This assignment is worth 5\% of the final mark of the declarative programming course. The code (including inline documentation) is to be emailed to the departmental secretary Mr Vincent Sammut by Monday 8th May 2006 by 11am. You are also to confirm with Mr Sammut that the email was received and hand in a signed copy of the plagiarism form by the above mentioned time. You can work in pairs, so please ensure that your names appear in the code and that you both sign the cover sheet. Assignments handed in late will be marked to zero. This assignment is based on the previous assignment, so please refer to the previous assignment sheet to be able to answer these questions. Question 4 is the difficult question, so ensure you dedicate enough time to solve it.

The Department of Computer Science and AI takes a very serious view on plagiarism. Refer to the departmental website on plagiarism for more details:

\[
http://www.cs.um.edu.mt/resources/plagiarism/
\]

\textit{Answer everything.}

1. We would like to be able to show terms of type \texttt{Exp} as a string which is a valid Pascal expression. Make \texttt{Exp} an instance of type class \texttt{Show}. Note that without brackets, you may end up with expression such as \texttt{(Val 1 :+: Val 3) :*: Val 5} to be shown incorrectly as \texttt{1+3*5}. Fix this by bracketing all subexpressions.

2. Similarly, make \texttt{Prg} an instance of type class \texttt{Show}, outputting a program with Pascal-like syntax and meaningful indentation. For example, the program \texttt{factorial 7} would be shown as:

\begin{verbatim}
var counter:int;
var result:int;
result := 1;
counter := 7;
while (counter > 0) do
begin
    result := result * counter;
counter := counter - 1;
end;
println ("Factorial of 7 is:");
println (result);
\end{verbatim}
Start by defining an auxiliary function `show'` which, given a program returns the output as a list of lines. This makes indenting a chunk of code easy (by mapping onto the list a function which adds the indent to the beginning of the line). Then define `show` in terms of `show'`.

3. Define a function `renameVar` which given a program and a pair of variable names, returns the same program but with all free instances of variables matching the first name renamed to the second name. A free instance of a variable is a use of the variable occurring before the variable has been declared.

4. Define a function `uniqueNames`, which given a program, returns the same program but with no variables sharing a common name. You may assume that the variable names `fresh1, fresh2, etc` are not used in the program. The trick is to write an auxiliary function `uniqueNamesAux` which (i) keeps count of which fresh variables have been used (by taking an integer parameter `n` which says that `freshn` and beyond have not yet been used), (ii) keeps track of the variables currently in use by the program (by taking as a parameter a list of the variables in scope), and which (iii) every time it encounters a new declaration, renames it if the variable name is already in use. The function should return a pair — the next fresh variable number after all the renaming is carried out, and the transformed program.

```haskell
uniqueName program = program'
  where
    (_, program') = uniqueNameAuxiliary 1 [] program

uniqueNameAuxiliary n vars program = ...
```

5. Define a function `declarationsFirst` which, given a program returns another programs with declarations moved to the beginning of the program. Make sure that you rename variables to avoid clashes first.

6. Hence or otherwise, write a function `toPascal`, which translates a program into correct Pascal syntax.