An Introduction to Scheme−>C in 19 Prompts

Joel F. Bartlett

12 August 1988

Scheme−>C is an implementation of the language Scheme. Besides the usual interpreter, the implementation includes an unusual compiler which compiles Scheme to C. This allows stand-alone programs and programs combining Scheme and other programming languages. The implementation is also highly portable and when combined with a good C compiler, fairly efficient.

Please consider this annotated typescript an invitation to try it. Questions and comments are encouraged.

```
alerion 1 >man sci

SCI(1)               UNIX Programmer's Manual                SCI(1)

NAME
     sci - Scheme interpreter

SYNTAX
     sci [ option ]

DESCRIPTION
     The sci command invokes a Scheme interpreter.  The language
     accepted by ...
```

Using your favorite editor, create a file containing the Fibonacci function. Scheme code is generally expected to be in files ending with ".sc".

```
alerion 2>more fib.sc
;;; (FIB n) returns the Fibonacci number for n.

(module fib)

(define (FIB n)
    (cond ((> n 1) (+ (fib (- n 1)) (fib (- n 2))))
  ((= n 1) 1)
  ((= n 0) 0)
  (else  (error 'FIB "Argument is out of range: ~s" n))))
```

A ";" indicates that the rest of the line is a comment. The form `(module fib)`, which must

be the first form in the file, indicates that the functions in the file should be part of the module "fib". Typically the module name is the file name (less the ".sc" extension) of the source file.

```
alerion 3 >sci
SCHEME->C -- 08aug88jfb -- Copyright 1988 Digital Equipment Corporation
> (load "fib.sc")
MODULE form ignored
FIB
"fib.sc"
```

`(load "`*file-name*`")` loads a file into the interpreter. Each form in the file is evaluated and the result is printed on the standard output device. Note that the module form is currently ignored by the interpreter. Another way to load the file is to use `(loade "`*file-name*`")` which will also echo the text read from the file onto the standard output file. Since `FIB` was defined when this function was evaluated, a warning message is printed:

```
> (loade "fib.sc")
;;; (FIB n) returns the Fibonacci number for n.

(module fib)

MODULE form ignored
(define (FIB n)
    (cond ((> n 1) (+ (fib (- n 1)) (fib (- n 2))))
  ((= n 1) 1)
  ((= n 0) 0)
  (else  (error 'FIB "Argument is out of range: ~s" n))))
***** FIB is redefined

FIB
"fib.sc"
> (fib 1)
1
> (fib 2)
1
> (fib 0)
0
> (fib -1)
***** FIB Argument is out of range: -1
```

(`trace` *function* ...) allows one or more functions to be traced. (`untrace` *function* ...) removes tracing from selected functions, and (`untrace`) removes tracing from all functions.

```
> (trace fib)
(FIB)
> (fib 5)
(FIB 5)
  (FIB 4)
    (FIB 3)
      (FIB 2)
```

```
          (FIB 1)
          ==> 1
          (FIB 0)
          ==> 0
        ==> 1
        (FIB 1)
        ==> 1
      ==> 2
      (FIB 2)
        (FIB 1)
        ==> 1
        (FIB 0)
        ==> 0
      ==> 1
    ==> 3
    (FIB 3)
      (FIB 2)
        (FIB 1)
        ==> 1
        (FIB 0)
        ==> 0
      ==> 1
      (FIB 1)
      ==> 1
    ==> 2
  ==> 5
  5
  > (untrace)
  (FIB)
```

`(bpt` *function*`)` sets a breakpoint on function entry and exit. At the function call, the arguments are in `*args*` which may be changed. After completing inspection, type D̂ to evaluate the function. On function exit, the result is in `*result*` and the program stops for inspection. To continue with that result, one types D̂ or `(proceed)`. A different result may be returned by entering `(proceed` *expression*`)`. `(unbpt` *function ...*`)` removes breakpoints from selected functions, and `(unbpt)` removes all breakpoints. While at a breakpoint, one may return to the "top-level" interpreter by executing the function `(top-level)`.

```
> (bpt fib)
FIB
> (fib 1)

0 -calls  - (FIB 1)
0- *args*
(1)
0- ^D
0 -returns- 1
0- *result*
1
0- (proceed 23.7)
23.7
```

```
0- ^D
23.7
> (unbpt fib)
(FIB)
```

Breakpoints may also have a boolean function supplied which decides whether to take the breakpoint. Needless to say, such a function can also do things like count the number of times the function is called.

```
> (bpt fib (lambda (n) (set! fibcnt (+ 1 fibcnt)) #f))
FIB
> (set! fibcnt 0)
0
> (fib 5)
5
> fibcnt
15
> (unbpt fib)
(FIB)
> (fib 20)
6765
>^D
alerion 4 >
```

