CMPSCI 220 Programming Methodology

Expression Evaluator

Overview

In this assignment you will be exercising a number of different techniques that we have studied. In particular, you will be using functional programming, case classes, and pattern matching to implement an *expression evaluator*. An expression evaluator interprets a data structure representing an expression and produces a final result. In particular, your expression evaluator will interpret an expression such as "a + 10 * 4 / 2" and an environment *{ a → 6 }*, which maps variable names to their values, and interprets (evaluates) that expression (and its environment) to produce the result (in this case "26"). In doing so you will only use immutable data structures and case classes representing the expressions. Your evaluator will use Scala's pattern matching facilities to interpret the expression to produce the resulting value.

The program you implement will be capable of reading and evaluating files containing small expression programs (EP). The EPs use a simple language for defining variables and evaluating expressions. Here is an example of an EP program:

```
x = 43;
y = 10;
z = (x + y * 2)/3;
y + z
```

The result of evaluating this EP is the value 31.0 with the environment *{ z → 21.0, y → 10.0, x → 43.0 }*. The evaluator you will implement in this assignment produces this output:

```
R = 31.0 with E = {z -> 21.0, y -> 10.0, x -> 43.0}
```

Where R represents the result of the EP and E is the environment.

Objectives

The objectives for this assignment are as follows:

- To exercise and apply your understanding of immutable data structures.
- To exercise and apply your understanding of functional programming techniques.
- To exercise and apply your understanding of algebraic data types (e.g., class classes).
- To exercise and apply your understanding of pattern matching.
- To exercise and apply your understanding of version control.
- To exercise and apply your understanding of documentation and commenting.
- To understand and apply the construction of larger Scala programs.
- To apply unit testing using the ScalaTest framework.
- To understand and apply program evaluation.
- To use and apply the Option type.

Part 1: Setup

In this assignment you are provided an SBT project containing some starter code. You will need to implement most of the application. You should download the archive file **expr-evaluator-student.zip** associated with this assignment document to your machine. After you download the file you must decompress it to reveal the starter project. After you do this you will see a directory called **expr-evaluator-student**. You must copy this file into your git repository and do a first add/commit. You are welcome to complete this project in your Vagrant environment or on your host machine. If you choose to use your host machine you are responsible for installing Scala, SBT, and Git (Linux, Mac OS X, Windows). You are also welcome to use the EdLab environment by logging in over SSH. The **expr-evaluator-student** folder contains the following:

build.sbt
This is the SBT build file. We have provided you one that works for this project. If you poke around this file you will notice that this project uses Scala version 2.11.4 and has two dependencies. In particular, it depends on ScalaTest and Scala Parser Combinators. You will be using ScalaTest's FunSuite to write tests for this application and Scala's parser combinators are used by our parser to translate an expression program into a data structure (we did this for you).

.gitignore
This is the .gitignore file that ignores directories and files in your git repository. This contains a single entry to ignore the *target* directories in the SBT project. You are welcome to extend this with other entries such as editor backup files.

scalastyle-config.xml
This file is used to configure the Scala Style Checker. You do not need to edit this file, however, you can use it to run the *scalastyle* command in SBT to check your Scala style. We will run this on your code to ensure that you are following proper style and coding conventions.

eps/
This directory contains example programs in the EP language. You are welcome to write your own additional programs to add to those that we give you.

project/
This directory contains additional SBT configuration files. In particular, you will notice the **plugins.sbt** file which provides the definition for the *scalastyle* tool. There is no need for you to touch this file - but, we do encourage you to poke around to see how these files are defined.

src/
This is the root of the source directory containing your code. It contains two directories - **main** and **test**. As you are already familiar with, the **main** directory contains your application code. The **test** directory contains the ScalaTest tests that you will extend and run through SBT. Underneath **main** and **test** is the starting point for the package hierarchy of your application and tests respectively.

Spend a few moments going through the entire folder structure getting to know where things are. Look at the source code we have provided and then proceed to the next section.

Part 2: Building and Testing

In this assignment you will be using SBT (Scala Build Tool) to compile, test, and run your program. First, you should run the SBT tool to get into the SBT console (you can do this from any operating system from any command prompt):

```
$ sbt
>
```

Next, you can compile your source code using the **compile** command:

```
> compile
```

You will notice that if you execute this command without completing Part 3 below you will receive lots of errors. This is because you have not implemented the expression types which are required by the rest of the application. You will need to complete Part 3 before your application will compile.

