Eager Comprehensions in Scheme

The design of SRFI 42

Sebastian Egner

Philips Research Laboratories, The Netherlands
sebastian.egner@philips.com

Abstract

This article is about a certain style of programming iterative programs. It is based on a concept we have named "eager comprehension," which is a convenient and efficient alternative to tail recursion, do-loops, and lazy list comprehensions (aka "ZF expressions"). Eager comprehensions are syntactic forms that encapsulate the details of an accumulation process (counting elements, creating a list, etc.). Within these forms, expressions called generators hide the details of enumerating basic sequences (running through a list, through a range of integers, etc.). By combining these elements in a clearly structured and well-defined way, a concise and powerful notation for writing loops emerges.

Of course, this style of programming is not new—it is implicitly present in any form of loop-macro already—and so we discuss several concrete designs that aim for the same goal. Surprisingly, however, none of these designs has had much impact on Scheme, despite the fact that their common floor plan has been around for decades. A particularly clean new design, SRFI 42, on the other hand has already made some friends in the first few years of its existence. Explaining the design and implementation of SRFI 42 constitutes the main part of this article.

1. Introduction

The original motivation for working on a library for comprehensions in Scheme was my dissatisfaction with the available mechanisms for writing trivial loops. In addition, I wanted to create an efficient mechanism for converting data structures without a quadratically increasing number of conversion operations named *chalk->cheese*.

The most basic example for a trivial loop is the construction of a list of the first n non-negative integers, using the constructs available in the Revised^5 Report on the Algorithmic Language Scheme (R^{5}RS) [1] only. Maybe the shortest[1] and clearest (!?) expression for this is

```
(do ((k (- n 1) (- k 1))
     (x '() (cons k x)) )
  ((< k 0) x) )
```

[1] Please let me know if you can do shorter than this in R^{5}RS.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.
Copyright © 2005 Sebastian Egner.

This is terrible, not so much for the number of key strokes but as an example where details obscure intention.

In SRFI 42 that would be `(list-ec (: k n) k)`, for interactive use, or `(list-ec (:range k n) k)` if speed is worth another five key strokes. Since this article is not about the specification of SRFI 42, but about the design principles, a basic familiarity with the following document will be assumed from now on:

```
http://srfi.schemers.org/srfi-42/srfi-42.html
```

(Alternatively, there is a brief introduction in the appendix.) While initially the goal was adapting the comprehensions found in Haskell to Scheme, a number of insights turned this enterprise into a whole new direction and eventually led to the concept later coined "eager comprehensions." These ideas can be summarized as follows:

1. Truly lazy comprehensions are not an attractive option in Scheme because the overhead for non-strict data structures and explicit handling of continuations is high. Moreover, lazy comprehensions can be confusing in the presence of side-effects.
2. While list comprehensions and list generators are sufficient for comprehensions in lazy languages, in eager languages it is essential to be able to add application-specific comprehensions and generators easily—and without modifying the existing ones.
3. While simple comprehensions resemble mathematical set comprehensions, more complex expressions increasingly look like nested and parallel loops with accumulation of the results. In fact, that is what they are.
4. The fundamental eager comprehension has nothing to do with lists, but executes a command repeatedly according to its generators. The fundamental eager generator repeatedly modifies a state explicitly.

After these insights it was obvious that "bringing Haskell's comprehensions to Scheme" is the wrong goal to pursue. The more interesting question is "What would be a useful corresponding concept in an eager programming language?" The answer is quite surprising:

> *Eager comprehension:* A convenient style of programming nested and parallel loops with accumulation of results. Ideally, scope and syntax are easy to remember and the irrelevant details of the iteration are hidden from the user.

The concept can also be interpreted as an (essentially syntactic) abstraction mechanism from details of iteration: if you have a new data structure that has given rise to some natural iteration, then it might pay to encapsulate the details of this iteration process in a generator. Similarly, if there is a natural way of constructing a data structure from a sequence of states—a comprehension might be useful to applications.

Structure of this article. The remainder of this article is organized as follows. In Section 2 different notions of "comprehension" are introduced. These notions are related but must not be confused. In particular the term "eager comprehension" is being used as a reminder that this concept has in fact very little to do with lazy list comprehension.

Section 3 continues with stating the major design issues for (eager or lazy) comprehensions as a general and practically useful language construct. Section 4 discusses a number of concrete designs of loop facilities and comprehensions for the Lisp family of languages, and related work. The implementation of SRFI 42 is the subject of Section 5. It explains the overall structure and implementation strategy used in the reference implementation (which, unfortunately performance-wise, is the only one available till today). In Section 6 the performance of the portable reference implementation is compared with other libraries. Finally, for entertainment, Section 7 presents a modular way of adding *lazy* comprehensions to SRFI 42.

2. Concepts of comprehensions

In this section we briefly review different concepts of comprehension. For the sake of clarity we will always refer to them by a longer name than just "comprehension."

Set comprehension. The mathematical notation[2]

$$\text{“}\{f(x) \mid P(x), x \in S\}\text{”}$$

denotes the set of all values of the function f for arguments in the set S and satisfying the predicate P. The notation explicitly refers to a candidate element x, a predicate P, a universe S, and a mapping f. This notation is called a *set comprehension*.

The stated form is maybe the most frequently seen, but it is not the most fundamental. The most basic form of set comprehension is $\{y \mid Q(y)\}$, where $y = f(x)$, $P(x)$, and $x \in S$ have been combined into $Q(y)$. This concept, i.e. denoting a set defined by a predicate (formula), is the core of what is meant by "comprehension." While this concept has been in use for a long time already, it was not before the development of axiomatic set theory by Zermelo, Fraenkel and others in the 1920s that the idea was studied systematically. The notion of set comprehension, and its notation, is so natural that it has gradually become a mathematical standard, i.e. the reader of a mathematical article is expected to understand it without definition.

Comprehensions in programming languages. The notational convenience of set comprehension has inspired programming language constructs with similar intent: name the data structure defined by an expression for its elements. For example, in the SETL language [19, 20]

```
{n**2 : n in {0..9}}
```

denotes the set of the first ten integer squares. This construct, however, does not only specify the result but also an *algorithm* for constructing the result ("execute a loop over `n`, square the values, collected them in a data structure"). It is often convenient to ignore the algorithmic aspect, but most of the time this is not possible—after all, algorithms do take time, or may not terminate at all. For this reason, set comprehension in mathematics and in programming languages should never be confused.

Lazy comprehensions. While comprehensions were contained in some eager (aka call-by-value) programming languages for a long time, they only became popular once they were introduced for lazy lists in lazy (aka call-by-need) functional programming languages

[2] Instead of "|" also ":" and "," are in use.

	primitive	*purpose*
set	set	denote a set by properties
lazy	lazy list	sequence processing
eager	side-effect	writing nested and parallel loops with accumulation of results

Table 1. Different concepts of "comprehension."

(mostly based on a typed λ-calculus with normal order reduction) in the early 1980s. In contemporary syntax (Haskell), for example,

```
[x*x | x <- [0..]]
```

denotes the (infinite) lazy list of all integer squares. Its elements will be made explicit once they are needed.

Such a *lazy comprehension* provides a convenient notation for processing lazy lists by means of mapping, filtering, and concatenation. The primitive lazy comprehension (written `[`*exp*`|`*qual*$^+$`]` in Haskell) constructs a lazy list, and the primitive lazy generator (`<-`, read 'drawn from') binds a variable[3] to the elements of a lazy list. In addition, several generators can be nested, elements can be filtered from the sequence, and local variables can be defined.

Lazy list comprehensions are widely accepted due to their concise notation, and good readability in most cases. Their efficiency is as good as any (lazy) alternative. Their primary shortcoming is an implicit tendency to overuse them, i.e. to write complicated nested lazy comprehensions where an appropriate abstraction had better been introduced. The decreasing readability of more complicated lazy comprehensions is probably due to the use of infix operators and the "$[expr|outer..inner]$" scoping rule, which is not simply left to right.

From the point of view of programming language design, it is most informative to recall the historical development of lazy comprehensions [18, Chapter 7]—in particular that their true nature was not fully understood for a long time: lazy comprehensions were first introduced as part of the NPL language (Burstall, 1977) [25]. In NPL, however, comprehensions construct a set of objects. While this construct is closest to the mathematical notion, *lazy sets* are not nearly as useful as *lazy lists* are. It appears that lists and graphs are more fundamental to programming than sets (unordered collections); in addition, lists (in particular lazy lists) are a universal and natural mechanism of communication between different parts of a program. Consequently, set comprehensions were not essential and when NPL evolved into the Hope language (Burstall, 1980) [26] lazy comprehensions where not included.

Lazy list comprehensions made their debut in the KRC language (Turner, 1981) [27] as "ZF expressions." Later they were included in several other functional programming languages like Miranda [28, 29] (Turner, 1985). But still mathematical beauty has distracted the mind from proper programming pragmatics for some time: generators in lazy list comprehensions can denote infinite iterations. Hence, from a mathematical point of view the most natural way of advancing nested generators is by (Cantorian) diagonalization, also known as "dove-tailing." This is the only way of reaching every pair eventually in the case of an infinite inner generator. While diagonalization looks like a good idea at first, it is not. Mathematical "eventually" can be a long time, and in practice diagonalization is not worth a lot. Thus lazy list comprehensions evolved to run the generators in the straight-forward way, i.e. exhausting the inner loop before advancing the outer loop, while the diagonalizing variants slowly went out of fashion (not without constantly being reinvented).

