Scheme–>C notes for *An Introduction to Scheme*

Joel F. Bartlett

An Introduction to Scheme, by Jerry D. Smith, is a recent text on the programming language Scheme. Rather than being directed at a specific implementation of Scheme, it attempts to stick close to the dialect defined in the *Revised*^{4} *Report on the Algorithmic Language Scheme* that is the base for many implementations including Scheme–>C. This document provides section notes to point out the differences between TI PC Scheme used in the text and Scheme–>C. The user will also find it helpful to read *An Introduction to Scheme–>C in 19 Prompts* and have the *Scheme–>C Index to the Revised*^{4} *Report on the Algorithmic Language Scheme* and the *Revised*^{4} *Report on the Algorithmic Language Scheme* available for reference.

1.8 ((((((0))))))

Scheme–>C does not have an internal editor. Instead, users use the editor of their choice (which may or may not have parentheses matching) and then use **load** to load the file into the Scheme system.

1.10 PC Scheme and the Listener

This is what the interaction on page 9 looks like in Scheme–>C:

```
> (+ 3 2)
5
> (load "examples.sc")
SQUARE
"examples.sc"
> (square 2)
4
> (exit)
```

User input is prompted by ">", Scheme–>C files end with the suffix ".sc", and the command primitives **%c** and **%d** are not supported. User input is not evaluated until the user types return.

2.2 The Scheme Listener

Scheme–>C is started by the command **s2ci**. Once the command is executed, the window looks like this:

```
$ s2ci
Scheme->C -- 01sep91jfb -- Copyright 1989 Digital Equipment Corporation
>
```

When the evaluation of an expression results in an error, the debugger is entered. It prints an error message followed by a procedure call traceback. It then prompts the user with ">>" to allow commands to inspect the state of the computation where the error occurred. For now, simply type control-D to return to the main read-eval-print loop.

```
> (square 2)
***** SQUARE Top-level symbol is undefined
(SQUARE 2) in ENV-0
(EVAL ...)
(SCREP_REP ...)
(READ-EVAL-PRINT ...)
>> ^D
> (load "examples.sc")
SQUARE
"examples.sc"
> (square 2)
4
> (exit)
```

2.3 Simple Arithmetic

Most Scheme->C systems do not have bignums. Numbers are represented as either 29-bit integers or 64-bit floating point values.

Exercise for the student: Add bignums to Scheme->C.

The boolean constant for true is **#t** and false is **#f**. In keeping with tradition, both the empty list **()** and **#f** are considered to be false. It is good programming practice to not use the empty list **()** as a synonym for **#f**, and in IEEE compliant Scheme's your program won't work as the empty list **()** is a synonym for **#t**!

2.4.2 Constants

The constants **#!true**, **#!false**, and **#!null** are not implemented, use **#t**, **#f**, and **'()** respectively. The character constants **#\backspace**, **#\page**, and **#\rubout** are not implemented.

2.4.4 Literal Expressions

The value of **#f** is false, represented as **#f**, which is not the same as the empty list **()**. While both **#f** and **()** are considered to be false when evaluating a boolean expression, they are not equivalent. Note that the empty list is not a self-evaluating constant. In order to avoid an error, one must quote it when entering it into Scheme:

```
> ()
> (
***** EVAL Argument contains an item that is not self-evaluating: ()
(EVAL ...)
(SCREP_REP ...)
(READ-EVAL-PRINT ...)
>> ^D
```

```
> '()
()
>
```

3.2 The Global Environment

There is only one top level environment that contains both user and system definitions.

5.2 Logical Operators

Since **#f** and the empty list **()** are different, **#f** is always returned when a predicate returns false:

```
> (<= 4 3)
#F
```

5.7 begin

In the text, the programming convention is that one calls **newline** and then calls **display**. In printing to the terminal with Scheme->C, it is better to call **display** and then call **newline** as a newline is not automatically generated when Scheme prompts the user for additional input.

8.2 Characters

The following #*char-name* forms are supported: **#\formfeed**, **#\linefeed**, **#\newline**, **#\return**, **#\space**, and **#\tab**.

8.4.1 Character Predicates

The user need not define these as they are part of the system.

8.4.3 String Conversion Functions

See the Scheme->C documentation for information about **number->string** and **string->number**.

9.4 User-defined Port Operations

Ports in Scheme->C are represented as a pair of the symbol **port** and the procedure that implements it. For example:

```
> (current-input-port)
(PORT . #*PROCEDURE*)
>
```

Don't forget to put the call to **newline** after the second call to **display** when writing **addtwo**.

