Testing: Tasty

We will use the Tasty implementation of hunit, which is based on the Junit testing framework in Java.

Here are some useful links:

https://caiorss.github.io/Functional-Programming/haskell/UnitTest_Hunit.htm

http://hackage.haskell.org/package/HUnit

http://hackage.haskell.org/package/tasty

hunit enables you to create a hierarchical tree structure of tests, based on

- **Assertions** -- True or false assertions about the behavior of your code
- **Test Cases** -- Sequences of related assertions, which fail or succeed as a whole.
- **Test Groups** – Lists of Test Cases or other Test Groups
Testing: Assertions

You test your code by making **assertions** about the values returned by your code. There are two useful ways to do this, the first is

\[
\text{assertBool} :: \text{String} \rightarrow \text{Bool} \rightarrow \text{Assertion}
\]

This function takes a Boolean expression (something about your code you want to be true) and an error message. Your error message will be printed if the expression is false.

Examples:

\[
\text{assertBool} \text{ "3 is not less than 2!" } (3 < 2)
\]

\[
\text{assertBool} \text{ "4 in [2,3,4]?" } (\text{elem 4 [2,3,4]})
\]
Testing: Assertions

A second, and even more useful is

\[
\text{assertEqual} :: (\text{Eq } a, \text{ Show } a) \Rightarrow \text{String} \to a \to a \to \text{Assertion}
\]

This is similar to the previous, except that you give it two expressions, typically the correct value you expect, and a call to some function to produce that value; again, if they are not equal, then the error message is printed out.

\[
\text{assertEqual } "\text{factorial 5 = ?}" 120 (\text{factorial 5})
\]

An abbreviation for this assertion (without a warning message) is provided using the infix operator (@=?) so the previous assertion could be written as

\[
120 @=? (\text{factorial 5})
\]

however this does not allow you to give an error message.
**Testing: Test Cases**

A **test case** is a single assertion or a sequence of assertions in a do expression.

A test case succeeds ("OK") if all the assertions are true, and fails ("FAIL") otherwise; thus a sequence of assertions in a do expression act like they are connected with "and" (&&).

Test cases have labels which are printed out when the result is reported.

**Example:**

```hs
testCase "Singular Test Case" $ assertBool "What??" True
```

```hs
testCase "Sequence of Tests"
  do assertBool "should be true" True
  assertEqual "(2+1)/= 5 !" (2+1) 5
  assertEqual "4 /= 2 !" 4 2
```

The second testCase will succeed only if all three of the assertions succeed.
Testing: Test Groups

A test group is simply a label and a list of test cases.

tests = testGroup "ExampleTest"
  [ testCase "Fact test" $ assertEqual "fact 5 = ?" 120 (fact 5),
  testCase "Mem test" $ do assertBool "mem 3 []" (not (mem 3 []))
    assertBool "mem 3 [3]" (mem 3 [3])
    assertBool "mem 3 [_,3]" (mem 3 [2,3]),
  testCase "Mod test" $ assertEqual "5 % 3 = ?" 2 (5 % 3),
  testCase "Another test" $ 5 @=? 4
  ]

Each of the test cases will be tested individually and reported. Make sure to put a comma after each test case, since this is a list!

You may have to use parentheses to make sure they get parsed correctly.