[3] A pattern possibly containing variables to be precise.

Specialized eager comprehensions. Encouraged by the success of lazy comprehensions, designers of eager programming languages recently started to include comprehensions again. E.g. Python [11] contains list comprehensions. While these *eager comprehensions* can be quite useful, in particular for interactive use and scripting, they are much less universal in nature than their lazy counterparts. This is explained in greater detail in Section 3.3.

Eager comprehension as abstraction of iteration. Surprisingly, this perceived limitation is again due to a lack of understanding for the true nature of comprehensions, eager comprehensions this time. As explained above, lazy comprehensions for lists are fundamental. For eager comprehensions, however, *side-effect* (state) is the most basic concept[4].

As indirect evidence of this fact consider that any eager comprehension can be implemented in terms of

(`do-ec` *qualifier** *command*),

which executes *command* for each state in the sequence defined by the generators and tests *qualifier**. Similarly, each eager generator can be implemented in some form of state-transforming iterator, in the sense of `do`. Amazingly, this insight—which is made explicit here—is already implicit in the design of nearly any loop facility for the Lisp family of languages, but it has not been acknowledged as such.

While the "can be implemented by"-relation usually does not lead to the most fundamental concept, it does so in this case. Consider the alternative of implementing the eager comprehensions in terms of (eager) lists: the resulting implementation will be horrible! An accumulation process (e.g. counting) cannot start until the last element of the enumerated sequence has been produced and stored in a data structure. The resulting loss in performance, as a function of sequence length, is in fact unbounded.

Being built on this insight, SRFI 42 eventually reduces any comprehension to `do-ec`, and any generator to `:do`—which is some flexible but fixed loop structure (Section 5.2) based on explicit state transformation. In combination with a number of rules simplifying the syntax and introducing a clean scoping rule, this results in a facility for iteration that is both efficient and convenient.

3. Design considerations

In this section we discuss the main issues that affect the usefulness of a programming language construct for eager comprehensions, or for writing nested and parallel loops with accumulation of the result. We approach these issues by exploring design alternatives: which design decisions exist and what are their implications? Our primary goal is not a coherent and complete theory, but rather an informal discussion of the relative benefits of various designs in terms of convenience and effectiveness of the language construct for writing programs.

3.1 Mental complexity

Maybe the most important consideration is what could be called "mental complexity." As an anecdotal quantitative measure of mental complexity we propose to count the number of times the reference manual of a loop construct was consulted when reading other people's loops, multiplied by the years of experience of the reader with that particular construct.

More seriously, we would like to point out that any concept for eager comprehensions, or loops, represents a trade-off between simplicity and flexibility. This follows from the fact that loops cover a large scale of complexity in programs, from simple repetition to complicated nested and parallel actions with several conditions in between and numerous invariants. In effect, designing "the" loop construct might not be the right goal to aim for, and it might also be necessary to predefine frequent idioms of loops. An import tool for flexibility is orthogonality—for example in SRFI 42 every generator can be modified by adding another termination condition.

While the orthogonality idea is strong in Scheme, the iterative part of it has been somewhat neglected. (More on that in Section 4.1.) Nevertheless, the looping constructs that *are* available in $\mathrm{R}^5\mathrm{RS}$ are not too complicated to remember, i.e. mental complexity is relatively low. At the other extreme end, Common Lisp's `loop` might be found—highly flexible but also highly complicated. (Refer to Section 4.5.)

3.2 Interactive use vs. batch mode

Scheme can be used as an interactive system or for writing batch programs. Although these modes are just two extremes of an entire spectrum of human-computer interaction they are useful abstractions for evaluating designs. The two modes impose conflicting requirements: concise notation and flexibility is most important to interactivity, while robustness, efficiency, and readability are primary concerns for batch mode.

In the case of eager comprehensions, the key to efficiency is the use of typed state-based generators, i.e. programs that enumerate a sequence by modifying a local state (values of variables), the state being of statically known type (e.g. an integer counter). Note that this does not necessarily mean the state is updated by using `set!`, it could also mean that the state is updated by rebinding (as with tail-recursive procedures). If the state is represented in boxed data structures, or if each loop iteration requires dispatching, performance usually suffers. For this reason, most loop constructs for Scheme (or Lisp in general) concentrate on the batch mode, only. In SRFI 42, on the other hand, the requirements of interactive and batch mode are addressed by two different mechanisms (typed and dispatched generators) which can be mixed freely.

3.3 Modularity

Modularity for comprehensions means that new types of generators and new types of comprehensions can be added without modifying the already existing generators or comprehensions. For the sake of illustration, let us assume the new type "Fooziset" is yearning for comprehension.

In lazy comprehensions modularity is for free: adding a generator means writing a function returning a lazy list to be used on the right-hand side of the single binding and enumeration construct ("`<-`" in Haskell). Adding a comprehension means writing a function processing a lazy list, possibly constructed by a comprehension. In effect, the comprehension

```
foozi_of_list [x | x <- list_of_foozi s]
```

produces an element-wise copy of a Fooziset, whatever that actually means.

For eager comprehensions, on the other hand, modularity is a challenge. And what is more important, modularity is the key for creating an abstraction that goes beyond a mere idiom for frequent programs! Unfortunately, the importance of modularity for eager comprehensions has long been underestimated. Most designs make it either outright impossible to add new generators and comprehensions, or this is inconvenient and cumbersome. In effect, the users of the mechanism do not take the trouble of adding the comprehensions and generators they really require in the application—wasting a great opportunity for useful abstraction.

[4] We use the term 'state' here in an informal way, refering to the status of all bits that could possibly alter the future of an iteration. Later, in Section 3.5, we will clarify that a sequence of states may actually mean a sequence of binding environments.

scoping convention	*examples*
$[expr \mid inner..outer]$	Magma
$[expr \mid outer..inner]$	Haskell, Python, Erlang, (Mathematica), Swindle, …
$[inner..outer \mid expr]$	—
$[outer..inner \mid expr]$	SRFI 42
$[[expr \mid inner] \mid ..outer]$	Mathematica
$[outer \mid ..[inner \mid expr]]$	Ruby, Perl, GAP

Table 2. Possible scoping conventions

For example, a library for number theory would include a generator enumerating the prime divisors of an integer, together with its multiplicity, because that is what is needed in many places. A library for graphs, on the other hand, would provide generators for enumerating the vertices of a graph, or the edges leaving a particular vertex. All this is only possible through modularity.

For the design of SRFI 42 modularity has always been one of the top priorities (right after efficiency), and the biggest challenge. The breakthrough came when I learned about the technique of using hygienic macros in continuation-passing style (CPS) [8]. This mechanism allows fully modular definition of eager generators, and it has prompted me to start the design again from scratch. The result will be explained in greater detail in Section 5.

3.4 Scope

Eager comprehensions are programming language constructs for writing loops. As such they include syntactic binding forms for the loop variables. Where there is binding, there is scope. This means a loop variable is visible to some parts of the program but not to others—irrespective of whether this scope is specified or not, or whether there are simple rules to remember it. We emphasize this trivial fact because a conscious design of the scope is another critical factor for useful eager comprehensions.

In order to be able to talk about scoping, a language is needed to represent different approaches. For this consider the following simplified view of a comprehension: a comprehension consists of an expression *expr* and zero or more nested qualifiers *inner*, …, *outer*. If the qualifiers are generators, *inner* denotes the one spinning fastest and *outer* the one spinning slowest. Clearly, this terminology only makes sense if *inner* is in the scope of all bindings introduced by *outer*, and if *expr* is in the scope of both *inner* and *outer*. In other words, *expr*, *inner*, *outer* are pieces of code with a certain scoping relation (and control flow) with respect to one another. These pieces can then be composed into a comprehension syntax using '[', ']', and '|'. All possibilities, together with examples, are listed in Table 2. Some arguments are:

1. It is an advantage to have eager comprehensions mimic the notation of set comprehensions because it is widely known. Set comprehensions use the [*expr*|*qualifier**] convention, where the nesting of the qualifiers is not fixed and must be deduced from the context. For simple comprehensions, this is no problem and the mathematical notation looks extremely familiar.
2. The most simple conventions nest scope in one direction, i.e. $[expr \mid inner..outer]$ or $[outer..inner \mid expr]$. In a syntactically impoverished language like Scheme this is particularly attractive.
3. More complicated comprehensions will increasingly look like explicitly nested loops (`do`, named-`let`), and possibly be mixed with them. In Scheme, bindings are always introduced *before* the body, so it is an advantage to have outer bindings appear first.

These contradicting preferences naturally lead to the most popular choice $[expr \mid outer..inner]$ because it looks like a set comprehension (1.) while introducing bindings left-to-right (3.); refer to Table 2.

For SRFI 42 linearity in scope was considered most important (2.), which together with Scheme's preference for left-to-right binding (3.) leads to $[outer..inner \mid expr]$. In effect, SRFI 42 sports an extremely simple scoping rule:

> *The bindings introduced by a generator are visible to all subsequent expressions (qualifier or other) of the same comprehension, and only to these*[5].

