CS2104 Programming Language Concepts
Laboratory 5 : Parser for Lambda Expression in Scala

(Due: 16th November 2015 7pm)

Please submit your solution into IVLE workbin as a single Scala file. You are to use the the Parser combinator we have developed in Tutorial 9 as your starting point.

You are asked to design a parser for lambda calculus with let-construct of the following form:

```
<lam-expr> ::= <identifier>
             | <lam-expr> <lam-expr>
             | ( <lam-expr> )
             | let <identifier> = <lam-expr> in <lam-expr>
             | \ {<identifier>}+ . <lam-expr>
```

For applications, you must make sure that it associates to the left. (Hint: you may make use of rep1 to try get a number of lambda terms before forming the nested binary applications.) For lambda abstraction, you must make sure that it extends as far as possible to the right. We have already provided the following case class to model the abstract syntax for your lambda term, namely:

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class FApp(f: Term, v: Term) extends Term
case class Let(n: String,f: Term, v: Term) extends Term
```

Your task is the complete the parser and also to support multi-arguments lambda abstraction of the form `(\x y z. x (y z))` which will be parsed as:

```
Fun("x",Fun("y", Lam("z",FApp(Var "x",FApp(Var "y",Var "z"))))))
```

(Fun)

We have given you some test cases. Please design additional test cases for your parser.

Your code must work with the following test harness function:

```
def parse_lambda(x:String):Option[Term] = ..
```

In case of parsing errors, please return None.

BONUS Section (20%)

Write an interpreter for your lambda term that should work with the following test harness. This interpreter should use call-by-value semantics to reduce a given term to a normal form. It should not attempt to evaluate expressions under the lambda abstraction function. You will need to support beta-reduction and alpha-renaming operations to get your interpreter to work.

```
def eval_to_value(x:Term):Term = ..
```