Sample Problem 1

1. Implement an interpreter for the call-by-name untyped λ-calculus in either ML or OCaml. Your code should use the following datatype definitions:

```
datatype exp = Var of string
             | Lam of (string*exp)
             | App of (exp*exp)

exception UnboundVar
```

   For example, the λ-calculus expression $\lambda x.(xx)$ would be represented as the ML/OCaml value `Lam("x",App((Var "x"),(Var "x")))`. Your interpreter implementation should consist of two functions:

   (a) Implement a function `subst` such that `(subst "x" e2 e1)` yields the result of the capture-avoiding substitution $e_1[e_2/x]$. Your function should therefore have type

   $$\texttt{subst} : \texttt{string} \to \texttt{exp} \to \texttt{exp} \to \texttt{exp}$$

   (b) Recall that every λ-calculus expression e either terminates yielding an expression of the form $\lambda x.e_2$ or loops infinitely. Implement a function `eval` such that if λ-calculus expression e yields $\lambda x.e_2$ then `(eval e)` returns the pair `("x",e2)`, and if e loops infinitely then `(eval e)` loops infinitely. If you encounter an unbound variable while evaluating expression e, use `raise UnboundVar` to throw an exception and thereby halt evaluation. Your function should therefore have type

   $$\texttt{eval} : \texttt{exp} \to (\texttt{exp*exp})$$

Sample Solution

1. (a) OCaml implementation:

```
let rec subst x e2 e1 = (match e1 with
     Var y -> if y=x then e2 else e1
   | Lam (y,e) -> if y=x then e1 else Lam (y,subst x e2 e)
   | App (e3,e4) -> App (subst x e2 e3, subst x e2 e4));;
```

   ML implementation:

```
fun subst x e2 (Var y) = if y=x then e2 else e1
  | subst x e2 (Lam (y,e)) = if y=x then e1 else Lam (y,subst x e2 e)
  | subst x e2 (App (e3,e4)) = App (subst x e2 e3, subst x e2 e4);
```

(b) OCaml implementation:

```
let rec eval e = (match e with
     Var x -> raise UnboundVar
   | Lam (x,e) -> (x,e)
   | App (e1,e2) -> let (x,e)=(eval e1) in eval (subst x e2 e));;
```

ML implementation:

```
fun eval (Var x) = raise UnboundVar
  | eval (Lam (x,e)) = (x,e)
  | eval (App (e1,e2)) = let (x,e)=(eval e1) in eval (subst x e2 e);
```

Sample Problem 2: Denotational Semantics of a Relational Database Language. 50 pts

Consider an SQL like relational database programming language. Programs in this language manipulate a relational database. A relational database (RDB) stores a table (collection of tuples) for each relation name. Thus, each relation can be regarded as a collection of tuples. **(12 pts)** Define a semantic algebra for this database domain which supports the following operations Note: all definitions should be in proper lambda calculus format and should take care of error situations:

- *create*: creates an empty database.
- $retrieve(database, relname)$: given a relation name in a database it returns the table corresponding to that relation.
- $store(database, relname, table)$: given a relation name in a database and a table, it stores the table in the database under that name.

Assume that *table* has as its domain

$table = key \rightarrow onetuple$

and

$onetuple = key \times list$

where $key = \texttt{Nat}$ (set of natural numbers) and $list = \texttt{Nat}^*$ (list of natural numbers). Thus, a collection of tuples is modeled as a function. Each tuple is modeled as a pair, whose first element is the primary key and the second element is a list of remaining elements of the tuple. Note: DO NOT USE YOUR OWN DOMAIN DEFINITIONs for *table*, *onetuple*, *key*, *orlist*.

(13 pts) Extend the semantic algebra with the following operations:

- $create\text{-}rel(database, relname)$: given a database and a relation name, it initializes this relation to an empty collection of tuples in the database.
- $access(database, relname, key)$: given a database, a relation name, and a key, it returns the tuple in the relation that matches the key.
- $update(database, relname, newtuple)$: given a database, a relation name, and a tuple, it adds or updates the relation relname with tuple *newtuple* where *newtuple* is of type *onetuple* (note that the key can be extracted from the *newtuple*; if a tuple with that key is already present, it is replaced, if not, the tuple *newtuple* is added).

Now consider part of the Grammar for expressions in the database language:

$C \in Commands$

$E \in \mathit{Relational\text{-}Expression}$
$C ::= R[D] := E$
$E ::= E_1 \bowtie E_2 \;\mid\; E_1 - E_2 \;\mid\; \pi_n(E) \;\mid\; R(D)$

with the meanings:

1. In the productions aboe, $R[D]$ refers to the table corresponding to relation R in database D. A relational expression E returns a table (collection of tuples).
2. $R(D) := E$ computes result of relational expression E and stores under the relation name R in Database D.
3. $E_1 \bowtie E_2$ is the usual relational database join of two relational expressions E_1 and E_2. Two tuples (k, l_1) and (k, l_2) that have the same key and that have been joined will appear as $(k, \mathit{append}(l_1, l_2))$ in the resulting relation.
4. $E_1 - E_2$ produces a relation that is obtained by removing from relational expression E_1 the tuples of relational expression E_2 that are also present in E_1.
5. $\pi_n E$ takes a relational expression E as input and produces the relation consisting of 2 element tuples where the first element is the key in the tuples of relation returned by E and the second element is the nth element of the tuples of relation returned by E.

(25 pts) Give the valuation function of C and E. Note that some functions may be recursive.

Solution: Will be made available soon.