Consider a typed variant of our familiar simple language with imperative expressions, functions, and pairs, which we'll call “E6.” Its “concretized” abstract syntax is given by the following grammar:

\[
\begin{align*}
\text{prog} & ::= ( \{ \text{fundef} \} ) \text{ exp} \\
\text{fundef} & ::= ( 'fun' \text{ fname typ} ( ( \{ \text{var typ} \} ) \text{ exp} ) ) \\
\text{typ} & ::= '\text{Int}' \\
& \quad | '\text{Bool}' \\
& \quad | ( 'Pair' \text{ typ typ} ) \\
\text{exp} & ::= \text{var} \\
& \quad | \text{int} \\
& \quad | ( ':' \text{ var exp} ) \\
& \quad | ( 'while' \text{ exp exp} ) \\
& \quad | ( 'if' \text{ exp exp exp} ) \\
& \quad | ( 'write' \text{ exp} ) \\
& \quad | ( 'local' ( ( \{ \text{var exp} \} ) \text{ exp} ) ) \\
& \quad | ( 'block' \{ \text{exp} \} ) \\
& \quad | ( '@' \text{fname} \{ \text{exp} \} ) \\
& \quad | ( '+' \text{ exp exp} ) \\
& \quad | ( '-' \text{ exp exp} ) \\
& \quad | ( '*' \text{ exp exp} ) \\
& \quad | ( '/' \text{ exp exp} ) \\
& \quad | ( '<=' \text{ exp exp} ) \\
& \quad | ( 'pair' \text{ exp exp} ) \\
& \quad | ( 'fst' \text{ exp} ) \\
& \quad | ( 'snd' \text{ exp} ) \\
\text{fname} & ::= \text{letter} \{ \text{letter | digit} \} \\
\text{var} & ::= \text{letter} \{ \text{letter | digit} \}
\end{align*}
\]

The semantics of E6 expressions and functions are similar to those of previous homeworks. All scoping is static. The scope of each function name is the entire program, allowing two or more functions to be mutually recursive. The only variables are function parameters and locals. There are no user-defined globals, but the variable names true and false are predefined to the corresponding Boolean constants.

The language obeys a type discipline, distinguishing integers, booleans, and pairs; every variable and expression must belong to a unique type. Each function parameter is explicitly typed, as is the function result. For example, the code
(fun f Int (b Bool p (Pair Int Int))
  (if b (+ (fst p) (snd p))
   0))

defines a function f with return type Int, and two parameters: b, whose type is Bool, and p, whose type is a Pair of Ints. Local variables do not have to be explicitly typed, as their types can always be inferred from their initializing expressions.

It is a typing error to use an undefined function or variable name, or to define the same function name twice. If the same variable name appears twice in a parameter list or local declaration, the second appearance hides the first.

An E6 interpreter in Python (only) has been provided (hw6.py). It reads a file containing an E6 program in the syntax described above, echoes the program (to confirm correct parsing), performs some typechecking on it, executes it (possibly producing output from write expressions), and displays the overall result.

The typechecker catches some typing errors. For example, the program

(  
  (fun not Bool (y Bool) (if y false true))
  (fun f Int (x Int) (+ 2 3))
)
( @not (@ f 4))

causes a typechecking error, because f’s result type (i.e., Int) doesn’t match the argument type of not (i.e., Bool).

However, for many programs, the output of the typechecker will be the message:

   Typechecking failed: unimplemented

indicating that the necessary typechecking code is missing. Your task is to complete the typechecker, by doing proper typechecking at the seven places where the unimplemented message is currently generated. Use the existing code as a model. Your completed interpreter should be able to find all type errors in E6 programs. If you’ve done it properly, the interpreter should never fail during evaluation with a checked runtime error of any kind.

Put your modified interpreter into a file called sol6.py and submit it.