You can then run the **test** command in SBT to test your application.

```
> test
```

Where it will execute each *test suite* and display the results of the tests found in **src/test**. If there is a failure it will be pretty obvious. Again, if you execute the **test** command without completing Part 3 below you will get compilation errors. You will need to implement the code in Part 3 before your application will compile. Even after you complete Part 3 you will see failures when you run the tests because you haven't provided an implementation yet for the rest of the application. Your job will be to make those failures go away as well as provide your own tests to test your implementation.

Lastly, to run your application you can use the **run** command. This is the output you will see when you have completed this assignment in its entirety:

```
> run eps/01.e
[info] Running cs220.Main eps/01.e
R = 31.0 with E = {z -> 21.0, y -> 10.0, x -> 43.0}
[success] Total time: 0 s, completed Mar 1, 2015 1:10:22 PM
```

The **eps** directory, if you recall, contains example programs in the EP language. The **run** command automatically knows to execute the **cs220.Main** class as it is the only one containing a *main* method. If you had multiple classes with *main* methods it would prompt you for which one. As you can also see you can pass arguments to your program by specifying them after the **run** command. Naturally, you will not see this when you run it for the first time - you need to provide the implementation to get it working!

Remember, if you want to continuously **compile** or **test** you can prepend these commands with ~.

Part 3: Representing Expressions

As mentioned above, you will notice right away that the provided starter code does not compile out of the box. To get the application to compile you will need to supply the implementation of the expression types

that your evaluator will use to represent expression programs. Our expression evaluator expects 8 different expression types to be implemented. In particular:

`Var(name: String) extends Expr`

This represents a variable in our expression language. A variable when used in an expression (e.g., *a* + *b*) will be evaluated to a Value. Thus, in our environment if we have *{ a → 4.0, b → 10.0 }* then *a* would be evaluated to *4.0* and *b* would be evaluated to *10.0*. If a variable is used in an expression (e.g., *a = 4+5*) then it will add an entry into the environment where *4+5* is evaluated to the Value *9.0* and the new environment would be *{ a → 9.0 }*. You will work with environments in the next part.

`Number(value: Int) extends Expr`

This represents a number in our expression language. It evaluates to a Value that the evaluator can evaluate in an expression.

`Add(left: Expr, right: Expr) extends Expr`

This represents an add operation in our expression language (e.g., left + right). It evaluates to the Value of adding the left expression to the right.

`Sub(left: Expr, right: Expr) extends Expr`

This represents a subtract operation in our expression language (e.g., left - right). It evaluates to the Value of subtracting the left expression and the right.

`Mul(left: Expr, right: Expr) extends Expr`

This represents a multiply operation in our expression language (e.g., left * right). It evaluates to the Value of multiplying the left expression and the right.

`Div(left: Expr, right: Expr) extends Expr`

This represents a division operation in our expression language (e.g., left / right). It evaluates to the Value of dividing the left expression by the right.

`Assign(left: Var, right: Expr) extends Expr`

This represents an assignment operation in our expression language (e.g., left = right). It evaluates to the Value *V* of the right expression. In addition, it results in a new environment mapping
{ left → V }

`Program(exprs: List[Expr]) extends Expr`

This represents an expression program which is a list of any of the expressions above. When evaluated it will evaluate to the Value of the last expression. If any of the expressions are assignments then the environment will propagate through each expression

Your Task: You must implement each of the above expression types as case classes in the **src/main/scala/cs220/evaluator/Expr.scala** file. When you open this file you will see additional documentation and a TODO for this part. After you add each of the above case classes you should run the scala REPL within SBT to play with your implementation before moving on. (Note: you will not be able to run the REPL until you have implemented these case classes as you will get compilation failures). In particular:

```
> console
scala> import cs220.evaluator._
import cs220.evaluator._
```

```
scala> Var("a")
res0: cs220.evaluator.Var = Var(a)

scala> Number(4)
res1: cs220.evaluator.Number = Number(4)

scala> Add(Var("a"), Number(4))
res2: cs220.evaluator.Add = Add(Var(a),Number(4))
```

After you implement each of the case classes you must also override the **toString** method so that it displays the expression as an expression in our expression language rather than the default display of a case class. After you override the **toString** method in each of the case classes your output should look like this:

```
scala> import cs220.evaluator._
import cs220.evaluator._

scala> Var("a")
res0: cs220.evaluator.Var = a

scala> Number(4)
res1: cs220.evaluator.Number = 4

scala> Add(Var("a"), Number(4))
res2: cs220.evaluator.Add = a + 4
```

