HW 15 **Prof. Caldwell**
Due: 27 October 2011 **COSC 3015**

1 Parsing lambda terms

In class we presented a type to represent the abstract syntax of of lambda terms. In this assignment you will write code to parse terms and to build elements of the data type `Term` defined as follows:

```
data Term = V String
          | Ap Term Term
          | Abs String Term
 deriving Eq
```

We also presented some concrete syntax for this language which is slightly modified here to simplify the assignment. A right recursive BNF specification for the language of lambda terms with pairing and spreads is given as follows:

$$
\begin{array}{lcl}
\textit{term} & ::= & \textit{term1}\ \ (\textit{term} \mid \epsilon) \\
\textit{term1} & ::= & \quad \textit{identifier} \\
 & & \mid \text{"}\textbf{lambda}\text{"}\ \ \textit{identifier}\ \text{"."}\ \ \textit{term} \\
 & & \mid \text{"("}\ \ \textit{term}\ \ \text{")"}
\end{array}
$$

A *term* is a white-space separated sequence of one or more elements of the *term1* syntactic class. If there is more than one *term1* in the sequence then it is interpreted as an application (an `Ap`) which associates to the left[1].
Thus:

```
*Lambda_parser> applyParser pTerm "x y z"
Ap (Ap (V "x") (V "y")) (V "z")
```

Elements of the syntactic class *term1* take one of the following forms: it is an *identifier*; it is an abstraction which starts with the keyword **lambda** or it is a *term* which is enclosed in parenthesis. You can build your *identifier* parser with the parser `identifier` (which is in the file *Parser.hs*) though you need to make sure that the keyword `lambda` is *not* included among the class of identifiers for this language.

Exercise 1.1. Write `Term` parsers `pV` for variables, `pAbs` for abstractions (lambda terms), a parser `pParenTermfor` terms enclosed in parenthesis, and the `pTerm1` parser for the class *term1*.

Exercise 1.2 (Extra credit) Extend your parser for abstractions so that it allows one or more variable names between the keyword `lambda` and the ".". Do not change the type `Term`, simply generate nested `Abs` terms as the following examples show.

```
*Lambda_parser> applyParser pTerm "lambda x . x"
Abs "x" (V "x")
*Lambda_parser> applyParser pTerm "lambda x y . y x"
Abs "x" (Abs "y" (Ap (V "y") (V "x")))
*Lambda_parser> applyParser pTerm "lambda x y . z"
Abs "x" (Abs "y" (V "z"))
```

[1]Since left associativity is a bit tricky to implement in the parser, I have provided code for the *term* parser.