Comp 524 - Assignment 6
Lisp Variables and Functions

Date Assigned: 11/9/2021

Early Completion Date: Monday 11/29/2021 (after Thanksgiving)

Completion Date: Wednesday 12/01/2021 (Last day of classes)

No Late Completion Dates

About half of the tasks in this assignment are extra credit. Some of the extra credit tasks require you to write Lisp code and test it under your interpreter. Others require you to implement the more advanced Lisp features. Depending on what we are able to cover in class, we may add more extra credit. The initial set of tests available for this assignment when it is released will also be augmented until we reach the early submission date.

Tasks include:

- Run A6 checks that further evaluate the correctness of your A5 implementations.
- Implement setting and evaluation of variable (identifier) atoms.
- Implement declaration and calling of lambda expressions.
- Implement declaration and calling of functions named using setq. (extra credit)
- Implement declaration and calling of function expressions. (extra credit)
- Implement currying in Lisp (extra credit)
- Implement let in Lisp (extra credit)

Task 0: Change Class Registry

This assignment is an extension of the previous assignment. Change your class registry to implement ClassRegistryA6 found in the gradingTools.comp524f21.assignment6 package. Implement the methods relevant to your solution, and add stubs for the remaining.

Task 1: Extending Basic Functions

Run A6 checks (F21Assignment6Suite) that further evaluate your A5 implementations. You may get some surprises, so do this asap. Do this first before task 2.

Additionally there are some **extra credit** tests here that check your implementation for extra cases allowed in clisp not required for the last assignment:

and:

should be able to take n^1-n^i parameters and return n^i if all are non-nil or returns the first nil it encounters
or:
should be able to take n^1-n^i parameters and return the first non-nil argument or n^i (the last nil) if all values are nil
cond:
should return nil if there are no parameters. Thus:

```
[29]> (cond)

NIL
```

Normally each cond case/clause has a pair of items, the first being a condition Boolean expression and the second being a return expression.

If a clause has only one item, and it is not a Boolean expression, then the condition is assumed to be true, and the item is the result expression.

Thus:

```
[17]> (cond (t) ((+ 3 4)))

T
[18]> (cond (nil) ((+ 3 4)))

7
[19]> (cond (T))

T
[20]> (cond (Nil))

NIL
```

Task 2: Environment Lookup and Assign

Create a new class that is a subclass of main.lisp.evaluator.environment.AbstractEnvironment and implements the lookup, assign and `newChild()` methods described in the class material. Create stubs for the other unimplemented Environment methods, which will be changed later.

`lookup()` and assign() are discussed in the lecture PPTs. The `newChild()` method should return a new instance of your Environment implementation that passes to the constructor of its superclass the parent environment. This means your Environment implementation should have the first two constructors of `AbstractEnvironment`. The third one can be added later.

(Through your A6 main) call the `setClass()` method of main.lisp.evaluator.environment.EnvironmentFactory to ensure that this class is used for an environment.

Change the class registry to return this class in the appropriate method.

Currently the environment class set in this factory is `NullEnvironment.` This is a final class so not try to extend it. Extend instead `AbstractEnvironment` as mentioned above.

The provided environment and scope interfaces/classes make use of the `java.util.Optional` class. This class was added in Java 8 to help avoid `NullPointerExceptions`. An `Optional` instance may or may not contain a value, and this condition is checked with the isPresent() method. This allows you to return `Optional.empty()` instead of null to denote that a value does not exist or `Optional.of(x)` if it does. If there is a value present in the optional, it can be retrieved with the `get()` method.

Task 3: Setq Evaluator

Implement and register with the `OperationManager` through your `OperationRegisterer` implementation an evaluator for `setq` based on the outline given in the class material. You will need the assign Environment method implemented in the previous task.

Change the implementation of `ClassRegistry` to return this and other new evaluator classes you implement (through the appropriate methods).

