1. Consider a variant of our familiar simple language with imperative expressions and functions, and with an exception mechanism. We’ll call this language “E51.”

```
prog := '(' { fundef } ')' exp
fundef := '(' 'fun' fname '(' { var } ')' [ '(' { var exp } ')' ] exp ')' exp := var
| int
| '(' '==' var exp ')
| '(' 'while' exp exp ')
| '(' 'if' exp exp exp ')
| '(' 'write' exp ')
| '(' 'block' { exp } ')
| '(' 'throw' ')
| '(' 'catch' exp exp ')
| '(' '@' fname { exp } ')
| '(' '+' exp exp ')
| '(' '-' exp exp ')
| '(' '*' exp exp ')
| '(' '/' exp exp ')
| '(' '!=' exp exp ')
fname := letter { letter | digit }
var := letter { letter | digit }
```

The semantics of E51 are very similar to those of E4 (including the possibility of local variables in function definitions). In addition, there are mechanisms for throwing (raising) and catching (handling) exceptions. The throw expression throws an exception, which causes control to pass immediately to the nearest dynamically enclosing handler. No exception value is passed. Handlers are introduced by expressions (catch e₁ e₂). This means to evaluate e₁ with the handler e₂ active. If no exception is thrown during the evaluation of e₁, then the value of the whole expression is just that of e₁. If an exception is thrown during evaluation of e₁, and not is not caught within some nested handler in e₁, it will be caught by this handler: e₂ will be evaluated, and its value will be used as the value of the whole catch expression. Note that e₂ may itself throw an exception, which may be caught by an enclosing handler (but not by this one). For example, the bizarre program
(fun f (x) (+ 1 (throw)))
(fun g (y) (+ y (catch (Ơ f y) (+ 2 (throw)))))
(+ 4 (catch (Ơ g 6) 99))

evaluates to 103.

An E51 interpreter in SML (only) has been provided (hw5_1.sml). It reads a file containing
an E51 program in the syntax described above, echoes the program (to confirm correct parsing),
translates it into stack machine code, prints out the stack machine code (for debugging
purposes), executes it (possibly producing output from write expressions), and displays the
overall result. As usual, for more debugging help, you can trace the behavior of the stack
machine by executing the top-level command Machine.traceOn := true; before executing
Interp.interp.

Internally, the stack machine has been modified to maintain a stack of currently active
exception handlers, and new operations have been added to manipulate this stack.

Your task is to modify this interpreter so that exceptions pass a value back to the enclosing
handler. Handlers can use the value to distinguish between different kinds of exceptions, or
as input to calculations. The modified syntax for raising and handling is given by:

exp := ...
| '(' 'throw' exp ')
| '(' 'catch' exp var exp ')

The new parameter to throw defines the value to be passed back. The new var parameter to
catch gives the name of the variable (which must already be in scope) to receive the value
within the handler.

For example, the strange program

(
  (fun f (x) (if (<= x 9) (throw x) (+ x 2)))
  (fun g () (y 0 z 0)
    (catch (catch (Ơ f 4) y (throw (+ y 3)))
      z (+ z 10)))
)
(Ơg)

should evaluate to 17.

The parsing support for this modification is already present; you just need to uncomment it.
Otherwise, you’ll need to change the AST, printing, and compilation for catch and throw.
Also, unlike in previous assignments, you will need to change the implementation of the stack
machine slightly. (There are several possible approaches that will work.)

Put your modified interpreter into a file called sol5_1.sml and submit it.
2. Consider a *typed* variant of our familiar simple language with imperative expressions, functions, and pairs, which we'll call “E52.” Its “concretized” abstract syntax is given by the following grammar:

```
prog := '(' { fundef } ')' exp
fundef := '(' 'fun' fname typ '(' { var typ } ')' exp ')'
typ := 'Int'
| 'Bool'
| '(' 'Pair' typ typ ')
exp := var
  | int
  | '(' '=' var exp ')
  | '(' 'while' exp exp ')
  | '(' 'if' exp exp exp ')
  | '(' 'write' exp ')
  | '(' 'local' '(' { var exp } ')' exp ')
  | '(' 'block' { exp } ')
  | '(' '@' fname { exp } ')
  | '(' '+' exp exp ')
  | '(' '-' exp exp ')
  | '(' '*' exp exp ')
  | '(' '/' exp exp ')
  | '(' '<=' exp exp ')
  | '(' 'pair' exp exp ')
  | '(' 'fst' exp ')
  | '(' 'snd' exp ')
fname := letter { letter | digit }
var := letter { letter | digit }
```

The semantics of E52 expressions and functions are similar to those of E4 (without local variables in function declarations), with the addition of pairs and local expressions similar to those of E3. 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 `Int`s. 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 E52 interpreter in Java (only) has been provided (hw5.2.java). It reads a file containing an E52 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 six 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 E52 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 sol5.2.java and submit it.