Jedi 2.0

Jedi 2.0 extends Jedi 1.0 by adding blocks and programmer-defined functions.

Here's a sample session:

```
-> {1; 2; 3} + {4; 5; 6}
9
-> def cube = lambda (x) x * x * x
ok
-> cube(3 + 2)
125
-> def f2c = lambda (ft) {def c = 0.55; c * (ft - 32)}
ok
-> f2c(212)
100.0
-> f2c(98.6)
37.0
-> f2c(32)
0.0
-> f2c(-40)
-40.0
-> c
Undefined identifier: c
```

Notes:

- The value of a block is the value of the last expression in the block.
- The body of a function declaration can be a block.
- Identifiers declared in a block—like c = 0.55—are only visible within the block.

Our session continues:

```
-> def timesN = lambda(n) lambda (x) n * x
ok
-> def times5 = timesN(5)
ok
-> times5(3 + 2)
25
-> def compose = lambda(f, g) lambda (x) f(g(x))
ok
-> def twiceCube = compose(lambda(x) 2 * x, cube)
ok
-> twiceCube(10)
2000
-> twiceCube(5)
250
```

Notes:

- Jedi 2.0 programmers can define higher order functions such as combinators.
- The input to a combinator can be an anonymous function.

The session continues:

```
-> def fact = lambda(n) if (n == 0) 1 else n * fact(n - 1)
ok
-> fact(5)
120
-> fact(6)
720
```

Notes:

- Jedi 2.0 programmers can define recursive functions.

And finally:

```
-> def abs = lambda(x) if (x < 0) -1 * x else x
ok
-> abs(-9)
9
-> def delta = 100
ok
-> def isSmall = {def delta = 0.00001; lambda(x) abs(x) < delta}
ok
-> isSmall(99)
false
-> isSmall(-99)
false
-> isSmall(0.000009)
true
-> delta
100
```

Notes:

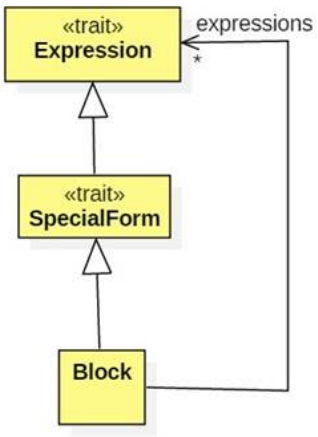
- Jedi 2.0 uses the static scoping rule. isSmall uses the delta of its defining environment, not the delta of its calling environment.
- Lambda creates closures, functions that remember their defining environment.

Implementing Blocks

A block is a special form that encapsulates a list of semicolon-separated expressions:

block ::= {exp_1; exp_2; ...; exp_n}

Here's the UML diagram:



Executing a block executes each expression in the block, then returns the value of the last one:

```
-> {write(1); write(2); write(3); 1 + 2 + 3}
1
2
3
6
-> {def x = 1; def y = 2; def z = 3; x + y + z}
6
```

```
-> x
Undefined identifier: x
```

Jedi 2 Environments

In the second example, 3 constants were declared, placing the bindings x = 1, y = 2, and z = 3 in the environment. But when the block was finished executing, the bindings disappeared. Where did they go? Take a look at Environment.scala. Notice that an environment can have an extension that is also an environment. The extension have an extension, and so on. Thus, an environment is actually a linked chain of hash maps ending with the global environment defined way back in the console. Also notice that the environment's override of the apply method—which normally retrieves values from a hash map—searches through the entire chain of hash maps.

The block's execute method creates a temporary extension of its input environment:

```
tempEnv = new Environment(env)
```

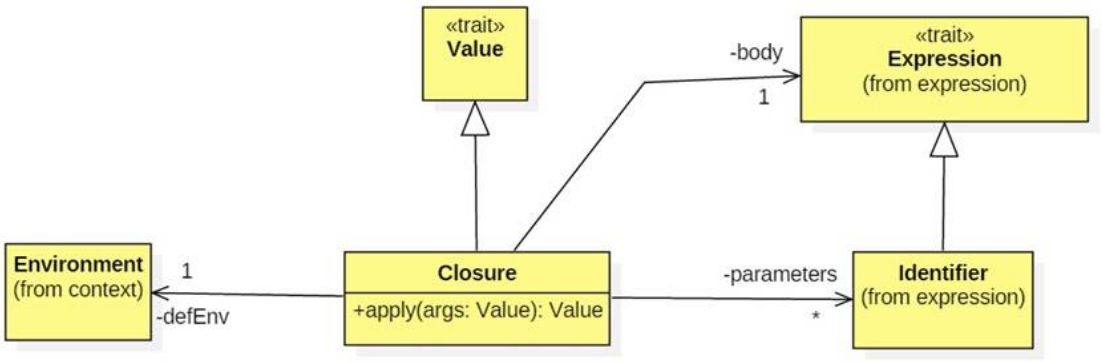
All of the block's expressions are executed relative to this temporary environment. This is where the bindings for x, y, and z in the example above will reside. Since tempEnv is local to execute, it becomes garbage when execute finishes and will eventually be swept away by the garbage collector.

P.S. Make sure Identifier.execute(env) is calling env(this) i.e., env.apply(this)) instead of the code presented in class for Jedi 1.0.

Implement Closures

Jedi 2.0 supports functional programming and implements the static scope rule. Therefore, functions are values that remember their defining environments. In other words, functions are closures.