You can nest test groups, essentially creating a tree of test cases, which will be displayed indented when the tests are run.
## Testing: Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Size</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td>Apr 23, 2019 at 11:04 PM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>dist-newstyle</td>
<td>Apr 23, 2019 at 10:59 PM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>Misc</td>
<td>Apr 20, 2019 at 5:18 PM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>src</td>
<td>Today at 8:41 PM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>Ast.hs</td>
<td>Today at 3:31 PM</td>
<td>10 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Check.hs</td>
<td>Today at 3:37 PM</td>
<td>6 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>EnvUnsafeLog.hs</td>
<td>Apr 22, 2019 at 2:00 PM</td>
<td>3 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Eval.hs</td>
<td>Apr 23, 2019 at 11:33 PM</td>
<td>20 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Exec.hs</td>
<td>Apr 15, 2019 at 8:37 PM</td>
<td>322 bytes</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>HelpShow.hs</td>
<td>Apr 23, 2019 at 11:11 PM</td>
<td>1 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Main.hs</td>
<td>Apr 23, 2019 at 2:06 PM</td>
<td>40 bytes</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Parser.hs</td>
<td>Today at 8:00 PM</td>
<td>24 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>ParserMonad.hs</td>
<td>Apr 23, 2019 at 11:06 AM</td>
<td>7 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Repl.hs</td>
<td>Apr 21, 2019 at 10:51 PM</td>
<td>1 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>TypeCheckingExample.hs</td>
<td>Apr 22, 2019 at 8:27 AM</td>
<td>2 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Static Type Checking Code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tests</td>
<td>Today at 8:40 PM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>CheckTest.hs</td>
<td>Today at 2:41 PM</td>
<td>289 bytes</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>EvalTest.hs</td>
<td>Today at 3:05 PM</td>
<td>289 bytes</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>ExampleTest.hs</td>
<td>Today at 7:45 PM</td>
<td>3 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Main.hs</td>
<td>Today at 7:48 PM</td>
<td>2 KB</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>ParserTest.hs</td>
<td>Today at 7:56 PM</td>
<td>838 bytes</td>
<td>Haskell...rc</td>
</tr>
<tr>
<td>Project.cabal</td>
<td>Today at 7:38 PM</td>
<td>819 bytes</td>
<td>Emacs...rc</td>
</tr>
<tr>
<td>Notes.txt</td>
<td>Apr 15, 2019 at 4:24 PM</td>
<td>361 bytes</td>
<td>Plain Text</td>
</tr>
</tbody>
</table>
Testing: Example

name: Project
version: 0.1.0.0
synopsis: Wayne Snyder
homepage: waysnysner@gmail.com
author: waysnysner@gmail.com
maintainer: waysnysner@gmail.com
category: Simple
cabal-version: >=1.10

library
  exposed-modules: Ast, Eval, Check, Parser, EnvUnsafeLog
  other-modules: ParserMonad, HelpShow
  ghc-options: -fwarn-incomplete-patterns -fwarn-incomplete-uni-patterns
  build-depends: containers, base >= 4.7 && < 5
  hs-source-dirs: src
default-language: Haskell2010

test-suite test
default-language: Haskell2010
type: exitcode-stdio-1.0
hs-source-dirs: tests
other-modules: ExampleTest
main-is: Main.hs
other-modules: ExampleTest
build-depends:
  containers, base >= 4.7 && < 5
  , tasty >= 0.7, tasty-hunit
  , Project
Testing: Example

```haskell
module Main where

import System.Environment
import Test.Tasty (defaultMain, testGroup, TestTree)

-- import all the files where tests of various files/features are located
import ExampleTest

-- Look at the comments in ExampleTest.hs for details on how to write tests

-- this will set up how to run this test suite
main = do
  setEnv "TASTY_TIMEOUT" "40s"
  setEnv "TASTY_QUICKCHECK_TESTS" "10000" -- TODO: I never trust less than 10000
  setEnv "TASTY_QUICKCHECK_MAX_SIZE" "50"
  defaultMain testSuite
unsetEnv "TASTY_TIMEOUT"
unsetEnv "TASTY_QUICKCHECK_TESTS"
unsetEnv "TASTY_QUICKCHECK_MAX_SIZE"

-- testSuite is a mast list of all tests you want to perform in all the various testing files

testSuite = testGroup "allTests" -- the name of this test group
  [ -- list here the name of all tests from those files you want to run
    ExampleTest.tests
  ]
```
Testing: Example

```haskell
module ExampleTest where

import Test.Tasty (testGroup)
import Test.Tasty.HUnit (assertEqual, assertBool, testCase, (@=?))

tests = testGroup "ExampleTest"
  [
    testCase "Simple integer: " $ assertBool "Should not be printed"  True,
    testCase "Simple float" $ do
      assertBool "True /= True !"  True
      assertBool "Reaches this one"  False
      assertBool "Doesn't reach this one" False,

    testCase "Is 5 == 5?" $ assertEqual "Should not be printed"  (5) (5),
    testCase "Is 5 == 4?" $ (5) @=? (4)
  ]
```
Testing: Example

