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.”

\[
\begin{align*}
\text{prog} &:= \left( \{ \text{fundef} \} \right) \text{exp} \\
\text{fundef} &:= \left( \text{'fun'} \ \text{fname} \left( \{ \text{var} \} \right) \right) \left[ \left( \text{exp} \right) \right] \text{exp} \\
\text{exp} &:= \text{var} \\
&\quad | \text{int} \\
&\quad | \text{int} \left( \text{exp} \right) \\
&\quad | \left( \text{'while'} \ \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{'if'} \ \text{exp} \ \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{'write'} \ \text{exp} \right) \\
&\quad | \left( \text{'block'} \ \text{exp} \right) \\
&\quad | \left( \text{'throw'} \right) \\
&\quad | \left( \text{'catch'} \ \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{'@'} \ \text{fname} \ \text{exp} \right) \\
&\quad | \left( \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{exp} \ \text{exp} \right) \\
&\quad | \left( \text{exp} \ \text{exp} \right) \\
\text{fname} &:= \text{letter} \ \{ \text{letter} \ | \ \text{digit} \} \\
\text{var} &:= \text{letter} \ \{ \text{letter} \ | \ \text{digit} \}
\end{align*}
\]

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 \text{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 (\text{catch} e_1 e_2). This means to evaluate $e_1$ with the handler $e_2$ active. If no exception is thrown during the evaluation of $e_1$, then the value of the whole expression is just that of $e_1$. If an exception is thrown during evaluation of $e_1$, and not is not caught within some nested handler in $e_1$, it will be caught by this handler: $e_2$ will be evaluated, and its value will be used as the value of the whole catch expression. Note that $e_2$ may itself throw an exception, which may be caught by an enclosing handler (but not by this one).
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:

```plaintext
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

```plaintext
( (fun f (x) (+ 1 (throw)))  
  (fun g (y) (+ y (catch (@ f y) (+ 2 (throw)))))  
)  
(+ 4 (catch (@ g 6) 99))
```

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 the following functional language, which we’ll call “E52.”

\[
\begin{align*}
\text{prog} & := \text{exp} \\
\text{exp} & := \text{var} \\
& \quad | \text{int} \\
& \quad | ('if' \ \text{exp} \ \text{exp} \ 'exp') \\
& \quad | ('let' \ \text{var} \ \text{exp} \ 'exp') \\
& \quad | ('letfun' \ \text{var} \ \text{var} \ \text{exp} \ \text{exp}) \\
& \quad | ('@' \ \text{exp} \ 'exp') \\
& \quad | ('+' \ \text{exp} \ 'exp') \\
& \quad | ('-' \ \text{exp} \ 'exp') \\
& \quad | ('*' \ \text{exp} \ 'exp') \\
& \quad | ('/' \ \text{exp} \ 'exp') \\
& \quad | ('<=$ \ \text{exp} \ 'exp') \\
& \quad | ('pair' \ \text{exp} \ 'exp') \\
& \quad | ('fst' \ \text{exp}) \\
& \quad | ('snd' \ \text{exp}) \\
& \quad | ('ispair' \ \text{exp})
\end{align*}
\]

As usual, comments may be included by enclosing them between \‘{\‘ and \’} characters, and they may be nested.

E52 is similar to some of our earlier languages, but lacks imperative features including assignment, \texttt{while}, \texttt{block}, and \texttt{write}. The local expression has been renamed \texttt{let} (to emphasize that it acts like an ML-style immutable binding), and only one variable can be defined. (Nested \texttt{lets} can be used to define multiple variables.)

The most important changes are in the treatment of functions, which are now defined using a locally-scoped \texttt{letfun} expression form. Functions are treated as just another kind of value, and they share the same name space as other values. To evaluate \(\texttt{(letfun } f \ x \ b \ e\) \), first create a function value whose formal argument is \(x\), whose body is \(b\), and whose environment is the current environment; then bind \(f\) to that function value and evaluate \(e\) in the resulting environment. Applications now take an arbitrary expression in the function position; this must evaluate to a function value. All functions take exactly one argument; pairs can be used to encode multiple arguments (or multiple results). Functions are completely “first-class,” i.e., they can be passed as arguments to, or returned as results of, other functions, and can be stored in pairs. Since there is now no need for a separate \texttt{(fun ...)} declaration form, a program is once again just an expression.

The web site gives several example E52 programs. Program \texttt{static.e52} illustrates that nested functions use static scoping rules; program \texttt{compose.e52} shows how to write a higher-order function that composes two existing functions.

An E52 interpreter in Java (only) has been provided (\texttt{hw5_2.java}). Note that the (single) environment now maps all identifiers directly to (immutable) values; since values can’t mutate, there’s no need for locations. Your task is to modify this interpreter to allow E52 functions to be \textit{recursive}. For example, program \texttt{map.e52} (which currently fails with an
unbound variable error) should work properly when you’re done. This requires only a very small change. But be careful: programs like static.e52 must also continue to work when you’re done!

Call your revised interpreter sol5_2.java and submit it.