A closure has three fields: parameters, body, and defining environment:



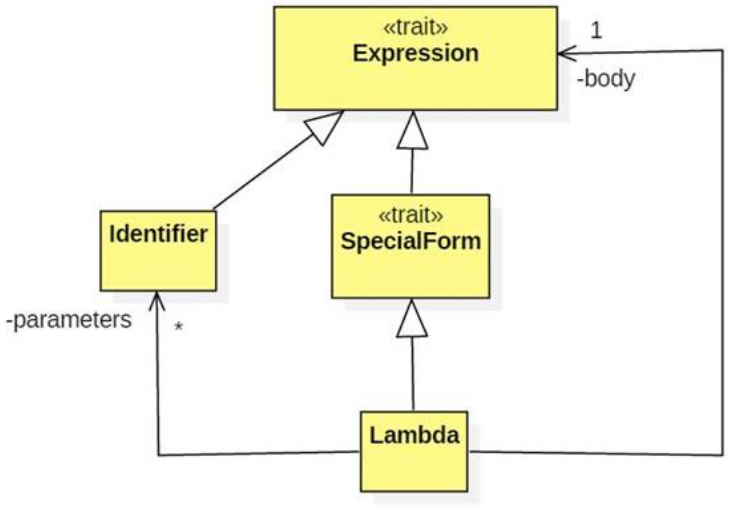
A closure's apply method expects an argument list as input. Like a block, it creates a temporary extension of its defining environment. It binds its parameters to its arguments (make sure the lists have the same size) in the temporary environment. (Use the handy Environment.bulkPut method for this).

Implement Lambda

A lambda expression has the form:

```
lambda (x, y, z, ...) expression
```

Here's the UML class diagram:



When a lambda is executed, it returns a closure. (Where does the closure's defining environment come from?)

Modify FunCall

Recall FunCall from Jedi 1.0:

```
case class FunCall(val operator: Identifier, val operands: List[Expression]) extends
Expression {
   def execute(env: Environment) { ... }
}
```

FunCall.execute begins by eagerly executing its operands to produce its arguments. In Jedi 1.0 execute could assume that its operator named a pre-defined ALU function. This could still be true in Jedi 2.0, but another possibility is that operator names a closure created by a lambda. In this case the closure's apply method must be called.

Implement Jedi2Parsers

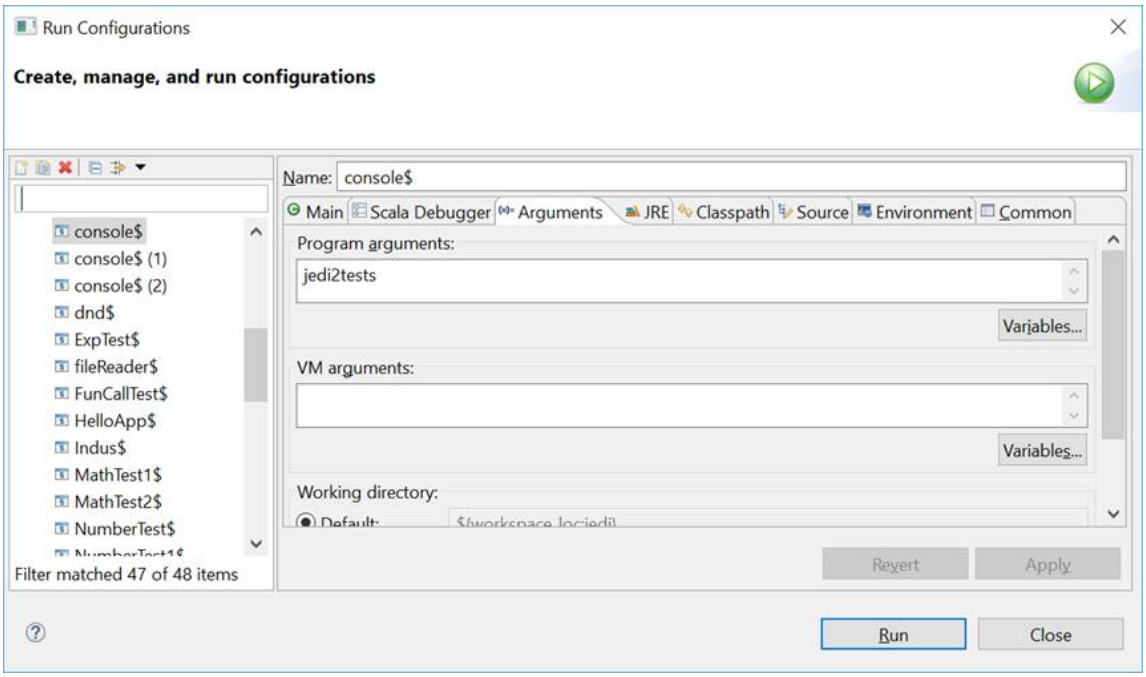
Complete Jedi2Parsers.scala. Note that it extends Jedi1Parsers but overrides the definition of the term parser to include lambdas and blocks.

P.S. Be sure to swap out Jedi1Parsers for Jedi2Parsers in your console.

Testing Jedi 2.0

In Eclipse add a file called jedi2tests.

In the Run Configurations dialog, set the program arguments to this file:



When the console runs, it should execute the expressions in this file instead of calling repl.

Here are my outputs.

Extra Features

Jedi 2.1 allows users to alter the scope rule and parameter passing mechanisms.