9.6 Strings as Ports

Strings can be opened as input ports by **open-input-string** and as output ports by **open-output-string**. See the index for more details.

9.7 A Utility for Reading Lines: read-ln

Users will not see either the backspace or rubout characters as they are handled by the workstation's terminal emulator.

10.2 Debugging and Lexical Scope

A pretty-print procedure is provided, **pp**, but it does not pretty-print the text of a procedure.

Breakpoints may be set on the entry and exit of any procedure defined in the top level environment. Rather than adding a call to **bkpt** as was done in the text, a user would set a breakpoint on each call to **<** and observe the value of **i**:

```
> (one-to-y-sqrd 3)

0 -calls  - (> 1 3)
0- i
1
0- ^D
0 -returns- #F
0- ^D

0 -calls  - (> 0 3)
0- i
0
0- ^D
0 -returns- #F
0- ^D

0 -calls  - (> -1 3)
0- i
-1
0- ^D
0 -returns- #F
0- ^D

0 -calls  - (> -2 3)
0- (top-level)
>
```

On each entry to **>**, the values of it's arguments are printed. The value of **i** can be examined by entering **i** followed by a return. When control-D or **(proceed)** is entered, the function is evaluated and the result is printed. To continue with the computation, enter control-D or **(proceed)**. Once it become clear that the program is in error, the user is able to return to the top level read-eval-print loop by entering **(top-level)**.

Tracing is done using the **trace** and **untrace** commands:

```
> (one-to-z-sqrd 3)
12
> (trace one-to-z-sqrd)
(ONE-TO-Z-SQRD)
> (one-to-z-sqrd 3)
(ONE-TO-Z-SQRD 3)
==> 12
12
> (untrace one-to-z-sqrd)
(ONE-TO-Z-SQRD)
>
```

The **bpt** command puts a breakpoint on both the entry and exit to a procedure. The **unbpt** command removes a breakpoint. Here's the example on page 141:

```
> (bpt sqr)
SQR
> (one-to-z-sqrd 3)
(ONE-TO-Z-SQRD 3)

1 -calls  - (SQR 1)
1- ^D
1 -returns- 2
1- ^D
1 -calls  - (SQR 2)
1- ^D
1 -returns- 4
1- ^D

1 -calls  - (SQR 3)
1- ^D
1 -returns- 6
1- (top-level)
> (unbpt sqr)
(SQR)
>
```

When both **sqr** and **one-to-z-sqrd** are traced, one gets the following output.

```
> (trace one-to-z-sqrd)
(ONE-TO-Z-SQRD)
> (trace sqr)
(SQR)
> (ONE-TO-Z-SQRD 3)
(ONE-TO-Z-SQRD 3)
  (SQR 1)
  ==> 2
  (SQR 2)
  ==> 4
  (SQR 3)
  ==> 6
==> 12
12
```

```
>
```

Tracing the recursive function **ftl** produces the following output.

```
> (trace ftl)
(FTL)
> (ftl 3)
(FTL 3)
  (FTL 2)
    (FTL 1)
      (FTL 0)
      ==> 1
    ==> 1
  ==> 2
==> 6
6
>
```

10.3 Debugging in a Lexically Scoped Environment

Conditional breakpoints can be placed on procedures using the **bpt** special form. The second argument is a test procedure that is either the name of a top level procedure or a lambda expression defining a procedure. On each entry to the procedure, the test procedure is evaluated with the arguments to the breakpointed procedure. When the test procedure returns a true value, the breakpoint is taken.

```
> (bpt one-to-n (lambda (x) (>= x 3)))
ONE-TO-N
> (mean-table 5)
=====================================
N   MEAN OF 1 TO N

1         1
2         1.5
3
0 -calls  - (ONE-TO-N 3)
0-
```

The first time the argument to **one-to-n** is **>=** 3, the breakpoint is taken. Once at a breakpoint, the **backtrace** procedure allows the call stack and environments to be inspected.

```
0- (backtrace)
(ONE-TO-N N) in ENV-0
(/ (ONE-TO-N N) N) in ENV-1
(DISPLAY (ONE-TO-N-MEAN N)) in ENV-2
(BEGIN (NEWLINE) (DISPLAY N) (DISPLAY "         ") (DIS ... in ENV-3
(EVAL ...)
(SCREP_REP ...)
(READ-EVAL-PRINT ...)
#F
0-
```

