Scheme: Debugging

Instructions on how to debug your Scheme programs can be found here.

`http://www.swiss.ai.mit.edu/projects/scheme/documentation/user_6.html`

I strongly recommend that you read the above page. What is contained below only covers some highlights.

There are several approaches to finding bugs in a Scheme program:

- Inspect the original Scheme program.
- Use the debugging tools to follow your program's progress.
- Edit the program to insert checks and breakpoints.

Only experience can teach how to debug programs, so be sure to experiment with all these approaches while doing your own debugging. Planning ahead is the best way to ward off bugs, but when bugs do appear, be prepared to attack them with all the tools available.

Scheme: Debugging (cont.)

1. Put **print statements** in your code. Don't forget to use "begin", to ensure the statements are evaluated and displayed in the right order.

```
(begin
  (display n)
  (display" is an integer. Marking recursive call with ")
   (display (car lst)))
```

Scheme: Debugging (cont.)

2. **Trace individual procedures** that you think may be problematic
   (trace <procedure-name>)

(Lots of examples of traces in our course notes!)

Scheme: Debugging (cont.)

3. Use the **debugger**

```
(debug)          ; to start the debugger
?                ; to find out what commands exist
A                ; prints arguments
I                ; prints the error statement
R                ; prints the execution history
E                ; enter a read-eval-print loop
R                ; print execution history of
                 ; current subproblem level
...
Q                ; to quit
```

Scheme: Debugging

4. Insert **Break points** into procedures

bkpt *datum argument ...*

Sets a breakpoint. When the breakpoint is encountered, datum and the arguments are typed (just as for error) and a read-eval-print loop is entered.

E.g.:

```
(define (factor n nDiv)
  (cond ((eq? n 2) (list 2))
        ((eq? n nDiv) (list nDiv))
        ((integer? (/ n nDiv))
          (begin (display "integer: ")
                 (bkpt n nDiv)
                 (cons nDiv
                       (factor (/ n nDiv) nDiv))))
        (else (factor n (+ nDiv 1)))
  )
 )
```

Scheme: Debugging (cont'd)

- **pp object [output-port [as-code?]]**
  - Prints the source code of a given procedure.
  - When debugging, you will have a procedure object but will not know exactly what procedure it is.

- **pa procedure**
  - Prints the arguments of a procedure

- **apropos string**
  - Search an environment for bound names containing string and print out the matching bound names.

Scheme: Debugging

5. Use the **edwin** environment

   On CDF, you can start edwin using the command

   ```
   scheme -edwin -edit.
   ```

   Type `<ctrl>-m` for some basic commands.

   This gives you a friendlier environment to test your procedures in, although it doesn't interact very well with *run*, so I wouldn't use it to test your interactive system.

Scheme: Programming Style

- **Use special suffix:**
  - "?" for predicates (i.e. functions returning #t or #f , e.g. member?)
  - "!" for any procedure with "side effects" (i.e. changes of bindings for non-local variables, e.g. set!)

- **Procedure definitions should be brief**
  - Oriented towards a single, well-defined task
  - Should be split into a number of subtasks if > 1 page

- **Comments:**
  - ; for comments on the same line with code
  - ;; for comments that run from beginning of line
  - ;;; for comments that describe the contents of the file (usually first in file)

- **Indentation:**
  - Indent procedure definitions like this, with the body starting a new line, and indented a few characters

```
(define (foo)
    15)
```

Scheme: Pgming Style (cont'd)

- Deeply nested cars and cdrs are often difficult to understand, and should therefore be avoided.
- Since Scheme is a dynamically typed language, the names of parameters should reflect their value.
- Most general guidelines on programming style also apply to Scheme programs.
  - Kernighan & Plauger (1974) andLedgard (1974) give good summaries of these.