While in principle it would also be possible to have a compiler derive the nesting of the qualifiers from the dependency graph, this is a fundamentally bad idea. It would allow reordering the control flow by renaming variables, hashing readability in the process.

3.5 The meaning of state

As the Scheme language supports genuine state and destructive modification of data structures, it is important to clarify what is actually meant by 'iteration state.' More precisely, the designer of eager comprehensions needs to take position with respect to the following questions:

1. What is it supposed to mean if the payload of a generator retains (a reference to) an iteration variable, and uses it in later iterations or even outside the loop?
2. If the payload modifies an iteration variable?
3. If the payload modifies the loop-defining arguments or defining data structures while a loop is in progress?

Before considering possible approaches to these questions, recall that Scheme uses the following model of 'variable' [1, Section 3.1]:

> An identifier that names a location is called a variable and is said to be bound to that location. The set of all visible bindings in effect at some point in a program is known as the environment in effect at that point. The value stored in the location to which a variable is bound is called the variable's value.

Concerning the first semantic question, consider the following program (in SRFI 42 syntax):

```
((cadr (list-ec (:range n 3) (lambda () n))))
```

The result of this expression depends on how `:range` updates its loop variable: by rebinding or by state modification?

In the state modification model, the variable `n` is bound to a single location, and `set!` is used during the iteration to store the integer for that iteration. In effect, the three procedures in the list constructed by `list-ec` contain a reference to the *same* location—and the result of calling any of these procedures will be the state after the entire loop. So the result will be either `2` or `3`, depending on the way the loop modifies `n`. In this model, iteration enumerates a sequence of states stored in a given set of locations.

In the rebinding model, `n` is bound to a new location for every iteration. In this case, the three procedures each retain a different location, and the result is `1`. The rebinding model has been adopted for the iteration constructs of Scheme [1, Section 4.2.4], probably due to a desire for conceptual simplicity. Consequently, it is also the choice for SRFI 42. It should be mentioned that the overhead of rebinding is the same as for any other tail-recursive procedure, and these are supposed to be efficient in Scheme.

[5] As with everything in Scheme there is no way to enforce this, but SRFI 42 is built on this rule; users may have reason to deviate from this but it is not encouraged.

Concerning the second semantic question, consider the following program (again in SRFI 42 syntax):

```
(list-ec (:range n 3) (begin (set! n 2) n))
```

The result of this expression depends on whether `:range` uses the variable `n` itself to hold the state of the iteration (in which case the result is `'(2)`), or if `n` is just a copy of the (hidden) state of the iteration (in which case the result is `'(2 2 2)`).

In Scheme [1, Section 4.2.4], named-`let` and `do` provide access to the state of the iteration itself. This allows arbitrary modification of the state, which can sometimes simplify termination conditions. For eager comprehensions, however, the variables visible to the payload might not hold the state at all (e.g. `:list` hides the rest list still to be enumerated). Hence, for eager comprehensions only two approaches make sense: Either define that the variables visible are always a copy, or define the effect of assigning to a loop variable as unspecified. The latter approach was chosen for SRFI 42 in the name of efficiency.

Concerning the third semantic question, consider:

```
(let ((n 3))
  (list-ec (:range k n) (begin (set! n 2) k)) )
```

Here, the question is whether `:range` does access the variable `n` for every termination test, or just reads `n` once to set up the loop. Again, different solutions are possible, but the choice becomes easier once it is understood that `n` could be replaced by an arbitrary expression. If `:range` would evaluate its argument expressions repeatedly, this could unintentionally come at a hight price. For this reason, SRFI 42 specifies that the argument expressions of generators are evaluated exactly once: before the loop is set up.

Related to the question what happens if the loop-defining argument is modified is the question what happens if the loop-defining data structure is modified. As there is no way of enforcing anything in Scheme, and copying entire data structure (even if desired) could become costly, the result of modifying a data structure while it is being traversed is better defined unspecified.

3.6 Parallel loops

Often several loops must be executed in lockstep, e.g. counting the lines while reading a file. We will call this "parallel loops," but this does not mean that the processing steps are executed concurrently. Several mechanisms for comprehensions do support such a combination, for example Glasgow Haskell's extension of Haskell98's lazy comprehensions [17], Swindle [7], and SRFI 42 [2].

In the case of lazy comprehensions, parallel generators are relatively straight-forward. Since lazy comprehensions require exactly one type of generator (running through a lazy list), it is sufficient to provide "zipping" two or more lazy lists before enumerating them. In effect, the usefulness of parallel lazy generators is primarily determined by their notation.

Parallel eager generators, on the other hand, are a greater challenge. While the concept of eager comprehensions often allows the user to ignore the details of a loop (i.e. setup, iteration, and termination of the generator), parallel generators can only be constructed by interleaving the different parts of the component generators. Clearly, for this interleaving to be modular it is necessary that every generators is represented by some fixed pattern giving access to the code for setup, iteration, and termination.

In Scheme, the natural solution for this is representing a generator by a procedure computing the next element, and eventually indicating termination. The setup part of a generator constructs the procedure. This approach is used for example in Swindle and for the dispatching generator (`:`) of SRFI 42.

A different approach is to reduce each generator to some fixed "standard loop structure," which provides access to the individual parts of the generator. Then the parts can be combined syntactically for merging two or more component generators into a single parallel generator. This is exactly what the `:parallel` generator of SRFI 42 does, i.e. merging "fully decorated `:do`-loops" (Section 5.2).

3.7 Index variables

A frequent special case of a parallel loop is with an additional index variable, i.e. a variable running through $0, 1, \ldots$ while the elements of another sequence are enumerated. There are two ways of supporting this: by using `:parallel` for combining an unbounded integer counter (with generator `:integer`) with any other generator, and by adding an index variable to the other generator itself.

The first method is universally applicable to any generator, and as such fully modular. The second method provides a more concise notation (important for interactive use), and it can be a little more efficient in case the other generator uses an index anyhow (e.g. `:vector`). SRFI 42 supports both methods.

3.8 Early stopping

An important factor determining the flexibility of a looping construct is a facility for terminating generators or comprehensions early. This is a different mechanism than testing qualifiers (aka guards or filters). The difference is best illustrated by an example.

Consider a predicate for testing if a positive integer is the sum of its proper divisors:

```
(define (perfect? n)
  (= (sum-ec (:range d 1 n)
             (if (= (modulo n d) 0))
             d )
     n ))
```

The `if`-qualifier prevents the inclusion of non-divisors into the sum but it does not stop the `:range`-generator. Now we start investigating perfect numbers:

```
(first-ec #f (: n 1 100) (if (perfect? n)) n)
  ⇒ 6
```

This time the entire comprehension was finished after computing the first perfect number. But assume we need the numbers up to and including the first perfect number:

```
(list-ec (:until (: n 1 100) (perfect? n)) n)
  ⇒ '(1 2 3 4 5 6)
```

In this case the generator `(: n 1 100)` is modified to terminate *after* producing the element for which the additional condition `(perfect? n)` became true. (Note also that the scoping rule of SRFI 42 stated in Section 3.4 dictates that the condition comes *after* the generator in the `:until` expression.) Alternatively, the generator is to terminate *before* producing the element violating an additional condition.

Both forms of early-stopping generators are needed frequently. For example, consider reading a line of text by reading individual characters from a port. Since the last line may or may not have a trailing newline, it is important to append each character read to the string, including newline. This requires the use of `:until`:

```
(define (read-line port)
  (string-ec (:until (:port c port read-char)
                     (char=? c #\newline) )
             c ))
```

(In fact, this was the motivating example for including both `:while` and `:until` in SRFI 42.) The `:while` form of early termination is even more frequent since it derives directly from a precondition of the payload of a comprehension.

Coming back to `first-ec`, the two most useful and frequent early-stopping comprehensions test a predicate on a sequence of values, stopping as soon as a violation is found. These comprehensions, named `any?-ec` and `every?-ec` in SRFI 42, can in fact be derived from `first-ec`.

3.9 Prefix vs. infix syntax

A trivial but highly visible matter is to what extent the syntax makes use of syntactic keywords in infix position (i.e. in a position not being the first after the opening parenthesis). Ultimately, this comes down to personal preference in the form of a compromise between simplicity and similarity with a natural language (which tends to be English). Most designs of comprehensions use an infix operator for the generators (‘`<-`’ is most popular) and possibly more infix operators for other qualifiers and options. This approach has the definitive advantage of reducing the number of parentheses.

In SRFI 42, on the contrary, no infix operators are used at all for the sake of (reducing) mental complexity. A comprehension defining *something* is probably named *something*`-ec`, and a generator defined by an object of type *type* is probably named `:`*type*. All generators are used in the syntax `(:`*type var arg**`)`, where *var* is a variable, optionally followed by an index variable *i* specified as `(index` *i*`)`.

For illustration, Table 3 shows expressions for the same nested loop in different programming languages supporting some form of comprehension. Keep in mind, though, that this is an extremely simple example where the meaning can be guessed at once. For more complicated expressions, infix notation, potentially even with precedences, adds to mental complexity.

