Homework must be submitted by mail to cs558acc@cs.pdx.edu. All submitted files (two for this assignment) must be sent as plain-text attachments to the mail message (the contents of the message itself will be ignored). It is your responsibility to submit the homework in the proper format.

All programs mentioned can be downloaded from the course web page.

1. Recall the ML interpreters used in previous assignments, such as the one for language E51 in assignment 5. They make use of a structure Env that implements an abstract data type (ADT) for environments. The signature of this ADT is given between the keywords sig and end:

   ```ml
   structure Env :>
   sig
       type 'a env
       exception NotFound of string
       val empty : 'a env
       val extend : 'a env -> (string * 'a) -> 'a env
       val lookup : 'a env -> string -> 'a
   end =
   struct
       ...
   end
   ```

   Your task is to write a new structure implementation (replacing the ... ) that implements the same functionality as the existing Env, but uses (unbalanced) binary search trees rather than lists. (Consult your favorite algorithms text for more details about binary search trees if you need them.) Use the following definition of trees

   ```ml
   datatype 'a env = LEAF
                    | NODE of string * 'a * 'a env * 'a env
   ```

   Do not alter the existing Env interface and don’t accidentally alter the behavior of the operators. In particular, remember that if an environment is extended twice with the same identifier, the more recent extension “hides” the previous one.

   Put your new structure definition (only) into a file sol7.t.sml and submit it. Note: You may wish to test your solution by using it in the context of an interpreter, but don’t submit the interpreter code – just the structure definition. Actually, it is easier to write an independent test driver for the ADT.
2. Recall the simple typed language E6 from homework 6. We now extend that language by adding support for defining type synonyms, i.e., we provide a way to give names to type expressions. The syntax of this new “E7” language is the same as E6 except for the following modifications:

```
prog := '(' { def } ')' exp
def := fundef
    | typedef
fundef := '(' 'fun' fname typ '(' { var typ } ')' exp ')'
typdef := '(' 'type' tname typ ')
typ := tname
    | 'Int'
    | 'Bool'
    | '(' 'Pair' typ typ ')
tname := letter { letter | digit } (but excluding 'Int' and 'Bool')
```

Type synonym definitions can only refer to previously defined types (or built-in types); thus, in particular, they cannot be recursive. (Note that type synonym definitions and function definitions behave differently in this respect.)

An full interpreter for E7 is provided in `hw7_2.java`. Modify this interpreter to support a new language feature: binary disjoint sum (discriminated union) types. The syntactic extensions are as follows:

```
typ := ...
    | '(' 'Sum' typ typ ')
exp := ...
    | '(' 'inleft' typ exp ')
    | '(' 'inright' typ exp ')
    | '(' 'case' exp var exp var exp ')
```

The type \( \text{Sum } t_l \ l t_r \) is a sum type with two tags, identified as “left” and “right;” the value carried with the left tag is of type \( t_l \) and the value carried with the right tag is of type \( t_r \).

The injection expression \( \text{inleft } t e \) creates a value of type \( t \) (which must be a \text{Sum} type) tagged “left” and carrying the value of \( e \). Similarly, \( \text{inright } t e \) creates a value tagged “right.” (Note that is necessary to give an explicit type parameter to these expressions because the full sum type cannot be inferred from the type of \( e \). Usually this will be a named type.) The expression \( \text{case } e v_l e_l v_r e_r \) is evaluated as follows. First \( e \) is evaluated; the result must belong to a sum type. If the result is tagged “left” then the current environment is extended by binding the associated value to \( v_l \) and \( e_l \) is evaluated in the resulting environment, yielding the result of the entire case. Similarly, if the result is tagged “right” then the associated value is bound to \( v_r \) and \( e_r \) is evaluated to yield the case result.

Also, \text{write} should work on \text{Sum} values; for example, the value corresponding to \( \text{inleft } (\text{Sum } \text{Int } \text{Bool}) 42 \) should display as “(L 42)” and similarly for \text{inright}. 

2
Here are the typing rules for the new expressions:

\[
\begin{align*}
\frac{TE \vdash e : t_l \quad t = (\text{Sum } t_l t_r)}{TE \vdash (\text{inleft } t e) : t} & \quad \text{inleft} \\
\frac{TE \vdash e : t_r \quad t = (\text{Sum } t_l t_r)}{TE \vdash (\text{inright } t e) : t} & \quad \text{inright} \\
\frac{TE \vdash e : (\text{Sum } t_l t_r) \quad TE + \{v_l \mapsto t_l\} \vdash e_l : t \quad TE + \{v_r \mapsto t_r\} \vdash e_r : t}{TE \vdash (\text{case } e v_l e_l v_r e_r) : t} & \quad \text{case}
\end{align*}
\]

To give more intuition about these expressions, suppose we had an ML type definition

```
datatype t = Left of int | Right of bool
```

This corresponds to an E7 type declaration

```
(type t (Sum Int Bool))
```

The expressions \((\text{inleft } t e_1)\) and \((\text{inright } t e_2)\) play the roles of the ML constructor applications \text{Left } e_1\) and \text{Right } e_2,\) and \((\text{case } e v_l e_l v_r e_r)\) is similar to the ML

```
case e of
  Left v_l => e_l
| Right v_r => e_r
```

In fact, we can view the ML \texttt{datatype} mechanism as an extended version of E7's sums, allowing arbitrary numbers of summands (instead of just two) and arbitrary constructor names (instead of the fixed names \texttt{left} and \texttt{right}).

The necessary parsing support for this extension is already present in \texttt{hw7.2.java}; you just need to uncomment it. You'll need to add a new subclass \texttt{SumTyp} of \texttt{Typ}, a new subclass \texttt{SumValue} of \texttt{Value} (and perhaps a new function \texttt{sumValue}) and the AST, printing, and evaluation code for the new expression forms. Be careful to maintain the invariants that only resolved types are compared using \texttt{equals}, stored into the type name environment, or returned by \texttt{check}.

Put your modified interpreter into a file called \texttt{sol7.2.java} and submit it.