COMPSCI 220 Programming Methodology

Project Assignment 06: 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 take an expression such as "a + 10 * 4 / 2" and an environment *{ a → 6 }*, which maps variable names to their values, and interpret (evaluate) the expression (in the context of the 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 programs 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.

Note: Included with this assignment is a driver, located in the **Main.scala** file. Instructions for how to use the driver to run your code are listed in the "Running the Driver" section below.

Learning Objectives

- 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 dat types (e.g. case classes)
- To exercise and apply your understanding of pattern matching
- To exercise and apply your understanding of documentation and commenting
- To understand the construction of larger Scala programs
- To understand and apply program evaluation
- To understand and apply the Option type.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate to the course staff immediately. Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates'

code. If you are confused about what constitutes academic dishonesty you should re-read the course syllabus and policies. We assume you have read the course information in detail and by submitting this assignment you have provided your virtual signature in agreement with these policies.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For many assignments, it will be useful for you to write additional Scala files. Any Scala file you write that is used by your solution **MUST** be in the provided *src/main* directory you submit to Moodle.
- The course staff are here to help you figure out errors (not solve them for you), but we won't do so for you after you submit your solution. When you submit your solution, be sure to remove all compilation errors from your project. Any compilation errors in your project will cause the auto-grader to fail your assignment, and you will receive a zero for your submission. **No Exceptions**!

Test Files

In the *src/test/scala* directory, we provide several ScalaTest test suites that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. *But, you should be aware that we deliberately do not provide you the full test suite we use when grading*.

We recommend that you think about possible cases and add new test cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero? Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise functions and methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. **Note that we will not be looking at your test cases** (unless otherwise specified by the assignment documentation), they are just for your use and will be removed by the auto-grader during the evaluation process.

You should not modify any source files in the *src/test/scala* directory as the auto-grader will be copying in the original public unit tests and additional private unit tests.

Before submitting, **make sure that your program compiles** with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the *src* directory have been altered in a way that will cause your submission to lose some (or all) points.

Import Project into IntelliJ

Begin by downloading the starter project, extracting it from the zip file, and importing it into IntelliJ. To import your project into IntelliJ you must run IntelliJ. If the IDE opens a previously created project, simply close the IDE window (do not close the IDE). It will then bring up a prompt with the following options:

- Create New Project
- Import Project
- Open
- Check out from version control

You should select **Import Project**. You will then need to find the downloaded project in the file menu and select **OK**. You should then select **Import project from external model** and then highlight **SBT**.

On the next screen make sure *Use auto-import* is selected and then click *Finish*. IntelliJ will then initialize your project (give it a minute) and then show you your project structure on the left.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide unit tests for source files (e.g., classes, methods, functions) that do not yet exist in your code. You can still run the other unit tests.

The project should normally contain the following root items:

- ***src/main/scala***: This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.
- ***src/instructor/scala***: This folder contains support code that we encourage you to use (and must be used to pass certain tests). **You must not change or add anything in this folder**. The auto-grader will replace your submitted *src/instructor* directory with the original during the evaluation process. If you make changes or add/remove anything it can lead to problems in your submission and will result in a 0 for the assignment.
- ***src/test/scala***: The test folder where all of the public unit tests are available.
- ***eps/:*** This directory contains the example programs in the EP language. You are welcome to write your own programs in addition to the ones we provide.
- ***tools/grading-assistant.jar***: This is the grading assistant that you can run to give you an estimate of your score as well as package your project to be submitted to Moodle. More on this later. **Do not remove**.
- ***activator***: This is a Unix helper script that can be used to run the activator interactive build tool. You can use this on Linux and Mac OSX. **Do not remove**.
- ***activator.bat***: This is a Windows helper script that can be used to run the activator interactive build tool. You can use this on Windows. **Do not remove**.
- ***activator-launch-1.3.2.jar***: This is a Scala/Java library that runs the activator tool and is used by the previously mentioned scripts. **Do not remove**.
- ***build.sbt***: This is the build definition file. It provides build information to activator to build your project. **Do not remove**.
- ***.gitignore***: This is a special file used by *git* source control to ignore certain files. You should not touch this file and please **do not remove**. This file may not be present if the assignment does not call for git usage.

Testing, Grading Assistant, and Console

As mentioned previously, you are provided a set of unit tests that will test various aspects of your implementation. You should get in the habit of running the tests frequently to see how you are doing and to understand where you might be going wrong. The ScalaTest testing framework is built-in to the *activator* tool and you can easily run the tests by issuing the following command from the command line:

```
> ./activator test
```

This will compile your code and run the public ScalaTest unit tests. After you compile and run the tests you will notice that a *target* directory has been created. The *target* directory contains the generated class files from your source code as well as information and results of the tests. *Activator* uses this directory so it must not be removed. After you run the tests you can get a grade report from the Jeeves tool by issuing this command:

> *scala -cp tools/grading-assistant.jar autograder.jeeves.Jeeves*

This will print a report to the console showing you the tests you passed, tests you failed, and a score based on the public tests. Although this gives you a good estimate as to what your final score might look like, it does not include points from our private tests. You may run Jeeves as often as you like, however, you must run the tests before your run Jeeves to give you an updated result.