Task 4: Variable Eval *(Not an Evaluator)*

To support variables, create a subclass of `main.lisp.parser.terms.IdentifierAtom` that implements the variable `eval()` method described in the class material.

Call the `setClass()` method of `main.lisp.parser.terms.IdentifierAtomFactory` to ensure that class is used for variables.

Change your implementation of your `ClassRegistry` to return this class.

Test your extended interpreter with code to do the following:

1. Assign to variable `A` the value `29`.
2. Assign to variable `B` the value of `A + 13`.
3. Enter the variable name B

Save your previous Lisp test file in some subdirectory of the project.
In a new Lisp test file, add the three s-expressions you entered to the task above.

Task 5: Lambda Evaluator

Implement and register an evaluator for lambda based on the outline given in the class material. Your implementation will use `main.lisp.evaluator.function.LambdaFactory`.

Remember to return this class in the class registry implementation.

The behavior of lambda expressions in clisp and the provided interpreter are slightly different.. When a lambda expression is printed in clisp, the output will be similar to "`#<FUNCTION :LAMBDA (X) X>`", whereas in your interpreter it will be "`LAMBDA (X) X`". This is because, in clisp, LAMBDA is actually a macro that will in many situations be replaced with (function (lambda …)). We will be adding this "function" operator later in the assignment.

Task 6: SExpression with Lambda Application

Create an "SExpression with Lambda Application" evaluator class that extends or delegates to `BasicExpressionEvaluator`, and assign it to the Class property of `ExpressionEvaluatorFactory` class by calling `setClass` in `ExpressionEvaluatorFactory`.

It should override the `eval()` method to support lambda application as described in the class material, falling back on the `BasicExpressionEvaluator` when the SExpression to `eval()` is not a lambda application.

Test your extended interpreter with three applications of lambdas that take 0, 1, and 2 parameters respectively, read all of their arguments (their bodies refer to all parameters), and all return the value 42.

Add to your test lisp file the three s-expressions you wrote for the three applications.

Task 7: Funcall Evaluator with Lambda Application

Write a new evaluator for `funcall` and register it. You should be able to implement it in terms of the lambda application evaluator.

In your `test.lisp` file create an implementation of `derivedSafe` in lisp that follows the expected behavior from the previous assignments. Define `listDrivedSafe` to be a lambda taking 3 parameters.

Task 8: Function Expression and Funcall of it

Implement function expressions and invocations of them directly or through funcall invocations, as described in the background material. Implement the copy Environment method, returning **this.**

Add to your test file the example given in class to motivate function expression:

```
(setq x 5)
(setq timesGenerator
     (lambda (x)
      (function
                (lambda (y) (* x y))
            )
        )
)
(setq twice (funcall timesGenerator 2))
```

Test your function expression using this definition.

Task 9: Curry (Extra Credit)

Implement the curry operator as described in the class lectures.

Implement in your test file the following functions:

```
(setq product3 (lambda (x y z) (* x (* y z))))
(setq product2 (curry product3 1))
(setq identity (curry product2 1))
```

Test your curry functionality using these definitions

Task 10: Lisp toString (Extra Credit)

Convert the `isList()` and `toString()` internal Java functions you wrote in A5 to Lisp. You are, of course, encouraged to write helper functions such as `toStringAsSExpression`() and `toStringAsList`(). This will be important in this assignment as you will be displaying atoms yourself rather than relying on the toString() functions of predefined atoms.

There are two ways to implement this code, which use `defun` or `setq`/`funcall`. When you test the code using your interpreter for this task, you will need to use `setq` and `funcall`. However, `setq` and `funcall` are awkward to use, so you might want to (a) use `defun` first and test it using clisp, (b) convert it to `setq`/`funcall` and test it again using Clisp, and (c) add the code to your lisp file and test it using your interpreter.