After you implement the case classes and the **toString** method you should be able to compile the rest of the starter code. You should also be able to run the tests for only the expression tests (**src/test/scala/cs220/evaluator/ExprTestSuite.scala**):

```
> test-only cs220.evaluator.ExprTestSuite
[info] ExprTestSuite:
[info] - The Number class exists
[info] - The Var class exists
[info] - The Add class exists
[info] - The Sub class exists
[info] - The Mul class exists
[info] - The Div class exists
[info] - The Assign class exists
[info] - The Program class exists
[info] - Add expression generates the correct infix string form
[info] - Sub expression generates the correct infix string form
[info] - Mul expression generates the correct infix string form
[info] - Div expression generates the correct infix string form
[info] - Complicated expression generates the correct infix string form
[info] Run completed in 442 milliseconds.
[info] Total number of tests run: 13
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 13, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Part 4: Environments

Your next task is to implement an *environment* abstraction. In particular, we want to be able to extend an environment with new variable bindings that result from an assignment. A variable binding is a pair *(V, A)*

where *V* is a variable and *A* is a value that is the result of an assignment *V* = *X* where *V* is the variable and *X* is an expression that has been evaluated to a Value *A*. An environment has three operations:

```
def lookup(v: Var): Option[Binding]
```

The **lookup** operation finds a variable in the environment and returns its binding.

```
def extend(v: Var, a: Value): Environment
```

The **extend** operation extends the current environment with a new environment containing a new binding *(V, A)*. Note that our environments are functional objects (immutable). That is, they are never updated directly - rather, the extend operation returns a new Environment object with the new binding. The result of executing multiple **extend** operations on an environment results in an environment with all of those bindings. Thus, {}.extend(v_1, a_1).extend(v_2, a_2).extend(v_3, a_3) results in the environment $\{ v_1 \rightarrow a_1, v_2 \rightarrow a_2, v_3 \rightarrow a_3 \}$.

```
def toList: List[Binding]
```

The **toList** operation returns the list of bindings in the order they were added to the environment. Thus, $\{ v_1 \rightarrow a_1, v_2 \rightarrow a_2, v_3 \rightarrow a_3 \}$.toList results in List((v_1, a_1), (v_2, a_2), (v_3, a_3)).

An **initial environment** E is an environment that does not contain any bindings (e.g., *{ }*). An extended environment is an environment that has been extended from another environment (either an initial environment or another extended environment) and includes those bindings provided by the **extend** operation.

Your Task: is to implement the **initial environment** and the **extended environment**. We have provided you starter code in **src/main/scala/cs220/evaluator/Environment.scala**. In particular, we have provided an implementation for bindings (**Binding**) and an abstract **Environment** class. We have also provided the skeleton outline of **InitialEnvironment** and **ExtendedEnvironment**. In addition, we have provided the constructor arguments for these classes. You must fill in the implementation for each of the above methods in each of these classes.

We have provided a *factory object* that represents the initial environment at the bottom of this file. This object should be the only way you interact with environments in the rest of the application (our tests use it):

```
object Environment extends InitialEnvironment
```

Again, after you implement the initial and extended environment you should compile and play with your implementation in the REPL. You can then run the test suite for environments and see how you did. You should take a look at the tests in **src/test/scala/cs220/evaluator/EnvironmentTestSuite.scala** to understand what the tests are doing.

```
> test-only cs220.evaluator.EnvironmentTestSuite
```

Notes: The initial environment is easy. There are no bindings in the initial environment, so the **lookup** is easy. It should be pretty clear how to implement the **toList** method on an empty environment. The **extend** method is straightforward, but you should note that you should be returning an extended environment. The extended environment is a little more complicated, but not too hard. The **extend** method is no different from the initial environment. The **lookup** functionality is a little more complicated. You should note that if you do not find a binding in the current environment you will need to lookup the binding in a previous (**prev**) environment.

Part 5: Evaluation and Interpretation of Expressions

In this part you must complete the implementation of the evaluator. The evaluator will receive an expression (**Expr**) and evaluate that expression. For example, given an expression program:

```
a = 5
b = 6
c = a + b * 2
c + 1
```

The evaluator will evaluate this expression to the Value 23.0. In particular, your evaluator will recursively visit the expressions in the tree representation (**Expr**) of the above program and evaluate each expression until it reaches a final result. In addition, your evaluator will manage the proper environment so that variables are substituted with their corresponding Values in the environment. In expression tree form, the above program looks like this:

```
Program(
  Assign(Var("a"), Number(5)),
  Assign(Var("b"), Number(6)),
  Assign(Var("c"), Mul(Add(Var("a"), Var("b")), Number(2)),
  Add(Var("c"), Number(1))
)
```

In order to evaluate the entire program the evaluator will need to evaluate each of the expressions in the program, starting with the first. Thus, first we evaluate *a = 5* followed by *b = 6* followed by *c = a + b * 2* followed by *c + 1*. Each of the assignments result in a new environment being passed to the next expression. To do this properly requires some rules. The rules below have the form *E => V [N]* where *E* is the expression to evaluate, *V* is the resulting Value, and *N* is the environment.

Number(n) => Value(n) [N]
A number is evaluated to a Value with no update to the environment N.

Var(n) => N(n) [N]
A variable is evaluated to the value found in the environment N.

Add(left, right) => Value(eval(left) + eval(right)) [N]
An addition is evaluated by evaluating its left operand and right operand and adding the resulting values with no update to the environment N.

Sub(left, right) => Value(eval(left) - eval(right)) [N]
A subtraction is evaluated by evaluating its left operand and right operand and subtracting the resulting values with no update to the environment N.

Mul(left, right) => Value(eval(left) * eval(right)) [N]
A multiplication is evaluated by evaluating its left operand and right operand and multiplying the resulting values with no update to the environment N.

Div(left, right) => Value(eval(left) / eval(right)) [N]
A division is evaluated by evaluating its left operand and right operand and dividing the resulting values with no update to the environment N.

Assign(left, right) => Value(eval(right)) [N.extend(left, eval(right))]
An assignment is evaluated by evaluating the right-hand side of the assignment and extending the environment N with the new binding (left, eval(right)).

Program(List(e_1, e_2, …, e_k)) => Value(eval(e_k)) [N^k]
A program is evaluated by evaluating each of its expressions, updating the environment for each assignment, and finally evaluating to the last expression e_k with the final environment N^k which is the environment resulting from the last expression e_k.

You should start by looking at the file **src/main/scala/cs220/evaluator/Evaluator.scala**. We have provided some starter code for you:

```
class EvaluationException(msg: String) extends RuntimeException(msg)
```

This class is used to throw exceptions during evaluation. The three cases you should consider are when the program is empty, if the evaluator encounters an unknown Expr type, and when a variable does not exist in the environment.

```
case class EvaluationResult(value: Value, env: Environment)
```

This class is used to represent the result of an evaluation. The result of an evaluation is a Value and an Environment. The rules we defined above dictate what is returned.

```
abstract class AbstractEvaluator
```

This class defines the operations the evaluator can perform. Each corresponds to the evaluation of one of the Expr types. In addition, there is an **eval** method that should pattern match on the type of Expr and dispatch to the proper **evalX** method, where **X** is the type of Expr. Each **evalX** method implements the rules above.

```
class SimpleEvaluator extends AbstractEvaluator
```

This is the class that you need to implement. We have provided the stubs for the methods that you need to complete. You should first start with the simple evaluations (e.g., **evalNumber**, **evalVar**) and then move to the more complicated forms such as **evalAssign**.

```
object Evaluator extends SimpleEvaluator
```

This is an object factory for the SimpleEvaluator that you implement. This is the only object that needs to accessed from the evaluator implementation. Indeed, our tests only use this object to perform testing.

We have also implemented Value in **src/main/scala/cs220/evaluator/Value.scala**. The Value class is simple and you can look at the documentation to understand what it looks like and how it is used.

We have also provided a parser library (**src/main/scala/cs220/parser/Parser.scala**) that can be used to parse a program from a String. This makes it easy to write small programs in the expression language rather than write out the data structure form (which you should use for simple testing in the REPL). To use it you do this from the scala REPL:

```
scala> import cs220.parser._
import cs220.parser._

scala> import cs220.evaluator._
import cs220.evaluator._

scala> val ExprParseSuccess(p) = ExprParser.parse("a = 5\na + 10")
p: cs220.evaluator.Program =
a = 5
a + 10

scala> p
res3: cs220.evaluator.Program =
a = 5
a + 10
```

The `ExprParser.parse` method returns a `ExprParseSuccess` object if it successfully parsed the expression program. You can use destructuring assignment (pattern matching assignment) to pull out the `Program` object. You can then use this to pass to your evaluator for testing. To test your individual evaluation methods you should create simple Expr trees and run your evaluator's **evalX** method on it:

```
scala> Evaluator.eval(Number(5), Environment)
res4: cs220.evaluator.EvaluationResult = EvaluationResult(5.0,{})
```

or