4. Concrete designs

In this section we consider existing concrete designs for programming language constructs that enable or simplify (or obfuscate) loops in the Lisp-family of languages. Related constructs for other programming languages are beyond the scope of this article—but with the exception of lazy comprehensions, and loops with genuine parallel semantics as present in Erlang and Occam, they are also not very interesting.

The list does cover some loop-macros from other Lisp dialects, most notably Common Lisp, because these constructs represent serious efforts to provide what is called eager comprehensions in this article. It should be noted, however, that none of the Lisp looping constructs ever came to popularity in the Scheme community, unlike SRFI 42 which surprisingly has gathered quite some friends already in the first few years of its existence. (My earliest sketches date from late 2000; the SRFI got published in the beginning of 2003.)

4.1 `Lambda`, named-`let`, and `do` ($\mathrm{R^5RS}$)

In Scheme the most important construct for writing loops are recursive procedures, often in a tail recursive form. As $\mathrm{R^5RS}$ requires implementations to provide proper tail recursion [1, Section 3.5], recursion also serves as an idiom for iteration. A particularly convenient notation for defining and immediately executing recursive procedures is named-`let` [1, Section 4.2.4]. In addition, Scheme contains the `do`-syntax for defining a single loop, based on explicit state [1, Section 4.2.4].

This design represents a careful choice for including only a few clean and powerful constructs into the language, conforming to the overall minimalistic design philosophy of Scheme. Regrettably, there are two major shortcomings in practice. Firstly, it is already complicated to write the ubiquitous simple loops (refer to the example in the beginning). And secondly, the components of a loop (startup, iteration, termination) are often scattered over large amounts of source code—even if this would be unnecessary. Yet, maybe surprising, no other mechanism for writing loops has achieved considerable acceptance in the Scheme community, leaving the programmer to her own devices.

4.2 “Macros for writing loops” (Kelsey)

The “Macros for writing loops” library [4] is distributed with the Scheme 48 system [3] as the `reduce` package.

It provides the syntactic forms `iterate` and `reduce` implementing the fundamental state based eager comprehension. There are predefined generators running through lists, vectors, strings, integer ranges, reading from a port, and executing a generator procedure (called stream). Other generators can be added fully modularly by defining a hygienic macro in continuation-passing style (CPS) [4, Paragraph “Defining sequence types”]. The comprehensions (`iterate`, `reduce`) define a single, possibly parallel, loop based on explicit state modification.

“Macros for writing loops” is the probably first new loop construct to be proposed for a long time. Moreover, the implementation technique of CPS macros is the key to modularity of comprehensions. In effect, “Macros for writing loops” was most influential to the design of SRFI 42, even though the resulting mechanisms and notations bear little resemblance.

4.3 Swindle (Barzilay)

The Swindle library [7] is a collection of modules extending the PLT Scheme system [5]. It is written for and in PLT. The module “misc.ss” of Swindle contains macros for defining eager comprehensions in the sense of this article.

More precisely, there are predefined comprehensions for side-effect, making a list, numeric summation, numeric products, counting, and general reduction (`collect-of`). Generators are predefined for (integer) ranges, lists, vectors, strings, integers, executing generator procedures, and hash-tables. Swindle allows parallel execution of generators, early termination of comprehensions, has local bindings and side-effects. Generators can be added fully modularly using the generator procedure interface. Swindle makes extensive use of infix notation for expressing generators (e.g. `(n <- 0 .. 10)`, qualifiers, options, and other constructs (infix `and` for parallel execution).

The mechanisms specified in Swindle and for SRFI 42 are very closely related in their principles, but differ considerably in the details. Both acknowledge the need for modularity and well defined scope.

4.4 SRFI 40 “A library of streams” (Berwig)

Although the final form of SRFI 40 [9] does not contain comprehensions anymore, its draft versions did. These comprehensions were of course lazy. During the discussion of SRFI 40, it was decided to split the standard into a lower level part (which became the final SRFI 40) and a higher level part, including lazy comprehensions, which was to become SRFI 41.

The lazy comprehensions of SRFI 40 provided the same benefits as other lazy comprehensions, that is modularity and simplicity. The downside of lazy comprehensions in Scheme is a substantial loss in performance due to the overhead of constructing lazy streams correctly and reliably.

Recall that a lazy stream is something much more sophisticated than a generator procedure (accessing a state hidden in its closure). This implies that lazy comprehensions really require efficient non-strict evaluation, or strictness analysis. While these methods are being used in lazy languages, they are usually not available in Scheme because most programs do not require it.

4.5 Common Lisp

The Common Lisp language [10] contains several constructs for writing loops, and nested eager comprehensions in the sense of this

language	*example*
Haskell	`[k*k | n <- [0..9], k <- [0..n-1]]`
Python	`[k*k for n in range(10) for k in range(n)]`
Ruby	`(0..9).collect {|n| (0..n-1).collect {|k| k*k}}.flatten!`
Erlang	`[K*K || N <- lists:seq(0,9), N >= 1, K <- lists:seq(0,N-1)]`
Mathematica	`Join @@ Table[Table[k*k, {k, 0, n-1}], {n, 0, 9}]`
Magma	`[k*k : k in [0..n-1], n in [0..9]]`
GAP	`Concatenation(List([0..9], n -> List([0..n-1], k -> k*k)))`
PLT, Swindle	`(list-of (* k k) (n <- 0 ..< 10) (k <- 0 ..< n))`
R^5RS, SRFI 42	`(list-ec (: n 10) (: k n) (* k k))`, or with typed generators:
	`(list-ec (:range n 10) (:range k n) (* k k))`
Scheme48, reduce	`(reduce ((count* n 0 10)) ((r '()))`
	`   (reduce ((count* k 0 n)) ((r r))`
	`     (cons (* k k) r) )`
	`   (reverse r) )`

Table 3. Examples of a simple nested loop.

article. These constructs include `do`/`do*`, `dotimes`, `dolist`, and `loop`.

`Do` is essentially the same as in Scheme, apart from the fact that Common Lisp also allows dynamic binding of variables (using `special`). `Do*` is a sequential-binding variant of `do`. `Dotimes` iterates over integer ranges, and `Dolist` over lists; these are rather specialized control structures.

The `loop` facility, on the other hand, could be interpreted as a general programming language in its own right (34 EBNF definitions, [10, Section 6.2 "LOOP"]). It is an extremely flexible mechanism for writing nested and parallel loops, possibly with early stopping, saving intermediate results, goto and labels, and several other features. Since it also supports various forms of accumulation of results, it should be seen as a syntactic form for eager comprehensions. These include comprehensions for making lists, appending, counting, max, min, summation, and general reduction. The syntax is mostly based on infix notation with syntactic keywords for clauses, options, and qualifiers.

The `loop`-syntax is one of the work horses of Common Lisp. It has evolved over a very long time towards higher and higher flexibility, often through the use of infix syntactic keywords. The mental complexity this has produced, however, is a big disadvantage in practice. In effect, the construct does not enjoy large popularity in the Scheme community.

4.6 Other iteration packages for Common Lisp

The "MIT LOOP" [35] is the predecessor of the Common Lisp `loop` facility. The "SLOOP package" (Schelter) [33] is an iteration facility generalizing MIT Lisp's `loop`. The "Yale LOOP Macro" (Ritter and Panagos) [34] is an implementation of the Yale `yloop` macro as described in [37]. All these loop facilities have in common that only the fundamental (side-effect) comprehension is implemented. The syntax is based on syntactic keywords in infix notation and the expressive power varies. Often new types of generators can be added, using the underlying macro facility (procedures as first class citizens did not exist in the language).

The "Series Macro Package" (Waters) [30, 31] implements a concept closely related to lazy comprehensions in the sense of this article. A "series" is essentially a data structure for a lazy list. The package contains operations for producing, processing, and consuming these data structures, or acting on their elements. The implementation is often able to transform the lazy operations into eager evaluation, producing efficient code for frequent loop structures.

The "Lisp comprehensions" (Lapalme) [32] is an adaptation of lazy comprehensions from Miranda into Common Lisp. In this work, Wadler's transformation of lazy list comprehensions [18, Chapter 7] is translated one to one into Lisp in order to mimic the (infix) notation of lazy list comprehensions in Miranda. As the essential conceptual difference between lazy and eager comprehensions is ignored, the resulting mechanism is only of limited usefulness in practice.

4.7 "The anatomy of a loop" (Shivers)

Recently, Shivers defined a new loop mechanism [22, 23, 24] for Scheme (in fact more generally), underpinned by a theory based on the notion of "control dominance." In a nutshell, control dominance is the static property that every access to a variable occurs within an explicit binding construct for that variable. This can be enforced by a type system restricting the control flow graph of the program.

In practice, this concept comes down to the following: all loops are reduced to a primitive loop template consisting of 8 parts, with the control flow graph being made explicit. On top of this resides a programming language very much in the style of a `loop`-macro with predefined generators, guards, and accumulators for the most common data structures. The single outer macro (named `loop`) can be seen as the fundamental eager comprehension, the 8-part loop as the fundamental eager generator (corresponding to `do-ec` and `:do` in SRFI 42).