Project $ cabal new-test
Resolving dependencies...
Build profile: -w ghc-8.6.3 -O1
In order, the following will be built (use -v for more details):
- Project-0.1.0.0 (lib) (configuration changed)
- Project-0.1.0.0 (test:test) (configuration changed)
Configuring library for Project-0.1.0.0..
Preprocessing library for Project-0.1.0.0..
Building library for Project-0.1.0.0..
Configuring test suite 'test' for Project-0.1.0.0..
Preprocessing test suite 'test' for Project-0.1.0.0..
Building test suite 'test' for Project-0.1.0.0..
Running 1 test suites...
Test suite test: RUNNING...

ExampleTest

    Simple integer: : OK
    Simple float:   FAIL
    Reaches this one
    Is 5 == 5?:  OK
    Is 5 == 4?:  FAIL
    expected: 5
    but got: 4

2 out of 4 tests failed (0.00s)
Test suite test: FAIL
Test suite logged to: /Users/snyder/Dropbox (BOSTON UNIVERSITY)/Documents/Teaching/CS320/Web/Homeworks and Labs/Project/dist-newstyle/build/x86_64-osx/ghc-8.6.3/Project-0.1.0.0/t/test/test/Project-0.1.0.0-test.log
0 of 1 test suites (0 of 1 test cases) passed.
cabal: Tests failed for test:test from Project-0.1.0.0.
**Testing: Quickcheck**

Quickcheck, which is used by Tasty, is a way of automatically generating tests cases. We will use it to automatically generate Ast expressions to see if our parser and showPretty functions are indeed consistent:

For any ast a:   a  ==  parse parser $ showPretty a 0

Here is a useful link:

[https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html](https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html)

**Quickcheck** enables you to create random expressions in your ast by generating all possible expressions under a certain size limit.
Testing: Quickcheck

```
name: TestingExample2
version: 0.1.0.0
synopsis: 
homepage:
author: Wayne Snyder
maintainer: waysnryder@gmail.com
build-type: Simple
build-depends: 
ghc-options: -fwarn-incomplete-patterns -fwarn-incomplete-uni-patterns
hs-source-dirs: src
default-language: Haskell2010

library
  exposed-modules: Ast, Lang, Parser, ParserMonad, Reader
  test-suite test
default-language: Haskell2010
type: exitcode-stdio-1.0
hs-source-dirs: tests
main-is: Main.hs
other-modules: ParserTest, EvalTest
build-depends: 
  containers, base >= 4.7 && < 5
  , tasty >= 0.7, tasty-hunit, tasty-quickcheck
  , TestingExample2
```
Testing: Quickcheck

Main.hs

module Main where

import System.Environment
import Test.Tasty (defaultMain, testGroup, TestTree)

-- import all the files where tests of various files/features are located
import ParserTest
import EvalTest
import Ast
import Lang
import Parser

-- Look at the comments in ExampleTest.hs for details on how to write tests

-- this will set up how to run this test suite
main =
    do
        setEnv "TASTY_TIMEOUT" "40s"
        setEnv "TASTY_QUICKCHECK_TESTS" "1000" --TODO: I never trust less than 10000
        setEnv "TASTY_QUICKCHECK_MAX_SIZE" "50"
        defaultMain testSuite
        unsetEnv "TASTY_TIMEOUT"
        unsetEnv "TASTY_QUICKCHECK_TESTS"
        unsetEnv "TASTY_QUICKCHECK_MAX_SIZE"

