-templates"
    ).getChild(
        "macroDoTheRealWorkAliasCode" ).Value )
#foreach( $alias in $aliases )
    #set( $stmt = $alias_code.replaceAll( "-macro_name-", $alias ) )
    Statement generated: $stmt
    #evaluate( $stmt )
#end

## client code
#delegateA( "Hello" "World" )
#delegateB( "Come" "on" )
#delegateC( "Shut" "up" )
```

Format Result

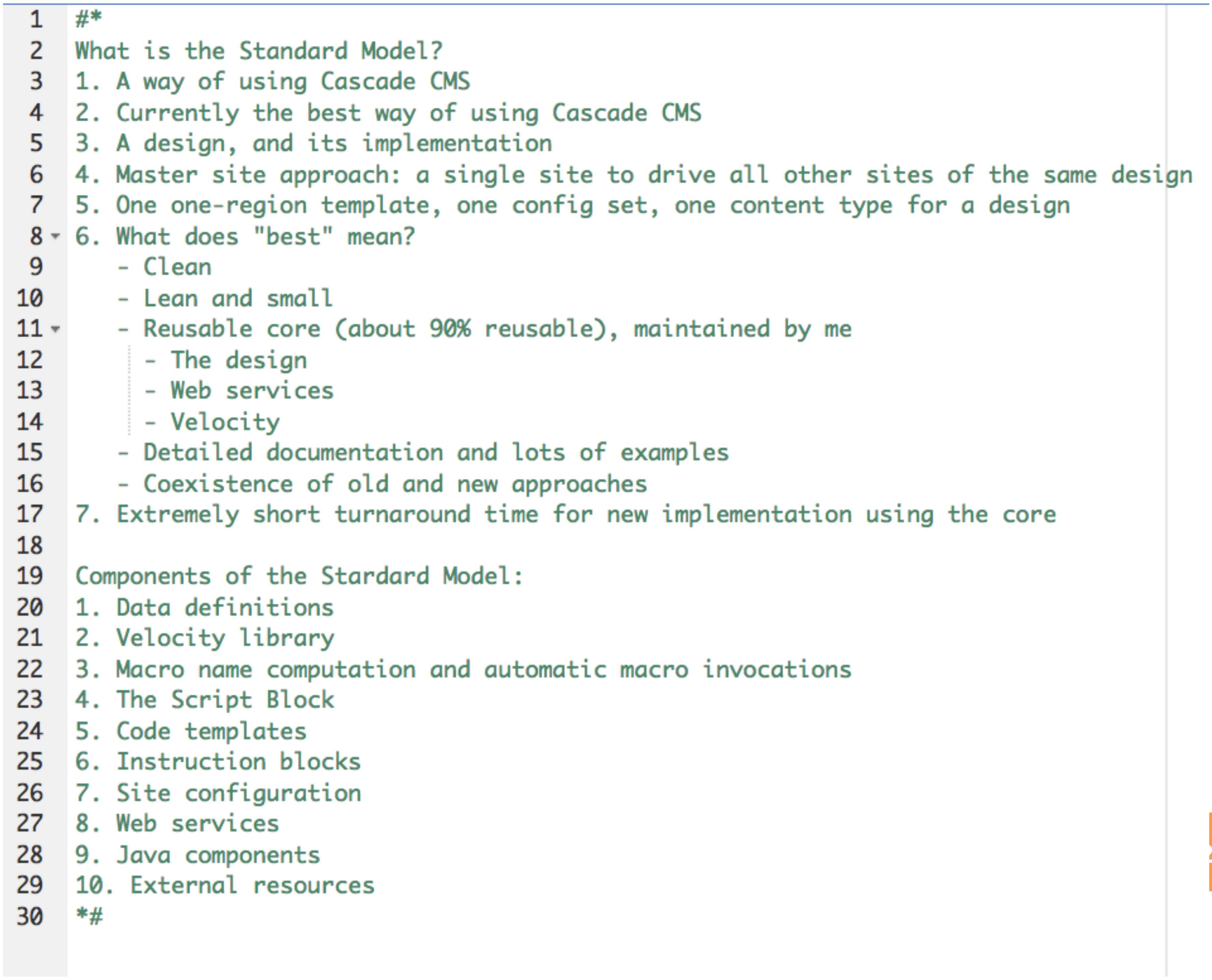
```

    Statement generated:
#macro( delegateA $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end


        Statement generated:
#macro( delegateB $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end


        Statement generated:
#macro( delegateC $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end



