CSE 341: Programming Languages

Spring 2006

Lecture 21 — Exceptions & Continuations

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 (**λ** (x) (+ (f 2 3) (- x 7)))

((**λ** (x) (+ (f 2 3) (- x 7)))) (f 4 (* 5 6))

Defn: what-to-do-next after the call (f 4 (* 5 6)) is its *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:

- (let/cc k $\boldsymbol{e1}$) : in $\boldsymbol{e1}$, bind `k` to "current continuation" (basically, the point immediately after the `let/cc`) then eval $\boldsymbol{e1}$
- (k $\boldsymbol{e2}$): "invoke" continuation bound to `k`, passing value $\boldsymbol{e2}$, in lieu of the value of $\boldsymbol{e1}$ (now aborted)

Exception idiom:

- Instead of handler, use `let/cc`
- Pass an appropriate function that invokes `k` to any function that needs to "raise"