Programming Paradigms
Seminar 4

One of the smallest universal programming language is known as the lambda calculus. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of computability. Lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent in power to the Turing machine.

The central concept in lambda calculus is the "expression" which can either be an identifier (or variable), a function or an application, as captured by the following BNF grammar:

```
<expr>        ::= <id> | <function> | <application>
<function>    ::= λ <id> . <expression>
<application> ::= <expression> <expression>
```

An example of a function is (λ x . x) which denotes the identity function. The identifier after "λ" is the parameter of the function, while the expression after the "." is the body of the function. Functions may be used in the left hand side of an application. An example is (λ x . x) y which could be evaluated by substituting the argument of the function application for its parameter in the function body. In the above example, we get:

(λ x . x) y = [y/x]x = y

Parenthesis may be added to disambiguate this notation. For example (λ x . x x) (λ x . x) denotes a function that is applied with an identify function as argument. It can be reduced as follows:

(λ x . x x) (λ x . x) = [(λ x . x)/x] x x = (λ x . x)(λ x . x)
= [(λ x . x)/x] x = (λ x . x)

For ease of implementation, we can capture the terms of lambda calculus with the help of abstract syntax tree in Oz. In particular, let us assume that our lambda terms can be built using the following data type:

```
<Expr>  ::=  <Id> | lam(<Id> <Expr>) | apply(<Expr> <Expr>)
                  | let(<Id>#<Expr> <Expr>)
```

Note that `<Id>` denotes an identifier that shall be specified as an atom. We also provide a `let` construct as a syntactic sugar to bind an identifier to its first expression whilst returning its second expression as result. In this lab assignment, we will attempt to write a library of functions that can be used to manipulate lambda terms.

Free variables

Each variable that is captured in either `lam` or `let` construct is said to be a "bound" identifier, while those that are not captured are known as "free" variables. For example, given `lam(x apply(y x))`, the variable `y` is free while variable `x` is bound. Similarly, in the term `apply(x let(x#y x))`, the first occurrence of `x` and `y` are said to be free, while the last occurrence of `x` is bound. Write a function, called `FreeSet` which would return all free variables in an expression. Some examples are:

```
{FreeSet apply(x let(x#y x))}          % returns [x y]
{FreeSet apply(y apply(let(x#x) y))} % returns [y] or [y y]
```

Environment/Mapping

During the evaluation of a lambda term, we often build an environment for the identifiers that map each given variable to its corresponding argument. We can capture this mapping as a list of pairs of the form <Env>=<List>(<Id>#<Expr>). To support this environment data structure, we may provide the following functions:

```
IsMember ::  {<Env> <Id>} → <Boolean>
Fetch    ::  {<Env> <Id>} → <Expr>
Adjoin   ::  {<Env> <Id>#<Expr>} → <Expr>
```

The IsMember function checks if a given identifier is present in the current environment, while Fetch will return the expression of the present identifier from the environment. If the identifier is not present, the original identifier is returned unchanged. Finally, the Adjoin function will add a new pair into the environment that overrides a previous mapping of the identifier, if it exists. Some examples are given below:

```
{IsMember [a#E1 b#y c#E3]  c}     % returns true
{IsMember [a#E1 b#y c#E3]  y}     % returns false
{Fetch [a#E1 b#y c#E3]  c}        % returns E3
{Fetch [a#E1 b#y c#E3]  d}        % returns d
{Adjoin [a#E1 b#y c#E3]  c#E4}    % [c#E4 a#E1 b#y]
{Adjoin [a#E1 b#y c#E3]  d#E4}    % [d#E4 a#E1 b#y c#E3]
```

Renaming

The bound variables of lam/let constructs do not carry any meaning by themselves and are essentially place holders to indicate binding of argument to corresponding identifiers. We can thus rename the bound identifiers without changing the meaning of a lambda term. For example, we have the following equivalences:

```
lam(z z) ≡  lam(y y)  ≡  lam(a a)
let(id#lam(z z) apply(id y)) ≡  let(a#lam(b b) apply(a y))
```

To carry out the renaming of bound identifiers, we must have the ability to generate unique identifiers. Let us use a function NewId that would generate a unique identifier of the form id<n>, as defined below:

```
Cnt={NewCell 0}
fun {NewId}
   Cnt:=@Cnt+1
   {String.toAtom {Append "id<" {Append {Int.toString @Cnt} ">"}}}
end
```

This function uses a Cell object to obtain a running number that is incremented with each invocation to NewId. Each {NewId} call would give unique identifier with each invocation. We expect NewId function to work as follows:

```
{NewId}    % returns id<1>
{NewId}    % returns id<2>
{NewId}    % returns id<3>
```

With the help of NewId, you are to define a function Rename that would return a new lambda term where the bound variables are uniquely renamed, as highlighted below.

```
{Rename lam(z lam(x z))}
     % returns lam(id<1> lam(id<2> id<1>))
{Rename let(id#lam(z z) apply(id y))}
     % returns let(id<3>#lam(id<4> id<4>) apply(id<3> y))
```

Substitution

Lambda terms are essentially evaluated with the help of substitution. We can implement such substitution with the following function:

```
Subs :: {<Id>#<Expr> <Expr>} → <Expr>
```

Using this function, we may reduce each application by substituting the argument for its parameter in the function's body. For example, the lambda term

```
apply(lam(x apply(x y)) lam(x x))
```

can be reduced by the following substitution process.

```
= {Subs x#lam(x x) apply(x y) }
= apply(lam(x x) y)
```

When applying substitution, we should always substitute only the free occurrences of its identifier in the main expression. We must ensure that the bound identifiers of the same name are not substituted. For example, the third occurrence of `x` below is not substituted as it is bound to an inner lambda term:

```
{Subs x#lam(z z) apply(x lam(x apply(x z))) }
= apply(lam(z z) lam(x apply(x z)))
```

Another subtlety with substitution is that the free variables of the argument to substitute into a given expression *must not clash* with the bound variables of the latter. For example, the substitution below has such a clash as the free variable z of the argument `lam(y z)` clashes with the bound identifier of `lam(z apply(x z))`.

```
{Subs x#lam(y z) apply(x lam(z apply(x z))) }
```

If we apply the substitution in a naïve way, we will get the following result:

```
apply((lam y z) lam(z apply((lam y z) z)))
```

where the third occurrence of `z` is now bound when it should have been free.

One solution to this problem is to rename each lambda term whose bound variable clashes with the free variables of the argument being substituted. In the above example, we should rename the lambda term as follows:

```
{Rename lam(z apply(x z))}
= lam(id<1> apply(x id<1>))
```

After this renaming, we can apply substitution in a safe manner, as follows:

```
{Subs x#lam(y z) apply(x lam(id<1> apply(x id<1>)))}
= apply(lam(y z) lam(id<1> apply((lam y z) id<1>)))
```

Implement a `Subs` function which adheres to the above stated conditions.