Since the control flow is made explicit in Shiver's proposal, the looping construct is extremely flexible. However, at present it is not known whether it is also inherently more powerful than the mechanism defined in SRFI 42, or essentially equivalent. This question comes down to whether the fundamental generators (8-part loop vs. `:do`) can be expressed in terms of each other. In addition, it is too early to judge if the additional flexibility is worth the associated mental complexity (8-part loop defined by an explicit control flow graph), and what the impact of the minor design decisions (e.g. infix notation) is on usability. Either way, Shivers' work has potential for further clarifying the true nature of iteration in functional programs.

4.8 SRFI 42 "Eager comprehensions"

The term "eager comprehension" was coined for SRFI 42 [2] in order to make sure the mechanism is never confused with the well-known lazy comprehensions. The reference implementation associated with SRFI 42 is portable under R^5RS with hygienic macros. As the SRFI found some acceptance in the community, implementations are included into several Scheme systems, including PLT [5] and Scheme 48 [3].

The SRFI specifies an extensive set of predefined comprehensions based on what makes sense in R^5RS. Some infrequent comprehensions are left out (e.g. `gcd-ec`), while others have been added for convenience (e.g. `any?-ec`). The predefined typed generators enumerate the standard data structures R^{5}RS. In addition, a dispatching generator ("`:`", read "run through") selects a generator based on the type of arguments given, e.g. the range $\{0, \ldots, n-1\}$ when given an exact integer n. Generators can be run in parallel and terminated early. Other qualifiers include tests (guards), local bindings, and side-effects.

The syntax is based on a simple naming convention and prefix notation without exception. The uniform and simple scoping rule "scope extends to the right until the enclosing comprehension ends" is used (Section 3.4). Generators can be added fully modularly by defining a (hygienic) macro using continuation-passing style (CPS), or by providing a suitable generator procedure. Comprehensions can be added as (hygienic) macros. An introduction to SRFI 42 from the perspective of a user, together with some examples, is provided in the appendix.

5. The implementation of SRFI 42

In this section the overall structure of the reference implementation for eager comprehensions in Scheme is explained. The reader is assumed familiar with the specification as laid down in SRFI 42 [2]. Moreover, it is assumed that the reader is familiar with Scheme's hygienic macro facility [1, Sections 4.3, 5.3, 7.1.5], because it is the primary tool for the reference implementation of SRFI 42.

5.1 A skeleton of eager comprehensions

The following is a simplified but self-contained (R^{5}RS) working skeleton of eager comprehensions:

```
(define-syntax do-ec
  (syntax-rules (if :do)
    ((do-ec q1 q2 r1 r ...)
     (do-ec q1 (do-ec q2 r1 r ...)) )
    ((do-ec (if test) cmd)
     (if test cmd) )
    ((do-ec (:do lbs ne? lss) cmd)
     (do-ec:do cmd (:do lbs ne? lss)) )
    ; call g in CPS, reentry at (*)
    ((do-ec (g arg1 arg ...) cmd)
     (g (do-ec:do cmd) arg1 arg ...) )))

(define-syntax do-ec:do
  (syntax-rules (:do) ; reentry point (*)
    ((do-ec:do cmd (:do (lb ...) ne? (ls ...)))
       (let loop (lb ...)
          (if ne?
              (begin cmd
                     (loop ls ...) ))))))

(define-syntax :do
  (syntax-rules ()
    ((:do (cc ...) lbs ne? lss)
     (cc ... (:do lbs ne? lss)) )))
```

This code defines the primitive eager comprehension `do-ec` and the primitive eager generator `:do`, utilizing a helper macro `do-ec:do` for generating code for `:do`.

Other generators can now be added without modifying the existing macros. E.g. after defining

```
(define-syntax :range
  (syntax-rules ()
    ((:range cc var n)
     (:do cc ((var 0)) (< var n) ((+ var 1))) )))
```

the following comprehension is operational:

```
(do-ec (:range n 5) (:range k n) (display k))
  ⇒ prints: 0010120123
```

The critical issue is the flexibility of the generator `:do` to which all other generators are being reduced. In the skeleton above (refer to `do-ec:do`), the generator `:do` can produce a single named `let` with an arbitrary number of variables (`lb ...`) and a single `if` guarding payload (`cmd`) and next iteration.

5.2 Fully decorated `:do`

In practice, the simple loop structure of the previous section is too restricted. In particular it is not possible to derive the variables visible to the payload from other state variables, to pre-process the arguments, or to terminate after executing the payload. On the other hand, complexity must be kept down.

The particular trade-off chosen for SRFI 42 is based on a fair amount of experimentation. It turned out that the following structure ("fully decorated `:do`") covers most relevant generators:

```
(let (outer-binding ...)
  outer-command ...
  (let loop (loop-binding ...)
    (if not-end-1?
        (let (inner-binding ...)
          inner-command ...
          ≪payload≫
          (if not-end-2?
              (loop loop-step ...) )))))
```

The `:do` generator specifies all variable parts, except for ≪*payload*≫ of course. It allows termination of the loop before or after the payload has been executed. Since many generators do not require "full decoration," a simple transforming optimizer simplifies boolean conditions, eliminates redundant `if` and `let`, and turns `let` without bindings into `begin`.

Note that the use of named-`let` allows iteration by rebinding (Section 3.5), using *loop-binding* and *inner-binding*. Updating by state modification is also possible by storing the iteration state in *outer-binding*, and modifying it using `set!` within `loop`. In fact, `:do` is the only generator in SRFI 42 that allows updating by state modification because no other generator passes the names of this variables in *outer-binding* to its ≪*payload*≫.

The chosen structure for `:do` is powerful enough, and yet still restricted enough, to support the following important constructions on generators:

- Any generator can be modified to terminate early, based on some additional condition, either before (`:while`[6]) or after (`:until`) the payload is executed.
- Two or more `:do`-generators can be merged into a single generator (`:parallel`) enumerating all sequences simultaneously.

For the sake of illustration, here is the complete implementation of the generator `:list` in SRFI 42 running a variable `var` through the concatenation of one or more lists, possibly with an additional index variable `i`.

```
(define-syntax :list
  (syntax-rules (index)
    ((:list cc var (index i) arg ...)
     (:parallel cc (:list var arg ...)
                   (:integers i)) )
    ((:list cc var arg1 arg2 arg ...)
     (:list cc var (append arg1 arg2 arg ...)) )
    ((:list cc var arg)
     (:do cc (let ())
             ((t arg))
             (not (null? t))
```

[6] The implementation is complicated by the fact that the scopes of the variables bound must be preserved while adding the termination condition. This means it is *not* sufficient to add a condition to *not-end-1?*.

```
                  (let ((var (car t))))
                  #t
                  ((cdr t)) ))))
```

The generator `:integers` runs through the infinite sequence of non-negative integers. The expressions supplied to `:do` correspond to the "fully decorated" structure given above, i.e. `(t arg)` is the *loop-binding* and `(var (car t))` is the *inner-binding*.

Note that the multiple-argument case cannot easily be converted into a nested loop because `:do` can only produce a *single* loop; nested loops would prevent generator-merging.

5.3 The dispatching generator

As an alternative to typed generators (`:range`, `:list` etc.) the dispatching generator `:` (read 'run through') of SRFI 42 first evaluates its argument expressions and then dispatches on the type of the values. In other words, `:` is a polymorphic generator. For example, `(list-ec (: x 3) x)` produces `'(0 1 2)` and `(list-ec (: x "abc") x)` produces `'(#\a #\b #\c)`. The purpose of the dispatching generator is making interactive use of comprehensions more convenient.

The implementation of `:` evaluates the arguments and calls a global dispatching procedure. The dispatcher is to construct a generator procedure which is then run to enumerate the sequence. A generator procedure g has a single argument. When called, g either returns the next value of the sequence, or, when the sequence ran out, it returns its argument. In the implementation, the argument given to a generator procedure is `(list #f)`, i.e. an object only `eq?` to itself.

For the sake of modularity, the dispatcher procedure can be retrieved and changed. Moreover, there is a macro producing a generator procedure from a typed generator; this greatly simplifies the definition of dispatching generators.

5.4 Grouping qualifiers with `nested`

In addition to defining new generators in a modular way it is also important to define new comprehensions. While in principle there is no problem (after all every eager comprehension can be reduced to `do-ec`), the fact that there can be an arbitrary number of qualifiers complicates the definition of new comprehensions. In the worst case, a variation of `do-ec` must be provided every time.

A simple trick being used in SRFI 42 keeps the amount of code for a new comprehension low. The syntactic keyword `nested` can be used for grouping an arbitrary number of qualifiers into a single equivalent qualifier understood by `do-ec`. This is illustrated by the definition of a folding comprehension:

```
(define-syntax fold-ec
  (syntax-rules (nested)
    ((fold-ec x0 (nested q1 ...) q r1 r2 r ...)
     (fold-ec x0 (nested q1 ... q) r1 r2 r ...) )
    ((fold-ec x0 q1 q2 r1 r2 r ...)
     (fold-ec x0 (nested q1 q2) r1 r2 r ...) )
    ((fold-ec x0 expr f)
     (fold-ec x0 (nested) expr f) )

    ((fold-ec x0 qualifier expr f)
     (let ((result x0))
       (do-ec qualifier
              (set! result (f expr result)))
       result ))))
```

The last case of the macro implements the functionality for the case that there is exactly one qualifier; the other cases of the macro collect all qualifiers into a single one. Now the list comprehension can be defined as

```
(define-syntax list-ec
  (syntax-rules ()
    ((list-ec r1 r ...)
     (reverse (fold-ec '() r1 r ... cons)) )))
```

Alternatively, the list could be `set-cdr!`'ed together, which may be faster (or not).

