CS2104 Programming Language Concepts
Laboratory Assignment 5 : Two Interpreters for Lambda Calculus

(Deadline : 12th November 2013 8pm)

Please submit your solution into IVLE workbin as a single OCaml file. You can use the utilities provided by globals.ml, debug.ml, gen.ml, etc but must not change their implementation. These utility modules need not be submitted.

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

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

The first interpreter is based on *call-by-name* which chooses the leftmost-outermost redex for reduction. The second interpreter is based on *call-by-value* which will choose the leftmost-innermost redex for reduction. The call-by-name interpreter will terminate more often than the call-by-value interpreter as its strategy of reduction is one that is based on laziness, where it reduces a term if it is definitely needed. However, the substitution mechanism used by call-by-name can lead to some repeated computation in addition to the use of suspensions. Your task is to implement the following functionalities that could be used to support the interpretation of lambda calculus:

(i) Implement a `Lambda.rename v nv t` method which will rename a variable v in a given term t to a new fresh variable nv. For example, during the following evaluation: (\ x . (\ y. y x)) y =eval_val(1)=> (_1.(_1 y)). we have to rename (\y. y x) ➔ (_1. (_1 x)) to avoid a name clash, and this was achieved by renaming via
`Lambda.rename y _1 (y x) ➔ (_1 x).`

(ii) Implement a `Lambda.subst v t1 t2` method that would replace every free occurrence v of a term t2 by a new sub-term t1. This substituition mechanism is used to support beta-reduction of let-construct and lambda abstraction:
```
a.  let v=t1 in t2 ➔ subst v t1 t2
b.  (\ v. t2) t1 ➔ subst v t1 t2
```

(iii) A method `Lambda.has_redex` that will attempt to check if there are reducible expressions in a given lambda term. This method is currently provided for you. Please study it to see how we count the number of outermost redexes.

(iv) Implement a method `Lambda.one_step_by_name` that will choose the leftmost outermost reducible expression for beta-reduction.

(v) Implement a method `Lambda.one_step_by_value` that will choose the leftmost innermost reducible expression for beta-reduction.

(vi) We use an object `(new name_generator "")` as a fresh name generator.

Complete the implementation for these two lambda interpreters. We provide some trace files (named as "trace_*") to help you with debugging each of the above methods