        Hello World!
        Come on!
        Shut up!

```



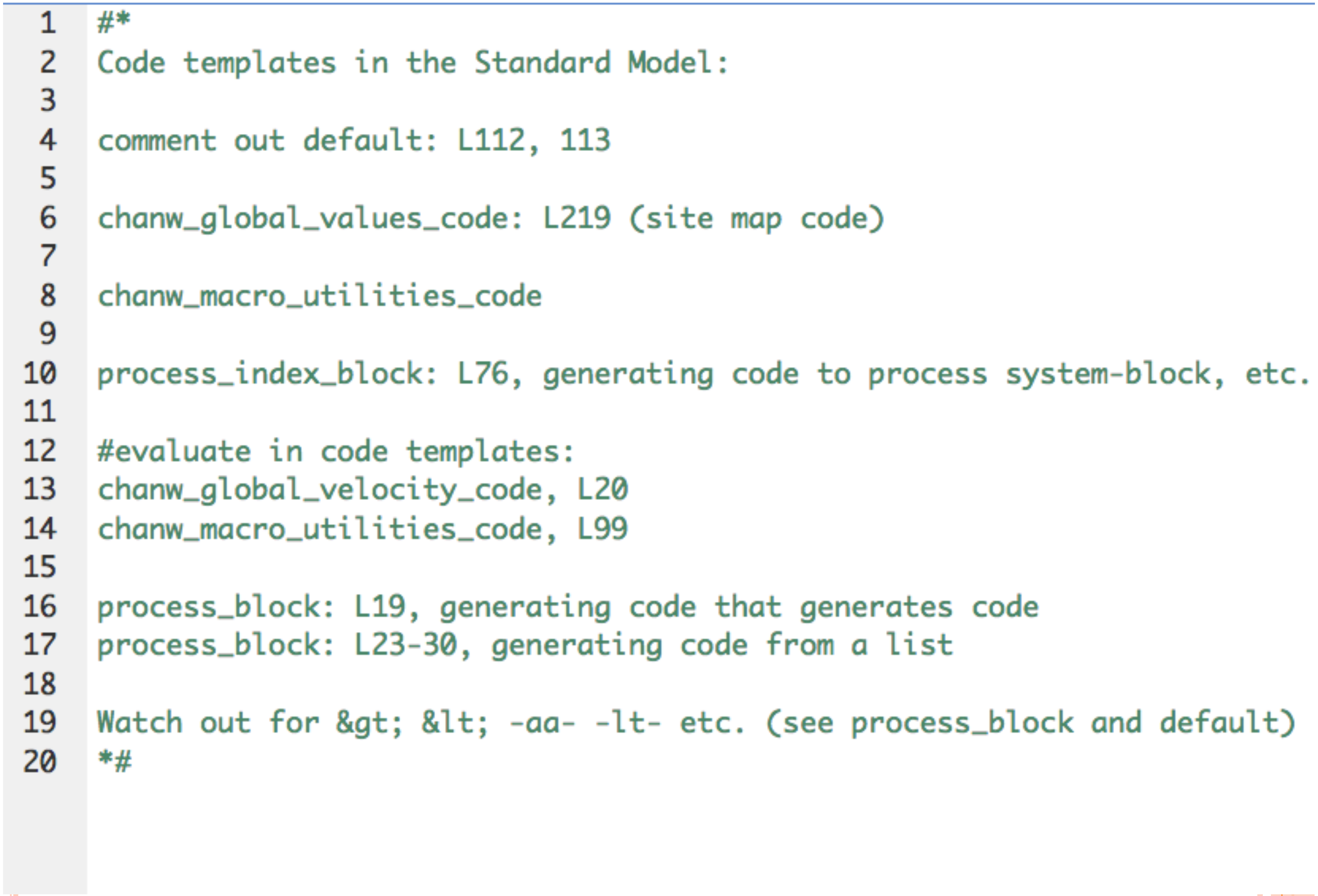
```
#*
Key components of the code-template mechanism:

