Homework 2

Please submit this work by Wednesday, April 16 at 11:59pm.

This week, we give you further practice with Standard ML programming, and we come closer to the main topic of the course. More specifically, these problems deal with the *implementation* of programming languages, which is, once again, while not being the main thrust of the course, motivate why we are interested in scientific programming language design in the first place.

Submit your work by committing updated versions of the seed code files in `hw2/L` and `hw2/P`.

Problem 1: The Programming Language L

This problem entails the implementation of a toy language L. Terms in the language L are as follows:

```
t ::= n            // nonnegative integer constants n, one or more digits
    | x            // variable names, one or more lowercase letters
    | +(t,t)       // addition
    | LET(x<-t,t)  // binding: "let x be t in t"
```

The concrete syntax of this language is designed first and foremost to be easy to parse.

The implementation framework for this language is in your subversion repository in the directory `hw2/L`. We have provided a set of datatypes and various trivial operations, leaving the interesting bits to you.

By inspecting the code, and specifically by looking at what remains to be done, you can infer what you have to do for this assignment. There are four main phases that need to be implemented, each of which lives in its own module:

- lexical scanning, whereby a string of source code is split into a sequence of tokens,
- parsing, wherein a sequence of tokens is composed into an abstract syntax tree,
- scoping, whereby each variable is assigned a unique integer stamp, and
- evaluation, where the whole program is evaluated to whatever its integer value is.

The first three of these phases each has its own kind of error: syntax errors, parse errors, and unbound variable errors, respectively. Some examples:

- `"+<9,9>"` doesn't scan; there is a syntax error because angle bracket tokens are not in the language.
- `"9+9"` scans but doesn't parse; there is a parse error because infix plus is not part of the syntax design.
- `"LET(x<-8,y)"` scans and parses, but contains an unbound variable `y`.

If a program makes it through the first three phases of the interpreter, there should never be an error in the evaluation phase.*

* *This is a sweeping claim, and admittedly, there may be some pathologies involved with 1000-digit*

numbers or million-letter variable names that are not addressed in this rather informal language semantics! If you don't set out to sabotage the interpreter, you should be OK.

With respect to lexical scanning: your scanner should tolerate any number of spaces (whitespace characters like space, tab and newline) between tokens (but not in the middle of tokens). That is to say, the program`"LET( x <- 8 ,x ) "` is legal, while`"L E T ( x < - 9, x)"` is not. You can use the Standard ML basis library function`Char.isSpace : char -> bool` to detect such spaces. More broadly, for documentation on various useful operations on characters and strings, look here andhere.

In order to parse a sequence of tokens, you may use a simple *recursive descent* strategy. That is, once you've identified which kind of expression you are looking at in a token sequence, you can recursively parse the subexpressions and compose them together into the corresponding AST.

Problem 2: Pattern Match Checking

The second problem concerns pattern match checking. Consider a *match* to be a sequence of patterns, as in the left hand sides of the branches of a`case` statement. Since we are only concerned with pattern match checking for this problem, we will work with *lists of patterns* rather than full-blown case expressions; for the purposes of checking, that suffices.

In the seed code, a representation of a certain small set of datatypes is given, corresponding to the following running SML example:

```
datatype apple  = Fuji | Honeycrisp | GrannySmith
datatype citrus = Lemon | Lime | Orange
datatype fruit  = Apple of apple | Citrus of citrus
```

According to the `pat` type that you'll find in your seed code, a pattern is either a variable, a wildcard, a data constructor pattern (like `Fuji`, `Citrus c`,`Citrus _` or `Citrus Lime`), or a tuple of patterns:

```
datatype pat
  = Var of string * Ty.ty
  | Wild of Ty.ty
  | Con of Con.con * (pat option)
  | Tuple of pat list
```

Variables and wildcards carry their types with them, so determining the type of a variable or wildcard is immediate. The type of a constructor pattern can be determined by typing the constructor (see the function `Con.typeOf`, provided in the seed code), and the type of a tuple pattern is just the tuple type of all its subpatterns.

The result of checking a match (a pattern list) is embodied in the following datatype:

```
datatype match_check
  = DupVarName of string
  | TypeError of T.ty list
  | Redundant of P.pat
  | Inexhaustive
  | OK
```

This datatype names the various conditions under which a match check fails, as well as a successful check (`OK`). The properties to be checked are as follows:

- no individual pattern can include the same variable name more than once (`DupVarName`),

- all patterns in a match must be of the same type (`TypeError`),
- no pattern can completely overlap any preceding pattern (`Redundant`), and
- the match may not exclude any values in the type (`Inexhaustive`).

These examples illustrate each of the errors above in turn:

- the tuple patterns `(x,x)`, `(_,x,x)` and`(Apple a, Citrus a)` all contain duplicate variable names, so they are all illegal,
- the match `Fuji => ... | Lime => ...` is illegal, because the patterns do not share the same type,
- the match `x => ... | y => ...` is redundant, because there is nothing to match `y` that will not have matched `x`; the pattern`Fuji => ... | Fuji => ...` is also redundant, as is`Fuji => ... | Honeycrisp => ... | Fuji => ...`, and
- the match `Lime => ... | Orange => ...` is inexhaustive, since none of its patterns can match `Lemon`.

There are a few key operations left unimplemented in the match-check seed code; inspect the code as given and you will find them. In developing an approach to this problem, think about Standard ML patterns, and experiment with various matches in the SML REPL.