Since `(fib 20)` took a while to compute, it might be a good idea to compile it, so the interpreter is exited, and an augmented version of the interpreter is created which has a compiled version of FIB.

```
alerion 4 >man scc

SCC(1)              UNIX Programmer's Manual               SCC(1)

NAME
     scc - Scheme to C compiler

SYNTAX
     scc [ option ] ... file ...

DESCRIPTION
     The scc command invokes a Scheme compiler which accepts the
     language ...

alerion 5 >scc -i -o sc+fib fib.sc
fib.sc:
fib.c:
SC-TO-C.c:
alerion 6 >sc+fib
SCHEME->C -- 08aug88jfb -- Copyright 1988 Digital Equipment Corporation
> fib
#*PROCEDURE*
> (fib 1)
```

```
1
> (fib 0)
0
> (fib 20)
6765
>^D
```

Now for a little different example, where a Scheme version of the shell command "echo" is created as a stand alone program. The module form now has an additional component, `(main do-echo)`, which indicates that the `do-echo` function is the program main. As with any other UNIX program, the main is called with the arguments from the shell. This is done in Scheme by providing the main with a list of strings which are the arguments.

```
alerion 7 >more echo.sc
;;; ECHO - Echo Arguments
;;;
;;; % echo [options] [args]
;;;
;;; Option:
;;; -n newlines are not added to output

(module echo (main do-echo))

(define (DO-ECHO clargs)
    (let ((nonewline (and (cdr clargs) (equal? (cadr clargs) "-n"))))
 (do ((args (if nonewline (cddr clargs) (cdr clargs)) (cdr args)))
     ((null? args)
      (unless nonewline (newline)))
     (display (car args))
     (if (cdr args) (display " ")))))
alerion 8>
```

The program is loaded into the interpreter and tested with a few possible values. Note that the first argument is always the program name.

```
alerion 8>sci
SCHEME->C -- 08aug88jfb -- Copyright 1988 Digital Equipment Corporation
> (load "echo.sc")
MODULE form ignored
DO-ECHO
"echo.sc"
> (do-echo '("echo" "-n" "a"))
a#F
> (do-echo '("echo"  "a" "b" "c"))
a b c
#F
> (do-echo '("echo" "-n" "a" "b" "c"))
a b c#F
> (do-echo '("echo"))
```

```
#F
> ^D
```

Now, compile it and build a stand-alone program:

```
alerion 9 >scc -o scheme-echo echo.sc
echo.sc:
alerion 10 >scheme-echo *.sc
counter.sc echo.sc fib.sc fsm2.sc fsmexample.sc hello.sc
alerion 11 >scheme-echo -n *.sc
counter.sc echo.sc fib.sc fsm2.sc fsmexample.sc hello.scalerion 12 >
```

The next example shows the interface to routines written in other languages by building a program which uses the routines in the C library (described in the *ULTRX-32 Programmer's Manual*) to print out the current Greenwich mean time.

```
alerion 12 >more gmt.sc
;;; Print current GMT on standard output.

(module gmt (main gmt))

(define-c-external (time pointer) int "time")
(define-c-external (gmtime pointer) pointer "gmtime")
(define-c-external (asctime pointer) pointer "asctime")

(define (GMT clargs)
    (let ((current-time (make-string 4)))
 (time current-time)
 (display (c-string->string (asctime (gmtime current-time))))))
```

The procedure *time* stores the number of seconds since GMT. Jan. 1. 1970 in the location referenced by *pointer*. *gmtime* converts that value to a *tm* structure and returns a pointer to it. *asctime* then converts the referenced *tm* structure to a string and returns a pointer to it. In order to display it, *c-string-¿string* is used to make a Scheme copy of the string.

```
alerion 13 >scc -o gmt gmt.sc
gmt.sc:
alerion 14 >gmt
Thu Aug 11 22:19:08 1988
```

To allay any doubts that this implementation might not be Scheme, we conclude with the following "proof by example", produced by Eugene Kohlbecker:

```
alerion 15 >more mondo.sc
(module mondo (main call-mondo))

(define (call-mondo clargs) (mondo-bizarro) (newline))

(define (mondo-bizarro)
    (let ((k (call-with-current-continuation (lambda (c) c))))
```

```
 (display 1)
 (call-with-current-continuation (lambda (c) (k c)))
 (display 2)
 (call-with-current-continuation (lambda (c) (k c)))
 (display 3)))
alerion 16 >sci
SCHEME-C -- 08aug88jfb -- Copyright 1988 Digital Equipment Corporation
> (load "mondo.sc")
MODULE form ignored
CALL-MONDO
MONDO-BIZARRO
"mondo.sc"
> (call-mondo '())
11213
#F
> ^D
alerion 17 >scc -o mondo mondo.sc
mondo.sc:
alerion 18 >mondo
11213
alerion 19 >logout
```