# CS411 Problems 6

A. Demers

26 Apr – due 3 May

The two questions in the assignment both treat the untyped object calculus defined in Chapter 6 of [Abadi and Cardelli] and discussed in lecture. The syntax is reproduced here:

$$
\begin{aligned}
e \;::=\; & x \\
& \mid [\; \ldots \; m_i \sim \varsigma(x_i)e_i \; \ldots \;] \\
& \mid e.m \\
& \mid e.m \leftarrow \varsigma(x)e
\end{aligned}
$$

Here $m$, $m_i$, ... are method names, while $x$, $x_i$, ... are parameter names bound to **self** when a method is invoked.

## 1 Static Scope Evaluation Rules

In class we presented evaluation rules for the untyped object calculus given above. Our rules implemented dynamic scope, which admittedly was a bad idea. Fix this. That is, give static scope evaluation rules for the untyped object calculus.

□

## 2 Compiling the Object Calculus

In this question you are to define a "compile" function for untyped object calculus terms, analogous to the compile function we presented in lecture for IMPX expressions.

Assume a target language similar to the one we used in lecture. Specifically, a variable or memory location contains either an *int* or a pointer to an *item* in a (garbage-collected) heap. An *item* is a record with the following fields:

link: pointer

len: int

id[0..len]: int

val[0..len]: pointer

An item is created by a call of the form

**newitem**($n$)

This returns a pointer to a newly created item with the *len* field set to $n$ and all other fields set to zero (for *int* fields) or **nil** (for pointer fields). If you wish, you may provide a second parameter from which the *link* field will be initiated; e.g.

**newitem**($n, v$)

which is convenient for code sequences like

sp $\leftarrow$ **newitem**(1, sp); sp.val[0] $\leftarrow$ ...

Assume that any parameter or attribute label can be represented as a nonzero integer.

The target language provides arithmetic, "pointer-chasing," **if** - **then** - **else**, simple loops (if you want), and labels with computed **goto** statements. For example, a sequence like

```
    sp.link.val[1] ← L1;
    goto sp.val[0];
textrmL1 :
```

might appear in the code generated for method invocation.

You should pattern your compile function after the one presented in lecture. You will almost certainly need an eval-stack (represented as a list with head sp) and an environment stack (represented as a list with head ep). Object values themselves should be stored as items in the heap; and you'll probably want to store each method as a separate item containing an $\langle$ep, ip$\rangle$ pair.

Just as for IMPX, the compile function should be defined by induction on the structure of the term being compiled; and it will need an additional argument representing the lexical environment. You don't need to prove anything about your function, but you should describe it in enough detail that we are able to grade it sensibly.

□