5.5 Early-stopping comprehensions

The early-stopping comprehensions of SRFI 42, that is `any?-ec` and `every?-ec`, are reduced to the fundamental early-stopping comprehension `first-ec` with the syntax

(`first-ec` *default qualifier** *expr*) .

This comprehension evaluates the sequence of values specified by the qualifiers, stopping after the first value of *expr*. If the sequence is found empty, the result is *default*.

`Call-with-current-continuation` could be used for a non-local exit, but the reference implementation does not. With an eye on performance it is implemented by introducing an additional stopping variable and modifying each generator to stop once this variable is found true (which is made happen when control reaches *expr*).

6. Performance

The top priority for eager comprehensions is combining convenience and performance. In this section, the performance aspect is investigated more quantitatively.

The Sieve of Eratosthenes As an example we consider computing the primes in $\{2, \ldots, n-1\}$, $n \geq 0$, by the algorithm known as the "Sieve of Eratosthenes." The algorithm (200 BC) ticks off all true multiples of the next not yet ticked off number—and the primes are left over. The following program represents the ticks in a string[7], and uses SRFI 42 for the loops.

```
(define (primes n)
  (let ((p (make-string n #\1)))
    (do-ec (:range k 2 n)
           (if (char=? (string-ref p k) #\1))
           (:range i (* 2 k) n k)
           (string-set! p i #\0) )
    (list-ec (:range k 2 n)
             (if (char=? (string-ref p k) #\1))
             k )))
```

This program is compared with three alternatives:

- The typed generators `:range` are replaced by the dispatching generator `:` of SRFI 42.
- The comprehensions are implemented in Swindle.
- The `do-ec` is replaced by two nested `do`-loops, and the `list-ec` is replaced by a tail-recursive named-`let` constructing the result list.

Figure 1 shows the execution time, divided by n. A number of things can be observed.

Firstly, all four alternatives have reasonable performance and are able to compute the primes below 10^6 in less than $10\,s$. Secondly, only the "DO loop" variant shows the slow increase expected for this $\Theta(n \ln \ln n)$-algorithm. The other curves exhibit lower order terms, probably due to the overhead of setting up a loop—which is most pronounced for the procedure-based variants ("SRFI 42 (:)" and "Swindle").

Linear model of execution time The preceeding example is based on a meaningful algorithm, which is important for a realistic impression. Now we turn to synthetic algorithms with the goal of

[7] A wasteful but practical alternative to arrays of bits, which are absent in Scheme itself and its portable libraries.

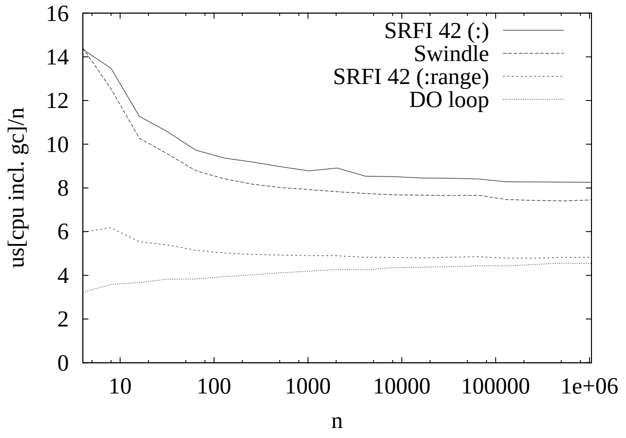

Figure 1. The "Sieve of Eratosthenes." MzScheme 208, Intel Pentium III Mobile, 1 GHz, Win2k.

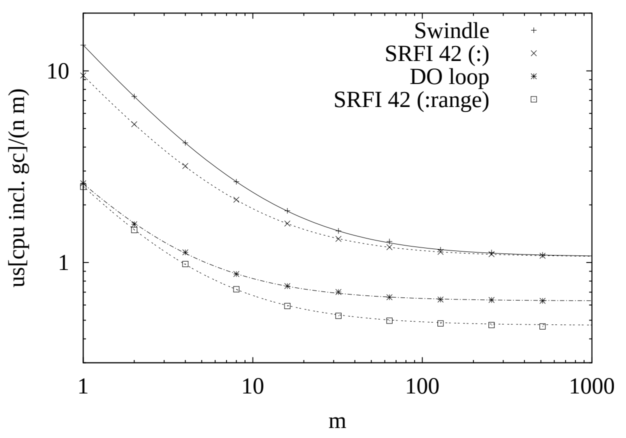

Figure 2. Two nested loops (n times outer, m times inner, $nm = 2^{24}$). MzScheme 208, Intel Pentium III Mobile, 1 GHz, Win2k.

obtaining quantitative information using an abstract model of the execution time.

It is reasonable to assume that the overhead of a loop grows according to a linear model consisting of a fixed startup overhead t_0 and some constant overhead Δt per iteration. The objective is to determine t_0 and Δt from measured execution times. For this we execute different implementations of the following nested loop:

for $k = 1..n$ do for $i = 1..m$ do *payload* od od,

where n and m integer parameters. In order to observe both startup- and iteration-overhead, the number m of inner iterations is varied, while fixing nm for obtaining sufficient total time. The data points in Figure 2 show the result.

Ignoring the time spent on the inner payload, startup- and iteration-overhead can readily be read off the curves as their start and end value. By fitting

$$t(n, m) = (1 + n)t_0 + (n + nm)\Delta t$$

to the data points in Figure 2, slightly more accurate estimates are obtained:

	$t_0/\mu s$	$\Delta t/\mu s$
Swindle	9.99	1.24
SRFI 42 (`:`)	6.59	1.21
DO loop	1.36	1.15
SRFI 42 (`:range`)	1.38	0.60

The curves associated with these parameters are shown in Figure 2.

The particular values obtained here should be taken as an indication, only. They are heavily dependent on the execution model of the underlying Scheme system (interpreted, byte-code, or native). Nevertheless, there is a remarkable gap between the eager comprehensions based on procedures and on direct state modification ("SRFI 42 (`:range`)"). As a rule of thumb, procedures cost a factor of two per iteration and five to ten in startup. We expect this gap to widen for Scheme systems with more sophisticated compilation but did not investigate this quantitatively.

7. Eager comprehensions "lazified"

For what it is worth, eager comprehensions can be turned lazy in a fully modular way. More precisely, it is possible to define the fundamental lazy list comprehension (`stream-ec` that is) in a such way that any *eager* generator can be used with it—without modifying the macros for the generators. Conversely, the eager generator `:stream` enumerates a lazy stream, i.e. runs a variable through the elements. For the streams we use SRFI 40 [9], which provides (even) lazy lists called "streams" as new data structures. With modification, it would also be possible to use simpler odd streams, for example those presented in [36].

The comprehension expression

(`stream-ec` *qualifier** *expr*)

constructs a stream for the sequence that a corresponding `list-ec` would create. The use of `stream-ec` is best explained by example:

```
(define s
  (stream-ec (: x 10) (begin (display x) x)) )

(stream-null? s)
   ⇒ [prints: 0] #f

(stream-null? (stream-cdr s))
   ⇒ [prints: 1] #f

(list-ec (:stream x s) x)
   ⇒ [prints: 23456789] '(0 1 .. 9)
```

In other words, the payload expression `(begin (display x) x)` is to be evaluated on demand, resulting in the digits being printed as shown.

It is an impressive illustration of the powerful mechanisms available in Scheme that `stream-ec` can in fact be defined in a modular way. A possible implementation:

```
(define-syntax stream-ec
  (syntax-rules (nested)
    ((stream-ec (nested q1 ...) q etc1 etc ...)
     (stream-ec (nested q1 ... q) etc1 etc ...) )
    ((stream-ec q1 q2             etc1 etc ...)
     (stream-ec (nested q1 q2)    etc1 etc ...) )
    ((stream-ec expression)
     (stream-ec (nested) expression) )
    ((stream-ec qualifier expression)
     (let ((value         #f)
           (produce-value #f)
           (next-value    #f))
       (define (tail)
         (stream-delay
          (if (call-with-current-continuation
               (lambda (cc)
                 (set! produce-value cc)
                 (next-value #f)
```

```
              #f ))
          (stream-cons value (tail))
          stream-null )))
  (define (make-stream)
    (stream-delay
     (if (call-with-current-continuation
          (lambda (cc)
            (set! produce-value cc)
            (do-ec
              qualifier
              (call-with-current-continuation
               (lambda (cc)
                 (set! next-value cc)
                 (set! value expression)
                 (produce-value #t) )))
            (produce-value #f) ))
          (stream-cons value (tail))
          stream-null )))
  (make-stream) ))))
```

The macro combines all qualifiers into a single one using `nested` (Section 5.4) and uses the fundamental eager comprehension `do-ec` for enumerating the sequence defined by the qualifiers. `Call-with-current-continuation` is used to exit `do-ec` non-locally after producing a value and possibly resuming the very same loop again later.