```
scala> Evaluator.eval(Var("a"), Environment.extend(Var("a"), Value(12)))
res6: cs220.evaluator.EvaluationResult = EvaluationResult(12.0,{a -> 12.0})
```

Remember your evaluation methods evaluate expressions into values. As you can see in the second evaluation above we are using `Value(12)` in the environment mapping of `Var("a")`. This is enforced by the fact that all **evalX** methods must return a `EvaluationResult` which is parameterized by a Value and an Environment.

Notes: Most of the rules are very easy. After you get one of the operation expressions done correctly the rest is the same with different operations. The **evalProgram** method is a little more difficult. You need to make sure that you evaluate each of the expressions in order and pass the resulting environments to the next **evalX** method. We found recursion and pattern matching on List to be the easiest and most elegant approach. The evaluation of assignment is not hard, but you must remember to extend the provided environment properly to ensure that you assignment resulted in a new environment. The evaluation of variables is also not too hard, but you need to remember to perform a lookup in the environment and throw an exception if the variable was not found in the environment.

Part 6: Testing

After you complete each of the above parts you should be able to run the entire test suite:

```
> test
[info] ExprParserTestSuite:
[info] - Parsing a number
[info] - Parsing a variable
[info] - Parsing an add expression
[info] - Parsing an assignment expression
[info] EnvironmentTestSuite:
[info] - An empty environment is empty
[info] - An empty environment extended by one binding has length 1
[info] - Extending an environment with (v -> e) will return e on lookup
[info] - An environment has the correct string representation
[info] EvaluatorTestSuite:
[info] - Evaluation of a number expression
[info] - Evaluation of a simple variable expression
[info] - Evaluation of a simple arithmetic expression
[info] - Evaluation of an assignment expression
[info] ExprTestSuite:
[info] - The Number class exists
[info] - The Var class exists
[info] - The Add class exists
[info] - The Sub class exists
[info] - The Mul class exists
[info] - The Div class exists
[info] - The Assign class exists
[info] - The Program class exists
```

```
[info] - Add expression generates the correct infix string form
[info] - Sub expression generates the correct infix string form
[info] - Mul expression generates the correct infix string form
[info] - Div expression generates the correct infix string form
[info] - Complicated expression generates the correct infix string form
[info] ValueTestSuite:
[info] - The Value class exists
[info] Run completed in 261 milliseconds.
[info] Total number of tests run: 26
[info] Suites: completed 5, aborted 0
[info] Tests: succeeded 26, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

We have provided a number of tests, however, we will also apply private tests to your submissions.

Your Task: In this part you are also required to implement 10 additional tests using the ScalaTest FunSuite framework. To do this you need to create a new file in src/test/scala/cs220/Student**IDNUMBER**TestSuite.scala. You must replace **IDNUMBER** with your 8-digit Spire ID number. You can use the other test suite files as a guide to how you should implement yours. Make sure you specify the correct package. Inside this file you need to implement the class:

```
class StudentIDNUMBERTestSuite extends FunSuite
```

Again, replace **IDNUMBER** with your 8-digit Spire ID number. You should verify that your tests run from the SBT console.

We will be using your tests to test your code as well as all the other submissions. We will be giving bonus points to those students who develop tests that break other student submissions.

Part 7: Scala Style Check and Comments

Before you submit you should check your Scala style! To do this you simply boot up the SBT console and run the following:

```
> scalastyle
[info] Processed 6 file(s)
[info] Found 0 errors
[info] Found 0 warnings
[info] Found 0 infos
[info] Finished in 0 ms
```

You should pay attention to what this reports and see if you can fix the problem. We will be running scalastyle on your submission to see if you are following proper Scala style and coding conventions.

In addition, you should include ScalaDoc documentation for each of the methods you implement (except for your test functions. You do not need to provide documentation for the **Expr** classes as these are mostly self-explanatory. You are to provide documentation for the methods in Environment and Evaluator.

Part 8: Submission

After completing each of the above parts you must submit your work to both gitblit and Moodle. To submit to gitblit you should make sure that everything you have done has been committed:

```
$ git status
```

If there are no changes then perform the following:

```
$ git push
```

This will copy your work to the gitblit server.

Finally, submit your work to the activity associated with this assignment on Moodle for peer review. To do this you will need to create a zip archive file for your work. You should zip up only your expr-evaluator-student directory (not your entire github repository!). Here is the command you should execute in vagrant (or your host environment):

```
$ zip -r expr-evaluator-student expr-evaluator-student
```

You can then access the zip file from the shared folder on your host operating system and upload to the Moodle activity. By submitting this assignment you are providing your virtual signature as described on the course academic honesty pledge.