MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Lab Notes

6.821 programming assignments use Scheme+, an extension of Scheme extended with a few new features — see the Scheme+ tutorial (appendix A of the Scheme supplement of the course notes).

This handout provides information about how to access and run Scheme+. There are two ways to run Scheme+:

a. Run Scheme+ on your office or personal machine. To do so requires having a version of MIT Scheme running on your machine. section 2 gives instructions for getting Scheme+ and MIT Scheme.

   This option provides you with the most control over your lab time and resources, but may require some extra work if MIT Scheme is not already installed on your system.

   Scheme+ may run on other Scheme implementations with some modifications, but porting the extended features may take significant time.

b. Use an Athena workstation to run Scheme+. Section 1 gives instructions on how to use Scheme+ on Athena.

   This option provides a pre-configured environment with MIT Scheme already set up, but will not allow you to be as flexible in doing your labs. It may be difficult to find an Athena workstation when you want one.

Once you have Scheme+ up and running, see section 3 for some usage notes.

1 Running Scheme+ on Athena

MIT Scheme is available on Athena in the `scheme` locker. To access Scheme, type

```
add scheme
```

at an `athena%` prompt. This will attach the `scheme` locker to your directory structure at `/mit/scheme` as well as put the MIT Scheme binaries in your path. At this point, running Scheme+ on Athena is more or less like running Scheme+ on your personal machine, described in the rest of this handout.

If you are using Athena for 6.821, you should also be aware of the `6.821` locker. You can access the `6.821` locker by typing

```
add 6.821
```

at an `athena%` prompt, which will attach the `6.821` locker to your directory structure at `/mit/6.821`. All of the code for the course, including Scheme+, will be in the `6.821` locker. Refer to the `README` file in the `6.821` locker for more information.

2 Running Scheme+ on Your Own Machine

To run Scheme+ on your own machine, you must have a version of MIT Scheme. If you're using a machine in Tech Square, there's a good chance there's already a version of MIT Scheme on your machine. If not, read the details below on how to load one up (section 2.1). Once you have MIT Scheme, you can run Scheme+ by downloading the source and loading it up (section 2.2).

2.1 Getting MIT Scheme

a. First see if there's already a version of Scheme running on your machine. Typically, you can find out by typing `scheme` at the shell or `M-x run-scheme` in Emacs.[1] If there is a Scheme, this will start it up, and it will print out a herald like this:

```
Scheme Microcode Version 11.146
MIT Scheme running under SunOS
Type '^C' (control-C) followed by 'H' to obtain information about interrupts.

Scheme saved on Sunday November 21, 1993 at 9:15:23 PM
  Release 7.3.0 (beta)
  Microcode 11.146
  Runtime 14.166

1 ]=>
```

The key line in the herald is the one beginning with `Release`. If it's followed by a number beginning with 7.1 or higher you're in luck. If you're using a Sparc, you need at least version 7.3 — earlier versions do not have a native code compiler and the interpreter is much too slow. If you have an appropriate version you can go on to the section on loading Scheme+. Otherwise, continue with the next step, where you will load up an appropriate version of MIT Scheme.

b. If there's not already a Scheme on your machine, you can try to load one up. The appropriate files and instructions are available by anonymous FTP from `swiss-ftp.ai.mit.edu` in these directories:

```
pub/scheme-7.3
pub/scheme-7.4
```

Which version of Scheme should you try to load? The `README` file in each of the above directories describes what machines that version of Scheme runs on. In the past, we recommended using version 7.3 as it is latest version with which Scheme+ has been fully tested. Version 7.4 is an improved Intel x86-only release of MIT Scheme that runs on almost any x86 operating system except DOS. You shouldn't have any problems running Scheme+ with version 7.4, but if you do, please tell us.

We recommend *NOT* using any version of MIT Scheme that you may have installed for 6.001. 6.001 versions of MIT Scheme do not include the compiler. Syntaxing and compiling Scheme+, as described in section 2.2, can make loading and using Scheme+ significantly faster.

Sparc users must use at least version 7.3 to get suitable performance. Previous versions did not have a native code compiler for the Sparc, thus the entire Scheme system was interpreted which meant that it was very slow. Version 7.3 generates native code on Sparcs by compiling Scheme to C and then compiling the C. This gives quite reasonable performance.

