The Scheme of Things:
Streams versus Generators

William Clinger
Tektronix, Inc.
willc%tekchips.tek.com@tek.cs.net

With this article I'm changing the series title to "The Scheme of Things". The original title was a weak pun on the technical term *environment*. An environment, you see, associates names with their meanings. In the Scheme environment, for example, the word "environment" has the meaning just stated, while in some other parts of the programming language world the word "environment" has a less technical meaning that refers to the stuff that comes with a programming system. One of my purposes has been to explain the meaning of terms in the Scheme environment. Sometimes a word like "environment" or "stream" means something different in Scheme than it does in some other dialect of Lisp, usually because the Scheme terminology follows that of another well-established language or group of languages.

* * *

Following a discussion of the advantages of lazy evaluation on the `comp.lang.misc` mailing list, Kenneth Almquist asked an excellent question: "Has anyone made any comparisons between lazy evaluation and coroutines?" Herewith is such a comparison for the special case of streams versus generators.

A *stream* is a lazy list. Unlike an ordinary list, whose elements must already have been computed when the list is created, the elements of a stream are computed only on demand. Streams are particularly convenient for representing infinite sequences. For example, if `cons-stream` is a constructor for streams analogous to `cons`, then the sequence of natural numbers can be defined by:

```
; The stream of integers >= n.

(define (integers n)
  (cons-stream n (integers (+ n 1))))

(define natural-numbers (integers 0))
```

The title of one of the first papers to advocate the use of streams, "CONS should not evaluate its arguments" [1], shows why `cons-stream` can't be a Scheme procedure and must be a macro instead. The `head` and `tail` procedures that extract the "car" and "cdr" of a stream do evaluate their arguments, though.

Over 2200 years ago, before even Lisp was invented, Eratosthenes of Cyrene conceived one of the most beautiful programs ever to use streams. His idea was to compute all the prime numbers by starting with the stream of integers beginning with the first prime, 2, striking out all multiples of 2, taking the next prime, 3, striking out all multiples of 3, taking the next prime, striking out all its multiples, and so on. Today his program is widely known in the form of a benchmark that calculates the first 1899 primes 10 times, but we can better appreciate his genius by casting his algorithm in its more general form using streams:

```
; Given a stream s of integers in increasing order, returns the
; stream obtained by removing all multiples of n.

(define remove-multiples
  (letrec ((loop
            (lambda (s n m)
              (cond ((< (head s) m)
                     (cons-stream (head s) (loop (tail s) n m)))
                    ((= (head s) m)
                     (loop (tail s) n (+ m n)))
                    (else (loop s n (+ m n)))))))
    (lambda (s n)
      (loop s n (+ n n)))))

(define (sieve s)
  (cons-stream (head s)
               (sieve (remove-multiples (tail s) (head s)))))

(define primes (sieve (integers 2)))
```

It takes almost no time at all to compute the stream of all prime numbers, because lazy evaluation delays all the time-consuming calculation. You pay the piper when you print them, though, because the delayed calculations must be performed on demand.

A *generator* for a sequence is a procedure that returns successive elements of the sequence on successive calls. For example, a generator for the sequence of natural numbers can be written as:

```
; Given n, returns a generator for the integers >= n.

(define (integers n)
  (lambda ()
    (let ((ans n))
      (set! n (+ n 1))
      ans)))

(define natural-numbers (integers 0))
```

Generators are quite different from streams because invoking a generator changes the state of the generator. Taking the head or tail of a stream has no such side effect (unless the delayed calculation has side effects), which is why streams are used prominently in functional programming. Generators, which rely entirely on side effects, are simple examples of object-oriented programming in the Smalltalk sense.

The sieve of Eratosthenes looks much the same whether programmed using streams or generators:

```
; Given a generator g of integers in increasing order, returns a
; generator that generates the original integers less multiples of n.

(define (remove-multiples g n)
  (letrec ((m (+ n n))
           (self
            (lambda ()
              (let loop ((x (g)))
                (cond ((< x m) x)
                      ((= x m) (set! m (+ m n)) (self))
                      (else (set! m (+ m n)) (loop x))))))))
    self))

(define (sieve g)
  (lambda ()
    (let ((x (g)))
      (set! g (remove-multiples g x))
      x)))

(define primes (sieve (integers 2)))
```

For this problem at least, streams and generators both provide a short, clear solution.

Neither solution is as efficient as the sieve benchmark, which allocates a fixed array of flags, but the benchmark never gets beyond the first 1899 primes no matter how large a machine it runs on. The stream and generator solutions will calculate all the primes—provided we are infinitely patient and can supply a machine equipped with infinite RAM.

I tested the performance of the stream and generator solutions on a Macintosh II. The program using generators was able to find the first 1000 primes over five times as fast as the program using streams, and was able to find almost twice as many primes before overflowing the heap. While I would expect the program using generators to perform better than the one using streams when run in other implementations of Scheme as well, the differences in performance might not be so pronounced. MIT Scheme, for example, features special support for streams, but the simple implementation of streams I used in MacScheme comes straight out of *Structure and Interpretation of Computer Programs* [3] and the *Revised*^{3} *Report on the Algorithmic Language Scheme* [4]. Languages specifically devoted to lazy evaluation, such as Lazy ML, might well be able to support streams that perform even better than generators in Scheme.

Streams have one major advantage over generators: Since they don't depend on side effects, streams can be shared. We can define a single stream of primes and use it in all parts of a program that need to know about primes. Generators, on the other hand, get used up. If two modules of a program are drawing on the same generator of primes, then neither module will get to see all the primes. We must somehow create as many generators as there are consumers. This makes generators harder to work with than streams.

It wasn't obvious to me that the sieve of Eratosthenes could use generators without running into this problem. My original program had an unrelated bug, which I at first thought was caused by a generator with more than one consumer. In fact, I rewrote the program so that it cloned generators on the fly to avoid any sharing. I realized my mistake only after the rewritten program behaved exactly the same as the original.

In contrast, the program using streams ran correctly the first time I tried it, and I wrote it first. That

proves nothing, of course, except perhaps that I am more comfortable with functional programming than with object-oriented programming.

* * *

Eratosthenes of Cyrene was the first person to measure the diameter of the earth with any real accuracy. An astronomer and poet, he became director of the great Alexandrian library, invented a calendar with leap years, and contributed to the science of map-making [5].

* * *

I thank David Wise for his help with this article. He offered the following classification for academic questions like the one that inspired this article: A good question is one the professor has anticipated and is therefore prepared to answer. A very good question is one the professor hasn't anticipated, but is able to answer anyway. An excellent question is one the professor hasn't anticipated and can't answer.

Next time I'll report on the meeting of the Scheme language design community that took place at the end of June at MIT, and I'll try to explain the `mondo-bizarro` teaser I dropped on you in my article on continuations.

* * *

[1] D P Friedman and D S Wise. CONS should not evaluate its arguments. In S Michaelson and R Milner [editors], *Automata, Languages, and Programming.* Edinburgh University Press, 1976, 257–284.

[2] P Henderson and J H Morris, Jr. A lazy evaluator. In *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages.* ACM, January 1976, 95–103.

[3] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[4] Jonathan Rees and William Clinger [editors]. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21, 12, December 1986, 37–79.

[5] *Encyclopaedia Britannica,* 15th edition.