Environments are identified by the symbols **env-***i*. The value of an environment is an a-list of symbols and their values. It's often useful to use **pp** to print out an environment. An expression may be

evaluated within a specific environment by calling **eval** with two arguments, the expression and the environment.

```
0- env-1
((LOCATION . "inside one-to-n-mean") (N . 3))
0- env-3
((N . 3) (\d\o\l\o\o\p . #*PROCEDURE*) (PRINT-HEADER .#*PROCEDU
RE*) (HEADER-LINE . "======================================")(
HIGH-BOUND . 5))
0- (pp env-3)
((N . 3)
     (\d\o\l\o\o\p . #*PROCEDURE*)
     (PRINT-HEADER . #*PROCEDURE*)
     (HEADER-LINE . "======================================")
     (HIGH-BOUND . 5))#T
0- (eval 'high-bound env-3)
5
0- ^D
0 -returns- 6
0- ^D
2
4
0 -calls  - (ONE-TO-N 4)
0- ^D
0 -returns- 10
0- ^D
2.5
5
0 -calls  - (ONE-TO-N 5)
0- ^D
0 -returns- 15
0- ^D
3
======================================
#F
>
```

The same techniques that one uses to explore a program that has hit a breakpoint can be used to investigate a running program. Here a simple loop is run. When the user enters control-C the running program is interrupted. A breakpoint is put on **eq?** and then the program is continued by typing control-D. Once the breakpoint is hit, **(eq? 0 0)** is executed, and then **proceed** is used to change the result returned by **eq?**, which causes the loop to complete.

```
> (let loop ((i 0)) (if (eq? i 0) (loop i) 'done))
^C
***** INTERRUPT *****
(EQ? I 0) in ENV-0
(IF (EQ? I 0) (LOOP I) 'DONE) in ENV-1
(EVAL ...)
(SCREP_REP ...)
(READ-EVAL-PRINT ...)
>> (bpt eq?)
EQ?
```

```
>> ^D

0 -calls  - (EQ? 0 0)
0- ^D
0 -returns- #T
0- (proceed #f)
DONE
>
```

Finally, these techniques can be used to investigate the environment when an error occurs. Note that when control-D is entered to continue, Scheme returns to the top level read-eval-print loop.

```
> (let ((i 0)) (car (car (car i))))
***** CAR Argument not a PAIR: 0
(SCRT1_$_CAR-ERROR ...)
(CAR ...)
(CAR I) in ENV-0
(CAR (CAR I)) in ENV-1
(CAR (CAR (CAR I))) in ENV-2
(EVAL ...)
(SCREP_REP ...)
(READ-EVAL-PRINT ...)
>> i
0
>> env-0
((I . 0))
>> ^D
>
```

Exercise for the student: implement **assert**.

11.3 dir: A Utility for Listing Filenames (Implementation-specific)

In order to implement this in Scheme->C you'll need to implement your own version of **sort!** and use **(open-input-port "ls")** to generate a list of file names.

11.4 format: A Utility for Formatted Output

Scheme->C contains a procedure **format**. See the documentation for details.

13.2 Memory Organization

Exercise for the student: implement **append!**.

15.4 Macros [OPTIONAL]

Scheme->C macros implements "expansion passing" macros based upon the ideas found in *Expansion-Passing Style: Beyond Conventional Macros*, 1986 ACM Conference on Lisp and Functional Programming, 143–150.

The simplest form of a macro is a constant. The arguments to the special form **define-constant** are the symbol identifying the constant and the expression to evaluate to calculate it's value.

```
> (define-constant radius 23)
RADIUS
> (define-constant pi 3.14159)
PI
> (define-constant circumference (* pi radius 2))
CIRCUMFERENCE
> (define-constant area (* 3.14159 (* radius radius)))
AREA
> area
1661.90111
>
```

The second type of macro defines an in-line procedure. The form **define-in-line** associates a symbol with a procedure definition. All calls to the procedure are replaced by the lambda expression defining the procedure.

```
> (define-in-line (plus3 x) (+ x 3))
PLUS3
> (plus3 5)
8
>
```

The most general form of macro expansion allows the user to examine a procedure call and then selectively cause further macro expansion. The definition for **plus3** can also be written as:

```
> (define-macro plus3
     (lambda (form expander)
     (expander `(+ ,(cadr form) 3) expander)))
PLUS3
> (plus3 8)
11
>
```

The macro is defined by a procedure that takes two arguments: the form to be expanded, and a procedure to do further expansion. It's typical action is to build an expanded form and then call the further expansion procedure with the new form and the further expansion procedure as arguments. For examples of use of this type of macro expander, the reader is directed to the file **scrt/predef.sc** that defines the macros used by the compiler.

N.B. Macro definitions may not be placed inside procedure definitions.