The distribution contains pre-compiled executables for various target architectures. Please see the files `README`, `INSTALL`, and `NEWS` for more information about which platforms are supported and installing MIT Scheme.

c. Alternatively, check the information at `http://www-swiss.ai.mit.edu/scheme-home.html`

2.2 Getting Scheme+

2.2.1 Installing Scheme+

a. Copy the Scheme+ implementation to your machine from the course FTP server, `psrg.lcs.mit.edu`. This is the file name:

`pub/6.821/fall-02/code/scheme+/scheme+1.2.scm`

[1] If you have not used Emacs before, see section 3.4 for information about Emacs and key sequences like `M-x`.

While you're at it, you might as well grab `code/ps1/postfix.scm` in preparation for the first problem set.

In the future, we will refer to the directory `pub/6.821/fall-02` on `psrg.lcs.mit.edu` as the *course directory*. We will make all necessary files available there throughout the term.

b. This part is optional but highly recommended. It only need be done once for any version of Scheme+.

Syntaxing the Scheme+ implementation will greatly improve load time and compiling can increase performance. Go to the directory where you put your copy of the Scheme+ implementation and start up a Scheme with the `-compiler` option, either by typing

```
scheme -compiler
```

at the shell or by typing

```
C-u M-x run-scheme
-emacs -compiler
```

in Emacs.

To syntax the implementation, evaluate this Scheme expression:

```
(sf "scheme+1.2.scm")
```

This will print out a lot of stuff, including some warning messages that you can safely ignore, and it will create `scheme+1.2.bin`, which is a fast-loading version of `scheme+1.2.scm`. The difference in loading time between the two is orders of magnitude!

To compile the implementation, evaluate this Scheme expression:

```
(cf "scheme+1.2.scm")
```

Again, ignore the warning messages. This will create a compiled file `scheme+1.2.com`, which should run faster.[2]

2.2.2 Running Scheme+

Start up a Scheme, either by typing

```
scheme
```

at the shell or by typing

```
M-x run-scheme
-emacs
```

in Emacs. Evaluate this Scheme expression:

```
(load "scheme+1.2.scm")
```

Replacing the `.scm` extension with `.bin` will load the syntaxed version. Using the `.com` extension will load the compiled version.

After printing out a herald, this will plop you into a Scheme+ interpreter. Now you're ready to rock 'n roll for 6.821! Unfortunately, syntaxing and compiling Scheme+ programs is not currently supported.

3 Usage Notes

Scheme+ is implemented on top of MIT Scheme, and almost all MIT Scheme features are supported within Scheme+. Additionally, Scheme+ provides some extra features described in appendix A of the Scheme supplement. This section briefly describes those MIT Scheme features that are particularly useful for interacting with Scheme+ as well as notes about using MIT Scheme within Emacs.

[2] Unfortunately, there seems to be a bug in the Sparc compiler for version 7.3, so Sparc users will not be able to compile the Scheme+ implementation. They *can* syntax it, however.

3.1 Handy MIT Scheme Features

- `(pp` *`object`*`)` pretty-prints *object*. That is, it displays *object* on the screen in a readable format. If *object* is a procedure, `pp` prints out its definition.
- `(pwd)` prints the current working directory.
- `(cd` *`string`*`)` changes the working directory to that named by *string*.
- `(exit)` kills Scheme after requesting confirmation.
- *`'#@number`* refers to a value by its identity number. Every compound value has an identity number, which is shown in its `;Value` comment when it is displayed. For example:

```
> (define a (lambda (x) 3))
;Value 27: a

> (pp '#@27)
(named-lambda (a x) 3)
;No value

> (list 1 2 3)
;Value 29: (1 2 3)

> (pp '#@29)
(1 2 3)
;No value
```

3.2 Debugging Support

- `(trace` *`procedure`*`)` prints out information on every call and return of *procedure*.
- `(trace-entry` *`procedure`*`)` prints out information on every call of *procedure*.
- `(trace-exit` *`procedure`*`)` prints out information on every return of *procedure*.
- `(untrace` *`[procedure]`*`)` turns off tracing for *procedure*. Calling `untrace` with no arguments turns tracing off for all procedures.
- `(where` *`procedure`*`)` runs the environment inspector on the environment of *procedure*. The environment inspector supports a wide variety of options accessed by single-letter commands:

```
?   help, list command letters
A   show All bindings in current environment and its ancestors
C   show bindings of identifiers in the Current environment
E   Enter a read-eval-print loop in the current environment
O   pretty print the procedure that created the current environment
P   move to environment that is Parent of current environment
Q   Quit (exit environment inspector)
S   move to child of current environment (in current chain)
V   eValuate expression in current environment
W   enter environment inspector (Where) on the current environment
```

