```c
/* Michael E. Sparks, 11-17-16 (updated 10-16-20)

   caller.c - Driver application to demo interfacing C
              with Scheme code using the Guile library. */

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <libguile.h>

int main(int argc,char **argv) {
  unsigned int eltCnt=5,
               gameIndex;
  SCM exprToCall;

  if(argc > 1 && ((eltCnt=atoi(argv[1])) < 1 || eltCnt > 15)) {
    printf("Usage: %s (1 <= N <= 15)\n",argv[0]);
    return(EXIT_FAILURE); /* synonymous with return(1) */
  }

  scm_init_guile();

  do {
    printf("\nPlease select a game to play:\n\
  (0) *quit playing these games*\n\
  (1) Conway's \"Say It!\" Sequence\n\
  (2) Prime Finder\n\
  (3) Countdown\n\n");

    scanf("%u",&gameIndex);
    printf("\n");

    switch(gameIndex) {
      case 0 :
        gameIndex=false;
        break;
      case 1 :
        scm_c_primitive_load("sayit.scm");
        exprToCall=scm_variable_ref(scm_c_lookup("sayit"));
        scm_call_1(exprToCall, scm_from_int(eltCnt));
        break;
      case 2 :
        scm_c_primitive_load("fermat-little-theorem.scm");
        exprToCall=scm_variable_ref(scm_c_lookup("list-primes"));
        scm_call_0(exprToCall);
        break;
      case 3 :
        scm_c_primitive_load("sayit.scm");
        exprToCall=scm_variable_ref(scm_c_lookup("countdown"));
        scm_call_1(exprToCall, scm_from_int(eltCnt));
        break;
      default :
        printf("I did not recognize your selection.\n");
        gameIndex=true; /* not technically necessary! */
    }
  } while(gameIndex);

  return(EXIT_SUCCESS); /* synonymous with return(0) */
}
```

```scheme
;; Michael E Sparks, 11-17-16 (updated 10-16-20)

;; sayit.scm - Implementation of Conway's "Say It" sequence

;; Build up a list of dotted pairs--the car of each
;; such pair gives the length of the run, and the
;; cdr denotes the value of the run. The say-it sequence
;; is based on integer-valued symbols, though extension to
;; other classes could be accommodated by modifying the
;; "equals" predicate to suit. That's left as an exercise.
(define (markuprun type runlen rest)
  (cond ((null? rest) (cons (cons runlen type) '()))
        ((= type (car rest)) (markuprun type (1+ runlen) (cdr rest)))
        (#t (cons (cons runlen type)
                  (markuprun (car rest) 1 (cdr rest))))))

;; Converts a list of dotted pairs to a list of atomic elements.
(define (flatten src)
  (if (null? src)
      '()
      (append (list (caar src) (cdar src))
              (flatten (cdr src)))))

;; In Conway's say-it sequence, elements 2, 3, ..., N
;; depend only on the immediately preceding element.
(define (nextelt prevelt)
  (flatten (markuprun (car prevelt) 1 (cdr prevelt))))

;; Returns list of lists, each of which
;; is an element of the say-it sequence.
(define (sayitbuilder max)
  (let ((base (list 1)))
    (define (sayitaux prevelt currcnt)
      (if (>= currcnt max)
          '()
          (cons (nextelt prevelt)
                (sayitaux (nextelt prevelt) (1+ currcnt)))))
    (append (list base) (sayitaux base 1))))

;; Report elements of sequence in a print-friendly manner.
;; There's some subtle "for-each" vs "map" business going on
;; here, in particular w/r/t return values vs side effects;
;; grokking it's left as an exercise.
(define (sayitwriter seq)
  (let* ((dispn (lambda (n) (display n)))
         (procl (lambda (l) (map dispn l) (newline))))
    (for-each procl seq)))

;; Use the following expression as the principle "hook" for
;; calling code written in C.
(define sayit (lambda (x) (sayitwriter (sayitbuilder x))))

;; Shuttle launch sequence -> alternate behavior
(define (countdown x)
  (if (> x 0)
      (begin (display x) (newline) (sleep 1) (countdown (1- x)))
      (begin (display "Blastoff!") (newline))))
```

```scheme
;; Michael E Sparks (10-16-20)
;; Scheme code for efficiently finding primes

#|
Sketch out in BASIC what we're after:

$ cat > prime.bas << EOF
> 10 REM Brute-force prime number finder
> 20 FOR N = 50 TO 2 STEP -1
> 30 P = 1
> 40 FOR M = 2 TO N - 1
> 50 Z = N MOD M
> 60 IF Z = 0 THEN
> 70 P = 0
> 80 END IF
> 90 NEXT M
> 100 IF P = 1 THEN
> 110 PRINT N
> 120 END IF
> 130 NEXT N
> 140 PRINT 2
> EOF
bwbasic
Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
Copyright (c) 1993, Ted A. Campbell
Copyright (c) 1995-1997, Jon B. Volkoff

bwBASIC: load "prime.bas"
bwBASIC: run
47
43
41
37
31
29
23
19
17
13
11
7
5
3
2
bwBASIC: quit
|#

;; Always seed a random number generator!
(set! *random-state* (random-state-from-platform))

;; Scheme has a built-in for this (expt base exponent),
;; but we'll roll our own for illustrative purposes.
(define (power b p)
  (if (= p 0)
      1
      (* b (power b (- p 1)))))

(define (fermat-test n)
  (define (exp-then-mod base exp mod-rhs)
```

```
    (remainder (power base exp) mod-rhs))
  (let ((a (+ 1 (random (- n 1)))))
    (= (exp-then-mod a 1 n)
       (exp-then-mod a n n))))

(define (seems-prime? cand times-to-test)
  (cond ((= times-to-test 0) #t)
        ((fermat-test cand)
         (seems-prime? cand (- times-to-test 1)))
        (#t #f)))

(define (is-prime? cand)
  (define (test-it cand div)
    (cond ((or (= cand 2) (= div 1)) #t)
          ((= (remainder cand div) 0) #f)
          (else (test-it cand (- div 1)))))
  ;; Suppose cand % div = 0. Then, so too
  ;; does cand % (cand / div) = 0. However,
  ;; at least one of these <= sqrt(cand).
  ;; Thus, run-time of Omega(sqrt(cand))
  ;; rather than Omega(cand).
  (test-it cand (ceiling (sqrt cand))))

;; I would generally prefer to do the following recursively,
;; but am demonstrating here that Lisp can also accommodate an
;; iterative/ imperative approach (in contrast to pure functional
;; languages such as Haskell).
(define (list-primes)
  (display "Ceiling on primes? ")
  (let ((ceil (read)))
    (do ((p 2 (+ p 1)))
        ((> p ceil))
      (if (and (seems-prime? p 5) (is-prime? p))
          (format #t "~s\n" p)))))
```