Project 1: Programming in ML — due Wednesday 28 March

Total number of points available on this project is 140. Full credit is equivalent to 100 points.

We've learned that β-reduction of terms of the λ-calculus by hand is tedious and error-prone.

The goal of this project is to develop an interpreter for the λ-calculus that will automate the reductions. This program will follow literally the rules for β and η conversion, and the rules for substitution. The internal representation of λ-terms is essentially the same as the textual representation, though the data type makes the bracketting structure apparent, and pattern-matching easier. We call this kind of system "string rewriting".

We must first specify the internal representation for λ-expressions. The following type is suggested (but you are free to use a different type):

$$\textbf{datatype}\ \mathit{Expr} = \mathit{Var}\ \textbf{of}\ \mathit{string} \mid \mathit{Abstraction}\ \textbf{of}\ \mathit{string} * \mathit{Expr} \mid \mathit{Application}\ \textbf{of}\ \mathit{Expr} * \mathit{Expr}$$

Tasks:

1. (5 pts.) Write a function *lexpPrint* to convert an *Expr* into a character string in the fully-bracketted syntax for λ-expressions. Use the character # for λ.
2. (5 pts.) Write a function *lexpPrettyPrint* to convert an *Expr* into a character string in the loose syntax for λ-expressions.
3. (10 pts.) Write a function *lexpParse* to parse a character string containing the text of a λ-expression (which may be in the loose syntax) and convert it into an *Expr*.
4. (10 pts.) Implement an environment mapping identifiers to λ-expressions, with type *string* → *Expr*. There should be a mechanism to build new environments out of old ones by introducing new definitions for an identifier.
5. (10 pts.) Implement a top-level environment and appropriate input/output handling, so that it is possible interactivelly to add new definitions of named λ-expressions, and ask for expressions to be reduced and printed. It should be possible to print all intermediate steps of the reduction as well.
6. (40 pts.) Implement a generic reduction framework, in which several notions of conversion can be specified, and different reduction orders can be specified. For instance, one should be able to specify that either β and η conversions, or both, should be tried, and that applicative-order or normal-order reduction should be used.

   Implement β and η conversions, as well as a conversion that we'll call ζ that performs the "macro-expansion" of a name defined in the top-level environment.
7. (10 pts.) Test your program by reducing the several expressions from homework assignments: **SKK**; **K**(**SII**); **S**(**S**(**KS**)(**KI**))(**KI**); **SSSSSSS**.
8. (10 pts.) Structure your ML code well. Document your design and your program well. Submit program listing and transcript of a sample session including at least the reductions above.
9. (20 pts. extra credit) Do what is necessary so that strictness annotations can be added to the λ-bindings: even if normal-order reduction is otherwise used, β-conversion of a redex $(\underline{\lambda} v.B)E$ will cause E to be reduced first, before substitution into the body B.
10. (20 pts. extra credit) Write a function *lexpInterpretivePrettyPrint* to convert an *Expr* into a character string in the loose syntax for λ-expressions, but in a form that is easier for humans to read, as follows. Recognize subexpressions that are exactly as the Church numerals and the truth values and print them as 0, 1, etc. Also recognize `cons` and `nil` so that lists can be printed as `(cons 2 (cons 1 nil))`, for instance.