timeIt

The timeBlock macro times how long it takes to execute a block of code.

See the Boo benchmarks page for examples using this macro.

The original JIRA issue has the C# source for this macro. It has been converted to the boo code below (also, the name was changed):

```
#region license
// Copyright (c) 2004-2005, William P. Wood
// All rights reserved.
//
// Redistribution and use in source and
binary forms, with or without modification,
// are permitted provided that the following
conditions are met:
//
//     * Redistributions of source code must
retain the above copyright notice,
//     this list of conditions and the
following disclaimer.
//     * Redistributions in binary form must
reproduce the above copyright notice,
//     this list of conditions and the
following disclaimer in the documentation
//     and/or other materials provided with
the distribution.
//     * Neither the name of William P. Wood
nor the names of its
//     contributors may be used to endorse
or promote products derived from this
//     software without specific prior
written permission.
```

```
//
// THIS SOFTWARE IS PROVIDED BY THE
COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND
// ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY,
// OR TORT (INCLUDING NEGLIGENCE OR
OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF
// THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
#endregion

namespace Boo.Lang.Useful.Macros

import System
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
```

```
class TimeItMacro(AbstractAstMacro):
"""
Times a block of code.

@author William P. Wood
@author Sorin Ionescu
"""
 override def Expand(macro as
MacroStatement) as Statement:
  argumentCount = macro.Arguments.Count
  argumentIndex = 0
  localIndex = _context.AllocIndex()
  statement = Block(macro.LexicalInfo)
  macroBlock = macro.Block
  message = StringLiteralExpression(
   macro.LexicalInfo,
   'Elapsed time: ')

  elapsed = ReferenceExpression(
   macro.LexicalInfo,
   "___elapsedTime${localIndex}")

  start = ReferenceExpression(
   macro.LexicalInfo,
   "___startTime${localIndex}")

  dateNow =
AstUtil.CreateReferenceExpression('date.Now'
)

  # loop
```

```
  if argumentCount > argumentIndex and \
     macro.Arguments[argumentIndex] isa
IntegerLiteralExpression:
   loopsCount = cast(
    IntegerLiteralExpression,
    macro.Arguments[argumentIndex]).Value

   message = StringLiteralExpression(
    macro.LexicalInfo,
    "Elapsed time for ${loopsCount}
executions: ")

   ++argumentIndex

   # unroll
   if argumentCount > argumentIndex and \
      macro.Arguments[argumentIndex] isa
IntegerLiteralExpression:
    argument = \
     macro.Arguments[argumentIndex] as
IntegerLiteralExpression

    ++argumentIndex

    macroBlock = CreateTimerBlock(
     macro,
     loopsCount,
     argument.Value,
     localIndex)

   else:
```

```
    macroBlock = CreateTimerBlock(macro,
loopsCount, 8, localIndex)

  # message
  if argumentCount > argumentIndex and \
     macro.Arguments[argumentIndex] isa
StringLiteralExpression:
   message = macro.Arguments[argumentIndex]

   ++argumentIndex

  # elapsed
  if argumentCount > argumentIndex and \
     macro.Arguments[argumentIndex] isa
ReferenceExpression:
   elapsed = macro.Arguments[argumentIndex]
   message = null

   ++argumentIndex

  if argumentCount != argumentIndex:
   raise ArgumentException(
    "format is 'timeIt [loops [, unroll]] [,
message] [, elapsed]'")

  statement.Add(
   BinaryExpression(
    macro.LexicalInfo,
    BinaryOperatorType.Assign,
    start,
    dateNow))
```

```
  statement.Add(macroBlock)

  elapsedCalculation = BinaryExpression(
   macro.LexicalInfo,
   BinaryOperatorType.Subtraction,
   dateNow.CloneNode(),
   start.CloneNode())

  statement.Add(
   BinaryExpression(
    macro.LexicalInfo,
    BinaryOperatorType.Assign,
    elapsed,
    elapsedCalculation))

  if message != null:
   statement.Add(

AstUtil.CreateMethodInvocationExpression(
     macro.LexicalInfo,

AstUtil.CreateReferenceExpression('System.Co
nsole.Write'),
     message))

   statement.Add(

AstUtil.CreateMethodInvocationExpression(
     macro.LexicalInfo,
     AstUtil.CreateReferenceExpression(
```

```
      'System.Console.WriteLine'),
     elapsed))

  return statement

 private def CreateTimerBlock(
  macro as MacroStatement,
  loopsCount as int,
  unroll as int,
  localIndex):

  statement = Block(macro.LexicalInfo)
  whileStatement =
WhileStatement(macro.LexicalInfo)
  loopCounter =
ReferenceExpression("___loopCounter${localIn
dex}")
  unroll = 1 if unroll < 1

  if loopsCount >= 2 * unroll:
   statement.Add(
    BinaryExpression(
     macro.LexicalInfo,
     BinaryOperatorType.Assign,
     loopCounter,

IntegerLiteralExpression(macro.LexicalInfo,
0)))

   loopsCount /= unroll
```

```
   whileStatement.Condition =
BinaryExpression(
    macro.LexicalInfo,
    BinaryOperatorType.LessThan,
    loopCounter.CloneNode(),
    IntegerLiteralExpression(
     macro.LexicalInfo,
     loopsCount))

   whileStatement.Block.Add(
    UnaryExpression(
     LexicalInfo: macro.LexicalInfo,
     Operator: UnaryOperatorType.Increment,
     Operand: loopCounter.CloneNode()))

   for index in range(unroll):

whileStatement.Block.Add(macro.Block.CloneNo
de())

   loopsCount -= loopsCount * unroll
   statement.Add(whileStatement)

  for index in range(loopsCount, 0):
   statement.Add(macro.Block.CloneNode())
```

```
return statement
```

See also the --profile option for the mono command line tool to profile your application for bottlenecks.