C/C++ 2015/16 programming exercise 4

This exercise explores some advanced features of C++, combining both forms of polymorphism: virtual functions and templates.

Here is the grammar of a simple language of expressions (loosely based on Lisp syntax):

$$
\begin{array}{lcll}
E & \rightarrow & c & \text{(constant)} \\
E & \rightarrow & x & \text{(variable)} \\
E & \rightarrow & \texttt{(} \otimes L \texttt{)} & \text{(operator application for some operator } \otimes\text{)} \\
E & \rightarrow & \texttt{(= } x\ E\ E\texttt{)} & \text{(let binding)} \\
 & & & \\
L & \rightarrow & E\,L & \text{(expression list)} \\
L & \rightarrow & &
\end{array}
$$

Abstract syntax trees for this grammar can be represented in C++ using the Composite pattern as a collection of classes. Then expression evalution can be implemented using virtual functions.

The values in this language could have various types. In the simplest case, they could be restricted to integers, and the operators $\otimes$ could include $+$, $-$, and so on.

But there is nothing in the language that restricts it to only integers. If, for example, closures are added to the language, then a more complex value type would be needed.

Thus we could aim to write a more abstract representation that is *parametric* in the type of values that expressions can produce. To so, we use C++ templates in the data structure.

We can also make the operator application rule more generic. Rather than hard-wiring in a particular operation like addition, we can construct operator applications where a binary operator (such as addition) and a unit element (such as 0) are supplied and are then folded into the list L of arguments (like fold in OCaml or Haskell). That is, start with the unit element and then apply the binary operators to the list elements in succession.

The template declarations for such a parametric AST type are in:

`http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/templatesast.h`

Your task

Implement the `eval` function for all derived classes as member functions, using templates appropriately, in a file called `templatesast.cpp`. For example, you need to write a definition

```
template<typename V>
```

```
V Constant<V>::eval(env<V> *p)
{
  // code to evaluate a Constant
}
```

A sample main file is in
`http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/templatesastmain.cpp`
You are encouraged to make your own test cases for various value types.
Your code must compile on the lab machines using

```
clang++ -std=c++11 -Werror -o templatesast templatesastmain.cpp
```

Note that, due to separate compilation issues for templates, the `include`s are organized differently from previous exercises: the main file includes your code after including the .h file. Everything is compiled as a single file.

Marking

This exercise counts for 5% of the module mark.

2 points are given if your `eval` functions work on tests without let bindings.

3 more points are given if your `eval` functions also work on tests with let bindings.

Make sure to test your code on the lab machines as above before submitting. Code that does not compile there gets 0 marks, no matter if it happens to work in your IDE. Your functions should not print anything, as that would confuse the marking script.