-- testSuite is a list of all tests you want to perform in all the various testing files

testSuite = testGroup "allTests" -- the name of this test group
    [ parserTest, -- list here the name of all tests from those files you want to run
      evalTest
    ]
module ParserTest where

import Test.Tasty (testGroup)
import Test.Tasty.HUnit (assertEqual, assertBool, testCase)
import Test.Tasty.QuickCheck

-- Import all the modules you need to do the test

import ParserMonad (parse)
import Ast
import Parser (parser)

-- This will generate random instances of types

instance Arbitrary Ast where
  arbitrary = sized arbitrarySizedAst

-- recursively and randomly generate instances up to a given size limit

arbitrarySizedAst :: Int -> Gen Ast
arbitrarySizedAst m | m < 1 = do i <- arbitrary -- will choose a random Integer
                     x <- elements ["x", "y", "z"] -- will choose random element from the list
                     node <- elements [LiteralInt i, Var x] -- so put all the non-recursive Ast expressions here
                     return $ node

arbitrarySizedAst m | otherwise = do l <- arbitrarySizedAst (m `div` 2) -- get ast half as big
                             r <- arbitrarySizedAst (m `div` 2) -- ditto
                             x <- elements ["x", "y", "z"] -- will choose random element from the list
                             ifAst <- arbitrarySizedIf m
                             node <- elements [Plus l r, Sub l r, Mult l r, ifAst, Let x l r]
                             -- list here all your binary Ast constructors
                             -- will choose from if expressions
                             -- this one takes a string and two asts

                             return node
Testing: Quickcheck

ParserTest.hs

-- recursively and randomly generate instances up to a given size limit

arbitrarySizedAst :: Int -> Gen Ast
arbitrarySizedAst m | m < 1 = do i ← arbitrary -- will choose a random Integer
                    x ← elements ["x", "y", "z"] -- will choose random element from the list
node ← elements [LiteralInt i, Var x] -- so put all the non-recursive Ast expressions here
   return $ node
arbitrarySizedAst m | otherwise = do l ← arbitrarySizedAst (m `div` 2) -- get ast half as big
                        r ← arbitrarySizedAst (m `div` 2) -- ditto
                        x ← elements ["x", "y", "z"]
                        ifAst ← arbitrarySizedIf m
                        node ← elements [Plus l r, Sub l r, Mult l r,
                                          ifAst, Let x l r]
                                          -- will choose from if expressions
                                          -- this one takes a string and two asts
   return node

-- break in thirds for mix-fix operators which have three separate sub-asts

arbitrarySizedIf :: Int -> Gen Ast
arbitrarySizedIf m = do x ← arbitrarySizedAst (m `div` 3)
y ← arbitrarySizedAst (m `div` 3)
z ← arbitrarySizedAst (m `div` 3)
   return $ If x y z

parserTest = testGroup
  "parser Test"
   [ testProperty "parse should return the same AST when fully parenthesized" $ (
     \x -> Just (x, "")) == (parse parser $ showFullyParen x)) :: Ast -> Bool),
   
   testProperty "parse should return the same AST when pretty printed" $ 
     (\x -> Just (x, "")) == (parse parser $ showPretty x 0)) :: Ast -> Bool) 
  ]
module EvalTest where

import Test.Tasty (testGroup)
import Test.Tasty.HUnit (assertEqual, assertBool, testCase)
import Test.Tasty.QuickCheck

-- Import all the modules you need to do the test

import Ast
import Parser
import Lang

zero = (LiteralInt 0)
one = (LiteralInt 1)
none = (LiteralInt (-1))
two = (LiteralInt 2)
ntwo = (LiteralInt (-2))
three = (LiteralInt 3)
nthree = (LiteralInt (-3))
four = (LiteralInt 4)
nfour = (LiteralInt (-4))

evalTest = testGroup
  "Eval Test"
  [testCase "Basic Arithmetic" $
    do
      assertEqual "2 + 4 =? " 6 (exec (Plus two four))
      assertEqual "2 + -1 =? " 1 (exec (Plus two none))
      assertEqual "2 - 4 =? " (-2) (exec (Sub two four))
      assertEqual "2 - (-4) =? " 6 (exec (Sub two nfour))
      assertEqual "3 * 2 =? " 6 (exec (Mult three two))
      assertEqual "2 * -2 =? " (-4) (exec (Mult two ntwo)),}
Testing: Quickcheck