Thus (assuming the use of defun):

```
[11]> (isList Nil)

NIL

[12]> (isList (cons 5 Nil))

T

[13]> (fisList (list 2 3 (list 4 5)))

T
```

```
T

[14]> (toString Nil)

"NIL"

[15]> (toString (cons 5 Nil))

"(5)"

[16]> (toString (List 2 3 (list 4 5)))

"(2 3 (4 5))"

[17]> (isList (cons 5 6))

NIL

[18]> (isList (cons 4 (cons 5 6)))

NIL

[19]> (toString (cons 5 6))

"(5 . 6)"

[20]> (toString (cons 4 (cons 5 6)))

"(4 . (5 . 6))"
```

The behavior assuming the use of setq and funcall:

```
[11]> (funcall isList Nil)

NIL

[12]> (funcall isList (cons 5 Nil))

T

[13]> (funcall isList (List 2 3 (list 4 5)))

T

T

[14]> (funcall toString Nil)

"NIL"

[15]> (funcall toString (cons 5 Nil))

"(5)"
```

```
[16]> (funcall toString (List 2 3 (list 4 5)))

"(2 3 (4 5))"

[17]> (funcall isList (cons 5 6))

NIL

[18]> (funcall isList (cons 4 (cons 5 6)))

NIL

[19]> (funcall toString (cons 5 6))

"(5 . 6)"

[20]> (funcall toString (cons 4 (cons 5 6)))

"(4 . (5 . 6))"
```

We will leave it ambiguous whether `(isList Nil)` returns true or not. As you see in the coe above, `(toString Nil)` returns "Nil" rather than "()". So it prints like a regular atom, not a list. If you define `(IsList Nil)` as `T`, then you will have three rather than two cases in toString().

You can use any of the functions in a correct A5 solution including the predefined ones such as `car`, `cdr`, `atom` and `eq` and the ones you needed to define such as cond. You can of course use defun and `setq/funcall`. In addition, you can use the following CLisp functions, illustrated below:

```
[21]> (concatenate (quote String) "hello" "wonderful" "world")

"hellowonderfulworld"

[22]> (write-to-string 4)

"4"

[23]> (write-to-string Nil)

"NIL"

[24]> (write-to-string 4.4)

"4.4"

[26]> (concatenate (quote String) "goodbye world")

"goodbye world"
```

If the first argument of concatenate is `(quote String),` the function concatenates the following arguments, assuming they are Strings. write-to-string can be used to convert any expression into a String. **You should invoke it only on atoms**, not arbitrary s-expressions, otherwise, it will be implementing toString, and more importantly, your code will not work with your lisp interpreter. For the same reason, **you should not use any other Lisp function in** clisp not implemented in your lisp interpreter.

You can first test the functions in clisp before testing them using your interpreter.

Task 11: Let (Extra Credit)

Implement the `let` operator as described in the class lectures. `let` works by creating a scope for a section of the code for instance the following would be the expected results:

(setq x 10)

> 10

(let ((x 4) (y 3)) (+ x y))

> 7

x

> 10

Submission Instructions and Feedback

The same as A5. If you call your test file **test.lisp**, you can keep other lisp files in your project.

Possible Issues

1. Making your environment an abstract class - you will get an instantiation exception.
2. Not calling Option. get() on the value returned by Environment.lookup() or Environment.get().
3. Not casting S-expression to IdentifierAtom before calling Scope.put() or Environment.assign().
4. Copying the main code from the Lisp Interpreter into your main - this is code duplication and will fail if we update the LispIntepreter main. Your main can and should call the lisp interpreter.
5. Overloading rather than overriding methods such as eval(Environment) by defining a method signature such as eval(MyEnvironment). Put the @Override annotation before the method header to make sure you are overriding.
6. Cond can take a "single parameter" in one of it's list arguments (extra credit)
7. Not evaluating arguments to funcall, which can result in a stack overflow.
8. The default Lisp Interpreter has a null environment so you need to set the class of the InterpreterModelFactory to your own environment class