Another very useful approach to test and play with the code you write is the console. You can run the console with this command:

> *./activator console*

This will load up the Scala REPL (read-eval-print-loop). You can type code directly into the console and have it executed. If you want to cut and paste a larger segment of code (e.g., function declaration) you simply type *:paste* in the console, then paste in your code, then type control-D.

Running the Driver

Provided with this project is the file Main.scala, which is a driver for running the evaluator. You can use it to interact directly with your code, instead of relying on the tests.
You can run Main.scala using activator:

> *activator "run eps/01.e"*

Main.scala requires the filepath of the EP file to run, and there are several in the "eps" directory for you to use.

Note: The quotes matter! `activator "run eps/01.e"` is *very* different from `activator run eps/01.e`.

Version Control

Version control is an important aspect of software development. We will be using *git*, a distributed version control system. Most assignments you submit will require you to interact with *git* to record the changes you make to your assignment. Interaction with *git* is just as important as completing the programming tasks so you must take it seriously – the auto grader will evaluate your commits and your will be scored accordingly. The following instructions help you initialize a *git* repository and make your first commit.

IntelliJ and Git

Before you begin modifications to your project you must initialize a *git* repository to record your changes in version control. Click the root folder in your project in the IntelliJ IDE. Next, click the **VCS** menu and select the **VCS Operations Popup...**. Select the **Create Git Repository...** entry. This will allow you to select the folder to create a new *git* repository. Choose the same directory your project resides in. This will initialize a new *git* repository for your project.

After you initialize a *git* repository in your project you must add your project files to the repository. You will notice a menu item at the bottom of the IDE window called **Version Control**. Select that menu item and it will show *Local Changes*.

First, you need to add the project files by drilling down into the *Unversioned Files*. Highlight all the unversioned files and then **Right-click** and select **Add to VCS**. You will now notice that the files have moved under **Default**. Now, you are ready to commit these files.

To commit the files to *git* you simply highlight the **Default** menu entry and click on the **Commit Changes** button (VCS with the up-arrow) on the left. This will bring up a dialog that will allow you to write a commit message. You should give your commit a useful message such as *adding project files* and click the **Commit** button on the lower right. It might ask you to review your files - you can ignore this. Simply click on **Commit**.

This will commit your files to *git* and return you back to the IDE window. You will now notice that there are no more files under **Default**. This means you have successfully committed all your files.

As you work through the project assignment you should *add* and *commit* your files frequently. The *grading assistant* will test that you have at least 10 commits. You will lose points if you do not have at least 10 commits - **this is an important part of the assignment and the experience in this course**.

Command Line and Git

It is also possible to initialize your project with *git* from the command line. To do this you must open a new Terminal (bottom of the IntelliJ IDE) and run the following command from your project directory:

```
> git init .
```

This is assuming you have *git* installed on your system. You can visit http://git-scm.com/download site to install *git*.

Next, to add and commit your changes run the following commands from your project directory:

```
> git add .
> git commit -am 'adding project files'
```

After you do this you should click on the **VCS** menu and select **Enable Version Control Integration...** and then select **git**. This will enable the **Version Control** tool at the bottom of the IDE so you can work with both the command line and integrated *git* support in IntelliJ.

That will do it! You should become familiar with *git* from both the IDE and the command line.

Part 1 – 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 Activator 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:

```
> activator 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) using the command
`activator "test-only cs220.evaluator.ExprTestSuite"`

Part 2 – 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 a chained 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
s**rc/test/scala/cs220/evaluator/EnvironmentTestSuite.scala** to understand what the tests are doing.

```
> activator "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 3 – 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.

Submission

When you have completed the changes to your code, you must run the following command to package up your project for submission:

> *scala -cp tools/grading-assistant.jar submission.tools.PrepareSubmission*

This will package up your submission into a zip file called *submission-DATE.zip*, where *DATE* is the date and time you packaged your project for submission. After you do this, log into Moodle and submit the generated zip file.

After you submit the file to Moodle you should ensure that Moodle has received your submission by going to the activity page for the assignment and verifying that it has indeed been uploaded to Moodle. To further ensure that you have uploaded the correct zip file you should download your submission from Moodle and verify that the contents are what you expect.

We do not allow re-submissions after the assignment due date **even** if you failed to submit the proper file or forgot. There are no exceptions so be very sure that you have submitted properly.