Handin 5 - Progsprog

Exercise (102)**.**

When you have integrated the extensions described above, and made sure your Scala code compiles, add comments at all steps in the following parts of the interpreter to explain how they work:

- case CallExp in the eval function
- case NewObjExp in the eval function
- case LookupExp in the eval function
- the evalBlock function
- the evalArgs function
- the getValue function
- the rebindClasses function

Exercise (103)**.**

Add calls to trace at the beginning of case LookupExp and case NewObjExp in the interpreter. Then run examples/counter.s, examples/intlist.s, and examples/pingpong.s using the MiniScala interpreter with option -trace enabled. Explain the order of the resulting sequence of lookups and object creations.

Exercise (106)**.**

In weeks 4, 5, and 9, we have seen different ways of defining and implementing operational semantics of recursive functions, and in week 8, we saw how to encode recursive function definitions using a fixed-point combinator. Let's study recursion a bit further using the factorial function as an example. First, here is our ordinary MiniScala definition of factorial:

```
1 def fac(n: Int): Int = if (n == 0) 1 else n * fac(n - 1)
```

It uses def, which permits recursion as we have seen in week 4. Here is another way to implement factorial:

```
1 val fac2 = (f, n) => if (n == 0) 1 else n * f(f, n - 1);
2 val fac = (n) => fac2(fac2, n)
```

Here, fac2 is a helper function for fac. Notice that neither of them use def (which permits recursive definitions) but val (which doesn't permit recursive definitions), and nevertheless, the resulting function is recursive! To understand how this works, try evaluating fac(3) by

hand, step-by-step. (If you have implemented the -trace option in your MiniScala interpreter, you can use that to check your answer.)
Explain, in your own words, how the new fac function works, in particular how it obtains recursion.

Let's start by evaluating `fac(3)` by hand in order to gain some intuition.

$$\begin{aligned}
\texttt{fac}(3) &= \texttt{fac2}(\texttt{fac2}, n)(3) \\
&= \texttt{fac2}(\texttt{fac2}, 3) \\
&= (f, n)(\texttt{fac2}, 3) \\
&= 3 \cdot \texttt{fac2}(\texttt{fac2}, 2) \\
&= 3 \cdot 2 \cdot \texttt{fac2}(\texttt{fac2}, 1) \\
&= 3 \cdot 2 \cdot 1 \cdot \texttt{fac2}(\texttt{fac2}, 0) \\
&= 3 \cdot 2 \cdot 1 \cdot 1 = 6
\end{aligned}$$

.

While direct recursion isn't implemented in `ValDecl`, we can still achieve it with this setup. What allows to achieve it is the definition of `fac2`, which passes some function `f` along in each call, evaluating it repeatedly until `n=0`. Since the function is passed along it can be accessed at each step. The neat trick is, that while self-referencing isn't allowed for `ValDecl`s, `fac2` doesn't explicitly reference itself, it instead references some generic function `f`. The recursion is then achieved by defining `fac`, which gives `fac2` as an argument to itself.