CEng 242 Homework 2

Due 30^{th} March 2003

In this homework you will simulate static binding applied in ML like program phrases.

You are given the following datatype definitions:

```
type ValDec = {ID:string, EXP: string, BODY:string list};
type Environment = {ID:string, EXP: string} list;

datatype Declaration= Simple of (bool*ValDec)
        | Collateral of (bool*ValDec list)
        | Local of (Declaration list)*Declaration;
```

A `ValDec` typed value defines a binding occurrence for an identifier. `ID` field stores the identifier name. `EXP` stores a description of the identifier. It works like a comment which will be used to distinguish identifiers with same name. `BODY` contains a list of strings which stands for a list of applied occurrences. Assume that this is like any ML expression with all constants, keywords, operators, punctuation are removed and only used identifier names are left.
For example ML declaration:

```
val x = if a<b then (f a)+(g b)  else (g (f a))+b+x;
```

can be represented by:

```
{ID="x",EXP="Some integer x",BODY=["a","b","f","g","x"]}
```

An `Environment` typed value defines a list of bindings. Each binding is an (`ID, EXP`) pair corresponding the same fields in the `ValDec`.

A `Declaration` typed value is like a declaration phrase in ML . It can be a simple `val` declaration, a collateral declaration or a local declaration block.

`Simple bool*ValDec` stands for a `val` declaration in ML . First boolean value is the recursive flag. If it is `true` then `val` definition is recursive (like `val rec f=...`), else it is not recursive.

```
Simple(true,{ID="f",EXP="f the factorial",BODY=["f","x"]})
```

represents the ML phrase:

```
val rec f=fn x => x*f (x-1)
```

`Collateral bool*(ValDec list)` stands for a list of simple declaration combined by the collateral composition (`and` in ML). First boolean value is again the recursive flag modifying all components.

```
Collateral(true,[{ID="f",EXP="some f",BODY=["g","x"]},
                 {ID="g",EXP="some g",BODY=["f","x"]}])
```

represents the ML phrase:

```
val rec f= .... (g x) ...
and     g= .....(f x) ...
```

`Local (Declaration list)*Declaration` represents a local declaration block where the second `Declaration` value is the body of the declaration and it is evaluated with the local `Declaration list` environment. After environment will not include this declaration list but the declaration in the second element of the tuple. It corresponds to the `local ... in ... end` declaration in ML .

```
Local ([LD1,LD2,LD3],D)
```

represents the ML phrase:

```
local LD1;LD2;LD3
in D
end
```

Write the following ML functions:

1. `environment: Declaration list -> Environment`
   `environment` ***`declarations`*** will get a list of declarations and return the current environment after these declarations. There should be no duplicate identifiers in the resulting list.

2. `bindings: Declaration list -> {ID:string,EXP:string} list`
   `bindings` ***`declarations`*** will get a list of declarations, for all applied occurrences in the `BODY` fields of the declarations find the `EXP` field in the corresponding binding occurrences and return a list of these bindings in a list {`ID:string, EXP:string`} typed values.

   Assume initial environment is empty. Give bindings in order of appearance in the declaration list. If the applied occurrence is free (there is no binding occurrence for it) give `EXP` as "FREE".

Example:

```
val d = Local ([ Simple (false,{ID="x", EXP="first x", BODY=[]}),
                 Collateral(true,[ {ID="f", EXP="mutual f", BODY=["x","f","g"]},
                                   {ID="g", EXP="mutual g", BODY=["x","f"]} ]),
                 Simple (true,{ID="x", EXP="second x", BODY=["x"]})
                ] ,
                 Collateral(false,[ {ID="a", EXP="the a", BODY=["x","g"]},
                                   {ID="b", EXP="the b", BODY=["a","f"]} ])
              );

environment [d];

[{ID="a", EXP="the a"}, {ID="b", EXP="the b"}]

bindings [d];

[{ID="x", EXP="first x"},{ID="f", EXP="mutual f"},{ID="g", EXP="mutual g"},
 {ID="x", EXP="first x"},{ID="f", EXP="mutual f"},
 {ID="x", EXP="second x"},
 {ID="x", EXP="second x"},{ID="g", EXP="mutual g"},
 {ID="a", EXP="FREE"},{ID="f", EXP="mutual f"}]
```