- `(debug)` enters the stack inspector. This is probably the most useful debugging tool, but the hardest to use. The stack inspector also supports a wide variety of commands:

```
?   help, list command letters
A   show All bindings in current environment and its ancestors
B   move (Back) to next reduction (earlier in time)
C   show bindings of identifiers in the Current environment
D   move (Down) to the previous subproblem (later in time)
E   Enter a read-eval-print loop in the current environment
F   move (Forward) to previous reduction (later in time)
G   Go to a particular subproblem
H   prints a summary (History) of all subproblems
I   redisplay the error message Info
J   return TO the current subproblem with a value
K   continue the program using a standard restart option
L   (List expression) pretty print the current expression
O   pretty print the procedure that created the current environment
P   move to environment that is Parent of current environment
Q   Quit (exit debugger)
R   print the execution history (Reductions) of the current subproblem level
S   move to child of current environment (in current chain)
T   print the current subproblem or reduction
U   move (Up) to the next subproblem (earlier in time)
V   eValuate expression in current environment
W   enter environment inspector (Where) on the current environment
X   create a read eval print loop in the debugger environment
Y   display the current stack frame
Z   return FROM the current subproblem with a value
```

The most useful of these are `H`, `G`, and `E`. It is especially useful to type `H` immediately after evaluating `(debug)` because the subproblem history often shows exactly where the error is. `G` and `E` are useful for probing the subproblem history for important information. Using the debugger is by no means easy or natural at first, but it is incredibly powerful once you become accustomed to it.

3.3 Moving Between Scheme+ and Scheme

Those of you who know Scheme may want to access Scheme features within Scheme+ or move between Scheme+ and Scheme. The following Scheme+ procedures are helpful in this regard:

- `(scheme)` leaves Scheme+ and enters Scheme.
- `(scheme+)` leaves Scheme and enters Scheme+.

3.4 Using Scheme in Emacs

The preferred way to run Scheme is within Emacs. Scheme comes with an Emacs configuration file that aids in writing Scheme programs. One of the most important features is automatic indenting and highlighting matching parens. Emacs indents s-expressions in the Scheme pretty-printing style, thus making your code easier to read. Emacs also flashes the cursor to the matching left paren every time you type a right paren, thus helping you to keep track of parentheses.

Emacs commands are invoked by control sequences. For example, holding down the control key and pressing the `x` key followed by holding down the control key and pressing the `f` key will invoke the `find-file` command for loading a file into a buffer. This key sequence is denoted by the shorthand `C-x C-f`. The `find-file` command will then ask for the name of the file to be loaded by putting you in the minibuffer (the one line below the modeline with the time, etc.) with a line like `Find file: /home/szilagyi/`. You can edit the default file name (in this case `/home/szilagyi/`) before pressing return.

You can load MIT Scheme in Emacs by using the `run-scheme` command. To run this command you type the key sequence: press the `x` while holding the meta key, followed by typing `run-scheme`, followed by

pressing *enter*. This key sequence is written `M-x run-scheme` (the trailing *enter* is usually omitted). The key sequence discussed above for `find-file` is a hot-key sequence — you do not have to type the command name. The `find-file` command can also be invoked by `M-x find-file`.

If you want to change the options to Scheme, such as starting it in compiler mode, you must preface your command with `C-u`. If you type `C-u M-x run-scheme`, you will end up in the minibuffer with `Run Scheme: scheme -emacs`. `scheme -emacs` is the shell command used to invoke Scheme with the `-emacs` option telling Scheme to running inside emacs. You can edit the mini-buffer to add the `-compiler` option to start up Scheme in compiler mode. Thus the entire key sequence would be written `C-u M-x run-scheme` *enter* `-compiler`.

Here's a short list of other emacs commands that may be useful. You can always type `C-h C-h` for help. Typing `C-h t` will bring up the Emacs tutorial, which is an easy way to learn how to use Emacs if this is your first time. Typing `C-h m` while in Scheme-mode will describe the special Scheme commands and `C-h f` followed by the name of a command will give you documentation on that command:

```
C-x C-f   find-file             ;; loads file into a buffer
C-x C-s   save-buffer           ;; saves current buffer
C-x b     switch-to-buffer      ;; prompts for buffer name
C-h C-h   help-for-help         ;; emacs help
C-h m     ;; provides mode specific documentation
C-h f     ;; provide documentation for a command
M-x info  ;; runs documentation browser
C-x C-c   ;; exit emacs after asking to save modified buffers
C-a       beginning-of-line
C-e       end-of-line
C-v       ;; move down one page
M-v       ;; move up one page
M-<       beginning-of-buffer
M->       end-of-buffer
```

Typing `M-x run-scheme` will start Scheme in a buffer called `*scheme*`.