Control Flow

\[ (+ (f \, 2 \, 3) \, (- \, (f \, 4 \, (* \, 5 \, 6)) \, 7)) \]

Calls: always more to do ... (until the end)

(f 2 3) then (* 5 6) then (f 4 30) then (- whatever 7) then ...

Returns: What next? There’s always somebody waiting ...

e.g. waiting for \( (f \, 4 \, (* \, 5 \, 6)) \), we have \( (\lambda \, (x) \, (+ \, (f \, 2 \, 3) \, (- \, x \, 7))) \) \( ((\lambda \, (x) \, (+ \, (f \, 2 \, 3) \, (- \, x \, 7))) \, (f \, 4 \, (* \, 5 \, 6))) \)

Defn: what-to-do-next after the call \( (f \, 4 \, (* \, 5 \, 6)) \) is its \textit{continuation}

Scheme provides access to continuations!
Exceptions in Scheme

Recall exceptions in Java, ML: Transfer control to nearest *dynamically scoped* exception handler (i.e., nearest on “call stack”).

Transfer control: Forget what you’re doing. Result of entire program is now result of the handle (catch) in the “call stack” that existed when the handler was reached.

Scheme has a *more powerful* concept that can be a little less convenient for exceptions:

- You explicitly indicate what “handler” (*continuation*) to transfer control to.
- You do the transfer via a function application (that does not have function-application semantics)
- The continuation does not even have to be on the “call stack” when it’s transferred to!
Continuations for exceptions

Plan:

- Using continuations for exceptions (More details later, time permitting)

Syntax:

- \((\text{let/cc } k \ e_1) : \text{in } e_1\), bind \(k\) to “current continuation” (basically, the point immediately after the \text{let/cc}) then eval \(e_1\)

- \((k \ e_2)\): “invoke” continuation bound to \(k\), passing value \(e_2\), in lieu of the value of \(e_1\) (now aborted)

Exception idiom:

- Instead of handler, use \text{let/cc}

- Pass an appropriate function that invokes \(k\) to any function that needs to “raise”