Today’s pair-programming partnerships are:

- Ms. Liddell and Ms. Byrne
- Mr. Dewey and Ms. Kyaruzi
- Mr. Nordin and Mr. Mataire
- Ms. DeWitt and Mr. Champagne
- Mr. Schlager and Mr. Boateng Asante
- Ms. Jervis and Mr. Mulhall
- Mr. Zahid

In each pair, the partner listed first should log in and manage the keyboard and screen.

Exercises

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

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

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

B. CONTINUATIONS-LAB is an extension of the LETREC language described in section 3.4 of the textbook, with the addition of three new kinds of expressions, with the following grammar:

<expression> ::= not <expression>
| and <expression> <expression>
| or <expression> <expression>

Note that in this grammar `not`, `and`, and `or` are all keywords, not procedure names, and `and` and `or`, unlike their Scheme analogues, are strictly binary: Each must always have exactly two subexpressions, although you can nest `and`- or `or`-expressions to get the effect of continued conjunction or disjunction, as in

```
or zero?(elmo)
  or zero?(-(elmo, 1))
    or zero?(-(elmo, 2))
      zero?(-(elmo, 3))
```

To evaluate a `not`-expression, first evaluate its subexpression. It is an error if the value of the subexpression is not a `BoolVal`, that is, an expressed value of the variant constructed by `bool-val`. The value of the entire `not`-expression is the other `BoolVal`,
containing whichever Scheme boolean is not the one contained in the subexpression’s expressed value.

To evaluate an **and**-expression, first evaluate its first subexpression (the “conjuge nd” subexpression). It is an error if the value of this subexpression is not a **BoolVal**. If the value of the subexpression is the **BoolVal** that contains **#f**, this value is the value of the entire **and**-expression. On the other hand, if the value of the conjugend is the **BoolVal** that contains **#t**, then (and only then) should the interpreter evaluate the second subexpression (the “conjunct” subexpression). It is again an error if the value of this subexpression is not a **BoolVal**. If the value of the conjunct is a **BoolVal**, its value is the value of the entire **and**-expression.

The specification that the second subexpression in an **and**-expression is not evaluated if the first subexpression is false is called “short-cut evaluation” and is found in many programming languages, including Scheme, C, and Java.

(1) Formulate a similar short-cut evaluation rule for **or**-expressions.

(2) Which of the subexpressions and sub-subexpressions of the nested **or**-expression shown above will be evaluated under your short-cut rule evaluation rule in an environment in which **elmo** is bound to (num-val 1)?

**C.** (1) Which of the immediate subexpressions of **not**, **and**, and **or**-expressions build control context? Which are in tail position?

(2) How many new variants of the **continuation** data type will be needed in order to implement a continuation-passing interpreter for CONTINUATIONS-LAB?

**D.** All but one of the modules provided for the CONTINUATIONS-LAB language processor are complete and work correctly: The scanner recognizes **not** as a keyword token, the parser assembles appropriate syntax trees for **and**-expressions, and so on. The exception is **interpreter.scm**, which has three holes in it: one at line 51, where the new continuation variants will be defined; one at line 87, in the definition of **apply-cont**, where the continuations of these new forms will be applied to the values for which they are waiting; and one at line 135, in the definition of **value-of/k**, where the evaluation of the expressions of the three new kinds is performed. Fill in these holes, as follows:

(1) Hint: You will need one new kind of continuation to post-process the value of the subexpression in a **not**-expression. Start by writing the clause of **apply-cont** that will carry out that post-processing.

(2) Inspect the code to see what context information is available locally in **apply-cont** and what must be extracted from the continuation; this will tell you how many fields the continuation variant needs and what should be in them. Then you can write the appropriate clause of the datatype definition.

(3) Finally add the clause for **not**-expressions to the definition of **value-of/k**. It will include a call to the constructor for the new continuation variant.

(4) Repeat the preceding steps for any continuation variants that carry out post-processing on the value of the conjugend and/or conjunct subexpressions of an **and**-expression. Then repeat them again for **or**-expressions.