CEng 242 Homework 2

Due: 23^{rd} March 2001

In this homework you will be asked to write a simple implementation for structural type equivalence check. Assume you are given the following type definitions:

```
datatype Type = Unit | Prim of string | Cart of Type*Type
                        | Disj of Type*Type | Mapp of Type*Type
                        | Var of string ;

type Typedef = string*Type;

datatype  'a Result = Error of string | Ok of 'a;
datatype 'a Search = Found of 'a | NotFound;
```

Where the type `Type` will be used to represent a type, `Typedef` will be used to represent a type definition where the first element of the tuple is a string representing a type variable and second is the defined type. The type `'a Result` is a generic type for returning errors or actual values. `'a Search` is a generic type for returning a search result.

a) Write a function:
`declare:  Typedef list -> string -> Type -> (Typedef list) Result`
where `declare oldtypedefs typename typeval` will get a list of current type definitions a `typename` string representing the name of the new type and a `typeval` value which is the content of the type to be declared and return either `Error "`*`errorstr`*`"` or `Ok newtypedefs`, a list of typedefs where new definition is inserted to the beginning of `oldtypedefs`.

`declare` function should make the following checks with same order:

1. If type `typename` definition exists in the `oldtypedefs`. If exists return '`Error "Redefinition of `*`typename`*'
2. Every type variable occurrence in `typeval` by `Var `*`typestring`* either should be already defined in `oldtypedefs` or *`typestring`* should be equal to `typename`, the new type to be declared. Check should be done for all occurrences in `typeval` recursively in a depth first manner and only the first error case is returned by a '`Error "Undeclared type variable `*`typestring`*`"`'
3. Otherwise return `Ok `*`newtypedefs`*

b) Write a function:
`structequal:  Typedef list -> string -> string -> bool Result`
where `structequal typedefs type1 type2` will check if `type1` is structurally equivalent with `type2` and return either `Error "`*`errorstr`*`"` or `Ok true` if they are equal or `Ok false` if they are not.

Remember the structural equivalence rules are as follows:

- `Unit` $\equiv$ `Unit`
- `Prim` $s \equiv$ `Prim` $t \Leftrightarrow s = t$
- `Cart` $(T,U) \equiv$ `Cart` $(V,Y) \Leftrightarrow (T \equiv V \wedge U \equiv Y)$
- `Mapp` $(T,U) \equiv$ `Mapp` $(V,Y) \Leftrightarrow (T \equiv V \wedge U \equiv Y)$
- `Disj` $(T,U) \equiv$ `Disj` $(V,Y) \Leftrightarrow ((T \equiv V \wedge U \equiv Y) \vee (T \equiv Y \wedge U \equiv V))$

- `Var` $s \equiv$ `Var` $t \Leftrightarrow$
  $(s = t)$, or
  s and t are the variables tried to be checked for equivalence,i.e.
  (s =`type1` $\wedge$ t =`type2`) $\vee$ (s =`type2` $\wedge$ t =`type1`) , or
  type definitions s and t are expanded if the conditions described in the following paragraphs hold and if s is defined as type S and t is defined as type T, $S \equiv T$
- (`Var` $s \equiv T$) or ($T \equiv$`Var` s), T is not a type variable $\Leftrightarrow$
  type definition s is expanded if the conditions described in the following paragraphs hold and if s is defined as type S, $S \equiv T$
- Otherwise types are not equal.

Variable expansion mentioned above problematic since it introduces cyclic references and infinite recursion. So we will put a restriction to avoid cyclic references: A type variable can be expanded only once in a path. So you will use a list of expanded variable references for each type thread (for `type1` and `type2` separately, initial values will be `[]`) and expand the variable definition with the definition in the `typedefs` if the variable string is not included in the list of expanded references in the path. If it is already visited (expanded) check will simply return `Ok false` .

`structequal typedefs type1 type2` will check and return the following errors:

- if neither `type1` nor `type2` is defined in `typedefs`, return '`Error "Both` *`type1`* `and` *`type2`* `are undefined"`' (Note that *type1* and *type2* will be replaced by the string content of identifiers).
- if any of `type1`, `type2` or a variable required to be expanded in the recursive check process is not defined in `typedefs`, return '`Error "`*`typename`*` is undefined"`'

If no error occurred, either `Ok true` or `Ok false` is returned.

Examples:

```
val r=declare [] "t" (Disj(Unit,Cart (Prim "real",Prim "int")));
 val r = Ok [("t",Disj (#,#))] : (string * Type) list Result
val Ok current=r;
 val current = [("t",Disj (Unit,Cart #))] : (string * Type) list
val r=declare current "u" (Disj(Cart (Prim "real",Prim "int"),Unit));
 var r = Ok [("u",Disj (#,#)),("t",Disj (#,#))] : (string * Type) list Result
val Ok current=r;

structequal current "t" "u";
 val it = Ok true : bool Result

val r=declare current "v" (Var "t");
 val it = Ok [("t",Disj (#,#)),("u",Disj (#,#)),("v",Var "t")] ...
val Ok current=r;

structequal current "v" "u";
 val it = Ok true : bool Result... (v expands t and t=u)

structequal current "v" "t";
 val it = Ok true : bool Result... (v expands t and t=t)
structequal current "w" "t";
 val r = Error "w is undefined" : bool Result
```

```
val r=declare current "t" (Prim "real");
 val r = Error "Redefinition of t" : (string * Type) list Result

val r=declare current w  (Cart (Var "z",Prim "int"));
 val r = Error "Undeclared type variable z" : (string * Type) list Result

val r=declare [] "t" (Disj(Unit,Var "t"));
val Ok current=r;
val r=declare current "u" (Disj(Unit,Var "u"));
val Ok current=r;
val r=declare current "v" (Disj(Unit,Disj (Var "v",Unit)));
val Ok current=r;
val r=declare current "w" (Disj(Unit,Disj (Disj (Unit,Var "w"),Unit)));
val Ok current=r;

structequal current "t" "u";
 val it = Ok true : bool Result ... (t and u are checked so t=u)
structequal current "v" "t";
 val it = Ok true : bool Result ... (t is expanded once and t=v)
structequal current "w" "t";
 val it = Ok false : bool Result ... (t is not further expanded)
```

In your submissions you should not introduce extra top level declarations. Put and hide all other definitions in a `local ... in ... end;` block. Use the following pattern in your declarations:

```
datatype Type = Unit | Prim of string | Cart of Type*Type
                        | Disj of Type*Type | Mapp of Type*Type
                        | Var of string ;

datatype  'a Result = Error of string | Ok of 'a;
datatype 'a Search = Found of 'a | NotFound;
local
   (* all your local auxiliary functions and definitions here *)
in
  fun declare  (* definition of declare here *)  ;
  fun structequal  (* definition of structequal here *)  ;
end;
```

Further examples and test data will be available in a few weeks in the newsgroup.