The eager generator exhausting a stream can be defined as follows:

```
(define-syntax :stream
  (syntax-rules ()
    ((:stream cc var arg)
     (:do cc (let ())
             ((s arg))
             (not (stream-null? s))
             (let ((var (stream-car s))))
             #t
             ((stream-cdr t)) ))))
```

Since `:stream` is just another generator, it can of course be used in `stream-ec`—where it is executed lazily. And since `do-ec` understands guards and local definitions, we have implemented all there is to implement for *lazy* comprehensions in Scheme.

The bad news is `call-with-current-continuation` and the streams of SRFI 40 have a rather high price in terms of time and space consumption in most major Scheme systems. For this reason, the lazy comprehensions defined in this section should not be understood as a serious proposal for a programming language construct—but rather as of great educational and entertaining value. It should be emphasized, though, that lazy comprehensions can be very efficient, provided they are compiled properly.

8. Conclusions

Comprehensions are a particularly concise notation for writing nested and parallel loops with accumulation of results. In the past few years they have come to popularity in many programming languages, including Python and Erlang. When used wisely, comprehensions can improve readability, modularity, and possibly performance.

However, unlike the lazy list comprehensions (ZF expressions) of call-by-need functional languages (like Haskell), a corresponding concept in a call-by-value setting (like Scheme) has substantially different requirements in order to qualify for a generally useful programming construct. SRFI 42 is a specific design aiming at this goal. It is an impressive demonstration of Scheme's own flexibility that that the mechanism specified in SRFI 42 can be implemented naturally without extending the language itself.

Acknowledgements Mike Sperber has provided important input for eager comprehensions in Scheme, in particular he pointed me to the idea of "CPS macros." Without the discussions with Mike, SRFI 42 would probably not exist. Also I would like to thank Phil Berwig, the author of SRFI 40 ('A library of streams') for useful discussions on his lazy comprehensions for Scheme. Probably the biggest source of inspiration for my work presented here were Richard Kelsey's "Macros for writing loops"—even though the casual reader might not suspect this. I would like to thank Philips Research for making this work possible, and in particular my colleagues Philippe Coucaud, Zbigniew Chamski, and Kero van Gelder for their valuable remarks. Finally, I would like to thank the anonymous referees for their corrections and discussion. In particular the third referee brought up the important issue of semantics, and provided an example exposing the 'update by rebind vs. by side-effect' choice.

References

[1] R. Kelsey, W. Clinger, and J. Rees (eds.): Revised^5 Report on the Algorithmic Language Scheme. 20 February 1998. `www.schemers.org/Documents/Standards/R5RS`

[2] S. Egner: SRFI 42 "Eager Comprehensions". Finalized July 7, 2003. `srfi.schemers.org/srfi-42`

[3] R. Kelsey and J. Rees: The Scheme 48 System. `s48.org`

[4] R. Kelsey and J. Rees: "Macros for Writing Loops." The `reduce` library of Scheme 48 [3]. `s48.org/1.2/manual/s48manual_53.html`

[5] The PLT Team: PLT Scheme. `www.plt-scheme.org`

[6] PLT MzScheme. `www.plt-scheme.org/software/mzscheme`

[7] E. Barzilay: The Swindle Library for PLT Scheme [5]. The `collect`-macro of the module "`misc.ss`." `www.cs.cornell.edu/eli/Swindle/misc-doc.html#collect`

[8] E. Hilsdale, D. P. Friedman: Writing Macros in Continuation-Passing Style. Scheme and Functional Programming 2000. September 2000.

[9] P. L. Bewig: SRFI 40 "A Library of Streams." Finalized August 22, 2004. `srfi.schemers.org/srfi-40`

[10] LispWorks Ltd.: The Common Lisp HyperSpec (1996–2005), Chapter 6 "Iteration." `www.lispworks.com/documentation/HyperSpec/Body/06_.htm`

[11] G. van Rossum: Python Reference Manual, Release 2.4.1, 30 March 2005. Section 5.2.4 "List Displays". `www.python.org/doc/2.4.1/ref/lists.html`

[12] Wolfram Research: Mathematica Version 5.0, Documentation of `Table`. `documents.wolfram.com/mathematica/functions/Table`

[13] W. Bosma, J. Cannon: Magma (V2.11, May 2004) Documentation of "Sets" and "Sequences". `magma.maths.usyd.edu.au/magma/htmlhelp/part2.htm`

[14] Ericsson AB: Erlang, Reference Manual (Version 5.4.3). Section 6.22 "List Comprehensions." `www.erlang.se/doc/doc-5.4.3/doc/reference_manual/expressions.html#6.22`

[15] Martin Schönert et. al.: GAP—Groups, Algorithms, and Programming, (Version 3 Release 4 Patchlevel 4) Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[16] S. L. Peyton Jones (ed.): Haskell 98 Language and Libraries, The Revised Report, December 2002. Section 3.11 "List Comprehensions." `www.haskell.org/onlinereport/exps.html`

[17] The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4. Section 7.3.4. "Parallel List Comprehensions." `www.haskell.org/ghc/docs/latest/html/users_guide/syntax-extns.html#parallel-list-comprehensions`

[18] S. L. Peyton Jones: The Implementation of Functional Programming Languages. In particular, Chapter 7 "List Comprehensions" (Philip Wadler). Prentice-Hall, Hemel Hempstead, 1987.

[19] R. B. K. Dewar: The SETL Programming Language. 1979.

[20] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., and Schonberg, E.: Programming with Sets: An Introduction to SETL. Springer-Verlag,

New York, 1986.

[21] R. K. Dybvig: The Scheme Programming Language, 3rd edition. MIT Press 2003. Section 9.3. "A Set Constructor." `www.scheme.com/tspl3/examples.html#./examples:h3`

[22] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." Presentation given at *Daniel P. Friedman: A Celebration* (Bloomington (IN), December 3, 2004). `www.cs.indiana.edu/dfried_celebration.html`

[23] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." Presentation given at Laboratoire d'Informatique de Paris 6 (Paris, January 24, 2005). `www.lip6.fr/fr/liens/organise-fiche.php?theme=5&RECORD_KEY(organise)=id&id(organise)=98`

[24] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." To be published at ICFP 2005, Tallinn, Estonia.

[25] R.M. Burstall: Design Considerations for a Functional Programming Language. Infotech State of the Art Conference: The Software Revolution, Copenhagen, October, 1977.

[26] R. M. Burstall, D. B. MacQueen, and D. T. Sannella: Hope: An Experimental Applicative Language (1980). Conference on LISP and Functional Programming archive Proceedings of the 1980 ACM Conference on LISP and Functional Programming, pp. 136–143, Stanford University, California, United States.

[27] D.A. Turner: The Semantic Elegance of Applicative Languages, in Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture 1981, Portsmouth, New Hampshire, USA.

[28] D.A. Turner: Miranda: A Non-strict Functional Language with Polymorphic Type. Proceedings of a Conference on Functional Programming Languages and Computer Architecture, pp. 1–16, Nancy, France, 1985.

[29] D.A. Turner: An Overview of Miranda. ACM SIGPLAN Notices, Volume 21, Issue 12, December 1986.

[30] R. C. Waters: The Series Macro Package. ACM SIGPLAN Lisp Pointers, Volume III, Issue 1, July 1989.

[31] R. C. Waters: The Series Macro Package for Common Lisp. `series.sourceforge.net`

[32] G. Lapalme: Implementation of a "Lisp Comprehension" Macro. ACM SIGPLAN Lisp Pointers, Volume IV, Issue 2, April 1991.

[33] W. Schelter: The SLOOP Iteration Facility (1985). `www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/sloop/0.html`

[34] F. Ritter, J. Panagos: YLOOP: Portable Implementation of the Yale LOOP Macro (1986). `www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/yloop/0.html`

[35] Massachusetts Institute of Technology: The MIT LOOP Macro (1980, 1986) `www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/mit/mit_loop.cl`

[36] H. Abelson, G. J.Sussman, J. Sussman: Structure and Interpretation of Computer Programs. 2nd ed., MIT Press, Cambridge (MA), 1996.

[37] E. Charniak, C. K. Riesbeck, D. McDermott, and J. R. Meehan: Artificial Intelligence Programming, 2nd ed. Lawrence Erlbaum Associates, 1987.

Appendix: Summary of SRFI 42

For illustration, this appendix contains a brief introduction to SRFI 42 from a user's perspective, together with examples. The actual specification is available at

`http://srfi.schemers.org/srfi-42/srfi-42.html`

In its most simple form, a comprehension according to SRFI 42 looks like this (its value after `=>`):

```
(list-ec (: i 5) (* i i)) => '(0 1 4 9 16).
```

Here, `i` is a local variable sequentially having the values $0, 1, \ldots, 4$, and the squares of these numbers are collected in a list, which is the result. The following example illustrates most conventions of SRFI 42 with respect to nesting and syntax:

```
(list-ec (: n 1 4) (: i n) (list n i))
  => '((1 0) (2 0) (2 1) (3 0) (3 1) (3 2)).
```

In this example, the variable `n` first has value 1 then 2 and finally 3. For each value of `n`, the variable `i` assumes the values $0, 1, \ldots,$ `n` $-$ 1 in turn. The expression `(list n i)` constructs a two-element list for each binding, and the comprehension `list-ec` collects all these results in a list.