EvalTest.hs

evalTest = testGroup
  "Eval Test"
  [   
    testCase "Basic Arithmetic" $  
      do  
          assertEqual "2 + 4 =?\" 6  (exec (Plus two four))  
          assertEqual "2 + -1 =?\" 1  (exec (Plus two none))  
          assertEqual "2 - 4 =?\" (-2) (exec (Sub two four))  
          assertEqual "2 - (-4) =?\" 6  (exec (Sub two nfour))  
          assertEqual "3 * 2 =?\" 6  (exec (Mult three two))  
          assertEqual "2 * -2 =?\" (-4) (exec (Mult two ntwo)),  
    testCase "Compound Arithmetic" $  
      do  
          assertEqual "2 + 4 * 3 =?\" 14  (exec (Plus two (Mult four three)))  
          assertEqual "(2 + -4) * 3 =?\" 6  (exec (Mult (Plus two nfour) three))  
          assertEqual "2 * 3 + 3 * 2 - 4 =?\" 8  (exec (Sub (Plus (Mult two three) (Mult three two)) four))  
          assertEqual "2 * (3 + 3) * (2 - 4) =?\" (-24) (exec (Mult (Mult two (Plus three three)) (Sub two four))),  
    testCase "If Statements" $  
      do  
          assertEqual "if 3 then 4 else 2 =?\" 4  (exec (If three four two))  
          assertEqual "if 0 then 1 else 4\" 4  (exec (If zero one four))  
          assertEqual "if 3 * 0 then 1 else 2 =?\" 2  (exec (If (Mult three zero) one two))  
          assertEqual "if 3 * 2 then 1 else 2 =?\" 1  (exec (If (Mult three two) one two)),  
    testCase "Let Statements" $  
      do  
          assertEqual "let x = 4 in x * 2 =?\" 8  (exec (Let "x" four (Mult (Var "x") two)))  
          assertEqual "let x = 4 * -2 in x - 2 =?\" (-10) (exec (Let "x" (Mult four ntwo) (Sub (Var "x") two)))  
          assertEqual "let x = 2 in let y = x + 1 in y * 2 =?\" 6  (exec (Let "x" two (Let "y" (Plus (Var "x") one) (Mult (Var "y") two))))  
  ]
Testing: Quickcheck

TestingExample2 $ cabal new-test
Build profile: -w ghc-8.6.3 -01
In order, the following will be built (use -v for more details):
  - TestingExample2-0.1.0.0 (test:test) (first run)
Preprocessing test suite 'test' for TestingExample2-0.1.0.0..
Building test suite 'test' for TestingExample2-0.1.0.0..
Running 1 test suites...
Test suite test: RUNNING...
allTests
  parser Test
    parse should return the same AST when fully parenthesized: OK (0.17s)
      +++ OK, passed 1000 tests.
    parse should return the same AST when pretty printed: OK (0.15s)
      +++ OK, passed 1000 tests.
  Eval Test
    Basic Arithmetic: OK
    Compound Arithmetic: OK
    If Statements: OK
    Let Statements: OK

All 6 tests passed (0.32s)
Test suite test: PASS
Test suite logged to: /Users/snyder/Dropbox (BOSTON UNIVERSITY)/Documents/Teaching/CS320/Web/Homeworks and Labs/TestingExample2/dist-newstyle/build/x86_64-osx/ghc-8.6.3/TestingExample2-0.1.0.0/t/.0.0-test.log
1 of 1 test suites (1 of 1 test cases) passed.