1. Code templates, stored in an XML block, possibly with place-holders
2. A list of macro names, or macro name:param maps
3. Code to read a code template, and replace the place-holders with real values
   drawn from the corresponding list/map
4. Code to generate the required code
5. The machanism is recursive: generate code to generate code to generate code...
6. Consumers
*#
```



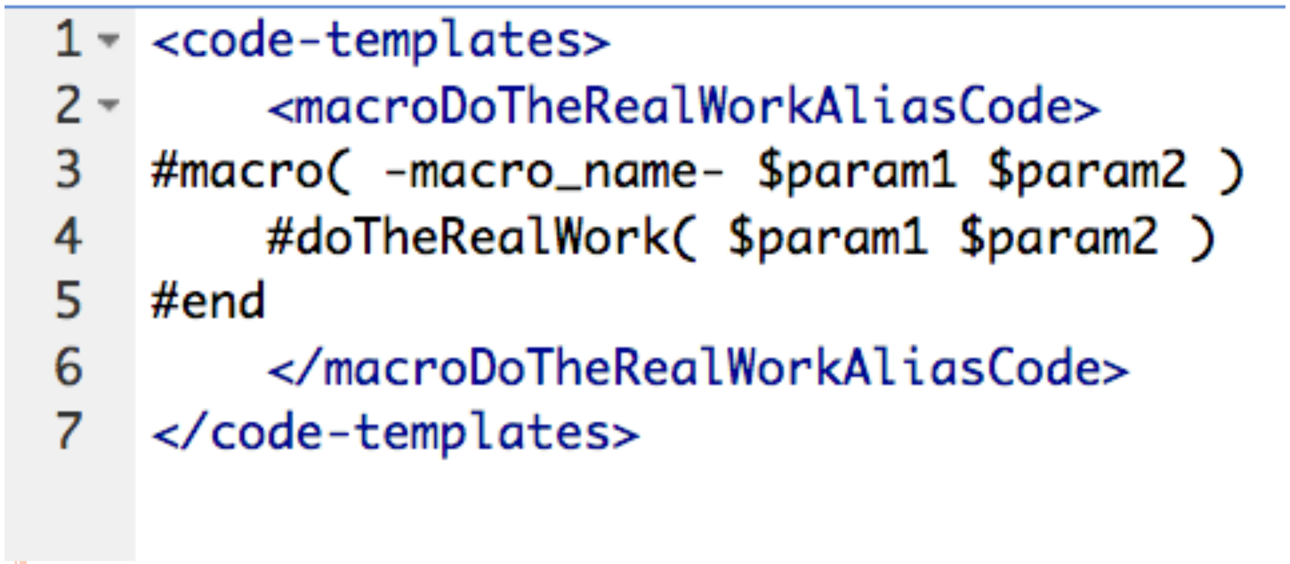
```
#*
Why code templates:

1. They can be used to generate large amount of code
   and reduce repeated code in an organization
2. They can be used to generate library code to promote reusability
3. They can be displayed on pages for illustrating purposes
4. They can be sources for synching

How to build a code template:
1. Identify a pattern
2. Write real code for a macros
3. Change the macro name and params to place-holders
4. Put code in an XML block
5. Create a list/map to test the code template

Something about generated macros:
1. They are available in memory
2. When they are invoked, no format information (e.g., format name/path) is needed
   see 10_use_library_code_to_process_system_page
*#
```



```
#*
What is the Standard Model?
1. A way of using Cascade CMS
2. Currently the best way of using Cascade CMS
3. A design, and its implementation
4. Master site approach: a single site to drive all other sites of the same design
5. One one-region template, one config set, one content type for a design
6. What does "best" mean?
   - Clean
   - Lean and small
   - Reusable core (about 90% reusable), maintained by me
     - The design
     - Web services
     - Velocity
   - Detailed documentation and lots of examples
   - Coexistence of old and new approaches
7. Extremely short turnaround time for new implementation using the core

Components of the Stardard Model:
1. Data definitions
2. Velocity library
3. Macro name computation and automatic macro invocations
4. The Script Block
5. Code templates
6. Instruction blocks
7. Site configuration
8. Web services
9. Java components
10. External resources
*#
```



```
#*
Code templates in the Standard Model:

comment out default: L112, 113

chanw_global_values_code: L219 (site map code)

chanw_macro_utilities_code

process_index_block: L76, generating code to process system-block, etc.

#evaluate in code templates:
chanw_global_velocity_code, L20
chanw_macro_utilities_code, L99

process_block: L19, generating code that generates code
process_block: L23-30, generating code from a list

Watch out for &gt; &lt; -aa- -lt- etc. (see process_block and default)
*#
```



```
<code-templates>
    <macroDoTheRealWorkAliasCode>
#macro( -macro_name- $param1 $param2 )
    #doTheRealWork( $param1 $param2 )
#end
    </macroDoTheRealWorkAliasCode>
</code-templates>
```

QUESTIONS?

- Reference sites: http://www.upstate.edu/cascade-admin/standard-model/index.php, https://github.com/wingmingchan/velocity

- Email: chanw@upstate.edu