Eager comprehensions in the sense of SRFI 42 are just hygienic macros. The basic syntactic form of a comprehension is

`(do-ec` *qualifier** *command*`)`,

i.e. zero or more *qualifier* arguments and a *command* argument. The `do-ec` comprehension enumerates the sequence of binding environments specified by the qualifiers and for each such environment evaluates *command* for side-effects. In a similar fashion, `(sum-ec` *qualifier** *expression*`)` sums the values obtained by evaluating *expression* for the sequence of binding environments specified by *qualifier**. If *qualifier** is empty (i.e. no qualifiers at all) then *expression* is evaluated once. The eager comprehension `list-ec` constructs a list of the values of its expression.

The most common qualifiers are generators. For example, `(:range i 5)` runs variable `i` through $0, 1, \ldots, 4$. The generator `(: i 5)` does the same but uses the type of its argument (i.e. `5`) to decide that it is a range of exact integers that is to be enumerated. In every iteration, `i` is bound to a new location where the integer for that iteration is stored. Other qualifiers are for filtering, e.g. `(if` *condition*`)`, or for side-effect, e.g. `(begin` *command*`)`. The full syntax of SRFI 42 is listed with comments in Table 4.

Checklist for adding comprehensions and generators

The following checklists can if the user wants to add application-specific comprehensions and generators in the style of SRFI 42.

For adding an application-specific comprehension:

1. Use the syntax (*accu*`-ec` ≪*outer*≫ *qualifier** ≪*inner*≫), with ≪*outer*≫ being a fixed list of parameter expressions (e.g. for default values), ≪*inner*≫ being a fixed list inner expressions (usually just *expression*), and *accu* refering to the accumulation process that is being executed.
2. Use the left-to-right scoping rule as much as possible.
3. Avoid syntactic keywords, in particular in infix position.
4. Evaluate parameter expressions exactly once, or at most once if their evaluation is control-flow dependent. Implement this by inserting `let`.
5. Make sure the implementation does not copy macro arguments, because that might lead to exponential growth in code size when nested.

For adding an application-specific typed generator:

expression → *comprehension* | . . .	
comprehension →	
(*ordinary-ec* *qualifier** *expression*)	evaluate *expression* for the sequence of binding environments (or states) specified by the qualifiers
| `(vector-of-length-ec` *k* *qualifier** *expression*`)`	`vector-length` of result known to be *k*
| `(fold-ec` x_0 *qualifier** *expression* f_2`)`	$f_2(x_n, f_2(x_{n-1}, \cdots f_2(x_1, x_0) \cdots))$ for $x_{1..n}$ from *expression*
| `(fold3-ec` x_0 *qualifier** *expression* f_1 f_2`)`	$f_2(x_n, f_2(x_{n-1}, \cdots f_2(x_2, f_1(x_1)) \cdots))$, or x_0 if $n = 0$
| `(do-ec` *qualifier** *command*`)`	evaluate *command* for side-effect
| *application-specific-comprehension*	define using hygienic macro, use checklist
ordinary-ec →	
`list-ec` | `append-ec` | `string-ec` | `string-append-ec`	
| `vector-ec` | `sum-ec` | `product-ec` | `min-ec` | `max-ec`	
| `any?-ec` | `every?-ec` | `first-ec` | `last-ec`	early stopping (aka short evaluation)
qualifier →	
generator	
| `(if` *expression*`)`	insert test (aka guard or filter)
| `(not` *expression*`)` | `(and` *expression**`)` | `(or` *expression**`)`	abbreviate `(if (not` *expression*`))` etc.
| `(begin` *command** *expression*`)`	insert side-effect
| `(nested` *qualifier**`)`	syntactic grouping of qualifiers
generator →	
`(:` *variables* *expression*$^+$`)`	dispatch on type (list,string,vector,integer,real,char,port)
| `(:list` *variables* *expression*$^+$`)`	elements of a (proper) list
| `(:string` *variables* *expression*$^+$`)`	characters of a string
| `(:vector` *variables* *expression*$^+$`)`	elements of a vector
| `(:integers` *variables*`)`	the infinite sequence $0, 1, \ldots$
| `(:range` *variables* *range-limits*`)`	exact integer range
| `(:real-range` *variables* *range-limits*`)`	real (either all exact, or all inexact) range
| `(:char-range` *variables* *min* *max*`)`	character range up to and including *max*
| `(:port` *variables* *expression* [*read*]`)`	*read* defaults to `read`
| `(:dispatched` *variables* *dispatch* *expression*$^+$`)`	calls *dispatch* to construct generator procedure to run
| `(:let` *variables* *expression*`)`	single value sequence (for introducing intermediate variable)
| `(:parallel` *generator**`)`	interleaved execution, until one a generators is exhausted
| `(:while` *generator* *expression*`)`	execute *generator* while *expression* is non-#f
| `(:until` *generator* *expression*`)`	execute *generator* until (and incl.) *expression* is non-#f
| `(:do` [`(let (`*ob**`)` *oc**`)`] `(`*lb**`)` *ne1?* [`(let (`*ib**`)` *ic**`)` *ne2?*] `(`*ls**`))`	loop by named-`let`, possibly decorated
| *application-specific-typed-generator*	define as hygienic macro in CPS, use checklist
range-limits → *stop* | *start stop* | *start stop step*	from *start* (default `0`) to *stop* (excl.) by *step* (default `1`)
variables → *identifier* [`(index` *identifier*`)`]	index variable runs through $0, 1, \ldots$
x_0, f_1, f_2 *min, max, read, dispatch, start, stop, step* → *expression*	

Table 4. Syntax of SRFI 42.

1. Use the syntax `(:`*type* *var* [`(index` *i*`)`] ≪*args*≫`)`, with ≪*args*≫ being the argument expression(s) defining the loop. Here *type* indicates the type of object to enumerate through.
2. Use the syntax `(:`*type* *var*$_1 \cdots$ *var*$_n$ [`(index` *i*`)`] ≪*args*≫`)` if there are always exactly n variables to iterate through.
3. Use the syntax `(:`*type* `(`*var**`)` [`(index` *i*`)`] ≪*args*≫`)` if there is a variable number of variables to iterate through.
4. Use the left-to-right scoping rule as much as possible.
5. Avoid syntactic keywords, in particular in infix position.
6. Make sure argument expressions are evaluated exactly once.
7. Update the iteration state by rebinding, i.e. make sure all variables visible to the payload (*var*, *i*) are bound either in *lb** (loop bindings) or in *ib** (inner bindings).
8. Support multiple arguments if that makes sense, but avoid zero arguments.

Examples

The factorial of a non-negative integer:

```
(define (factorial n)
  (product-ec (:range k 2 (+ n 1)) k) )
```

The sum of the divisors of a positive integer:

```
(define (sigma n)
  (sum-ec (:range d 1 (+ n 1))
          (if (zero? (modulo n d)))
          d ))
```

Pythagorean Triples with entries not exceeding n, i.e. (a, b, c) such that $a^2 + b^2 = c^2$ and integer $1 \leq a \leq b \leq c \leq n$:

```
(define (pythagoras n)
  (list-ec (:let sqr-n (* n n))
           (:range a 1 (+ n 1))
           (:let sqr-a (* a a))
           (:range b a (+ n 1))
           (:let sqr-c (+ sqr-a (* b b)))
           (if (<= sqr-c sqr-n))
           (:range c b (+ n 1))
           (if (= (* c c) sqr-c))
           (list a b c) ))
```

Quicksort with naive choice of pivots (stable):

```
(define (qsort xs)
  (if (null? xs)
      '()
      (let ((pivot (car xs)))
        (append
         (qsort (list-ec (:list x (cdr xs))
                         (if (< x pivot))
                         x ))
         (list pivot)
         (qsort (list-ec (:list x (cdr xs))
                         (if (>= x pivot))
                         x ))))))
```

Approximation of π by Bailey-Borwein-Plouffe's hex-digit extraction formula, i.e. $|(\texttt{pi-BBP}\ m) - \pi| \leq 16^{-m}$ for $m \geq 1$.

```
(define (pi-BBP m)
  (sum-ec (:range n 0 (+ m 1))
          (:let n8 (* n 8))
          (* (- (/ 4 (+ n8 1))
                (+ (/ 2 (+ n8 4))
                   (/ 1 (+ n8 5))
                   (/ 1 (+ n8 6))))
             (/ 1 (expt 16 n)) )))
```

Adding two vectors of equal length (simple program):

```
(define (vector+ x y)
  (vector-ec (:parallel (:vector xi x) (:vector yi y))
             (+ xi yi) ))
```

Adding two vectors of equal length (no intermediate lists):

```
(define (vector+ x y)
  (vector-of-length-ec (vector-length x)
    (:range i (vector-length x))
    (+ (vector-ref x i) (vector-ref y i)) ))
```

Reading a line from an input port, returning all characters read (including newline if present), or returning the eof object:

```
(define (read-line port)
  (let ((line
         (string-ec
          (:until (:port c port read-char)
                  (char=? c #\newline) )
          c )))
    (if (string=? line "")
        (read-char port) ; eof-object
        line )))
```

Reading a file, returning a list of the lines:

```
(define (read-lines filename)
  (call-with-input-file
   filename
   (lambda (port)
     (list-ec (:port line port read-line) line) )))
```