Lab: Improved exception handling
CSC 312, “Programming language implementation”
Department of Computer Science · Grinnell College
October 16, 2015

Exercises

A. Create a subdirectory called EXCEPTIONS-LAB in the directory you’re using for labs in this course and copy my implementation of the EXCEPTIONS programming language into it. The directory /home/stone/courses/languages/code/EXCEPTIONS contain the six language-specific modules that you need.

• expvals-and-environments.scm
• tokens.scm
• syntax-trees.scm
• scanner.scm
• parser.scm
• interpreter.scm

If you don’t already have the four utility modules test.scm, natural-numbers.scm, list-of.scm, and character-sources.scm, you’ll also need to copy them from the directory /home/stone/courses/languages/code into the parent directory of EXCEPTIONS-LAB.

B. EXCEPTIONS is an extension of the LETREC language. In order to provide sufficiently interesting examples, EXCEPTIONS has lists as a built-in data type and built-in expression variants for cons, car, cdr, null?, emptylist, and variable-arity list.

The features that deal with exceptions are located in two additional variants:

<expression> ::= try <expression> catch ( <identifier> ) <expression>
| raise <expression>

Evaluating a try-expression saves, but does not evaluate, the expression in the catch-clause and then evaluates the first subexpression. If an exception is raised during that evaluation, the identifier is bound to the value provided by the exception and then the body of the catch-clause is evaluated, and the resulting value becomes the value of the entire try-expression. If no exception is raised, the value of the first subexpression is the value of the entire try-expression.

Evaluating a raise-expression causes the evaluation of its subexpression, then recovers the handler information from the “nearest” try-expression — the one that the interpreter most recently started to evaluate but has not yet finished evaluating — binds the identifier to the value of its subexpression, and evaluates the body of the handler. Properly speaking, raise-expressions don’t have values of their own, because a continuation that is waiting for the value of a raise-expression will never receive it.

(1) In EXCEPTIONS, how would you write a try-expression with a handler that simply re-raises the exception that invokes it, with the same value?

(2) Which, if either, of the subexpressions of a try-expression is in tail position?

(3) Describe the algorithm that the continuation-passing interpreter for EXCEPTIONS uses, when an exception is raised, to find and apply the handler.

(4) Would it be possible to implement exceptions in an interpreter that did not use continuation passing? How?
C. Revise the EXCEPTIONS-LAB interpreter so that every variant of the continuation type includes a copy of the most recently encountered try-cont, so that a raise-expression can just use that copy instead of having to rely on apply-handler to carry out the linear search through the saved continuations.

D. Add to the EXCEPTIONS-LAB interpreter a stop-expression with the syntax

\[
<expression> ::= \text{stop} <expression>
\]

that evaluates its subexpression and then immediately terminates the program, providing the value of the subexpression as the program’s final answer.