Exercise 09

Task 1: OO with local variables

Extend one of the OO interpreters from the lecture with support for local variables (Let) in method bodies.

Answer:

Make `let` a normal expression: extend class `Expr` with the usual case class for `Let` from previous interpreters, add a case for `Let` to the `interp` function.

Task 2: OO versus functional programming

1. Show how to encode first-class functions with objects.

Answer:

There are several possibilities. A naive which would work in our little OO language would be:

Create a class with an apply method that contains the function's body and depends on the function's parameters (parameters of the apply method. Instantiate an object of that class - the resulting object is the first-class function. One can invoke the function by invoking the apply method of the object.

```
   val functionclasses = Map(
           'Zero -> Class(Nil, Map()),
           'Succ -> Class(List('pred), Map()),
           'add1 -> Class(Nil, Map('apply -> Method(List('arg),
             New('Succ, List(Id('arg))))))
           )
 val exp = Invoke(New('add1, Nil), 'apply, List(New('Zero, Nil)))
```

Here, `New('add1, Nil)` is the first-class function which we can pass around; we can call it by invoking its `'apply` method.

Scala currently also implements functions roughly like that - see Scala traits Function1 - Function22, http://www.scala-lang.org/api/current/scala/Function1.html etc.

2. Show how to encode objects with first-class functions. You may assume there is support for algebraic data types in the functional language.

One simple way is to encode an object via a first-class function that takes an element of an ADT m that encodes the objects' method names. To implement method dispatch, the function's body does a case distinction on m and then returns a lambda-expression in each case with the method's functionality. To implement inheritance, we can use an "else"-clause in the case distinction, which passes m on to a parent object if m was not caught before in the case distinction. See for example http://cs.brown.edu/courses/cs173/2012/book/Objects.html , 10.1.2: Objects by desugaring for a longer explanation.

To encode state for objects, use a language with state (e.g. adding boxes as we have seen before). There are slightly nicer ways for encoding objects via first-class functions, e.g. using records and existential types (see for example Pierce, Types and Programming Languages).

Simple example using Scala:

```
sealed trait M
        case object Add1 extends M
        case object Sub1 extends M
        case object Mul2 extends M

        val o1 = (m: M) => m match {
        case Add1 => ((x : Int) => x + 1)
        case Sub1 => ((x : Int) => x - 1)
        }

        val o2 = (m: M) => m match {
        case Mul2 => ((x : Int) => x * 2)
        case _ => o1(m)

    o2(Add1)(4) //5
    o2(Mul2)(4) //8
    o1(Mul2)(4) //MatchError
```

3. Discuss advantages and disadvantages of OO versus functional programming. What are good application scenarios, respectively?

Answer: Some ideas from e.g. http://stackoverflow.com/questions/2078978/functional-programming-vs-object-oriented-programming:

- Object-oriented languages are good when you have a fixed set of *operations* on *things*, and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone.
- Object-oriented languages are especially good for re-using/extending existing data structures.
- Functional languages are good when you have a fixed set of *things*, and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone.
- Pure Functional programming is good for concurrency: compose side-effect free parts of your program!