The R^{7}RS-Large Roadmap

Peter McGoron

August 29th, 2026

The R^{7}RS-Large Roadmap

Peter McGoron

August 29th, 2026

1. Hello everyone. I am Peter McGoron, a member of Working Group 2, which is responsible for writing the R^{7}RS-Large.
2. This talk is going to be about the roadmap for the new Report, and what we plan to include in it, above the original R^{7}RS.
3. I'd like to make a disclaimer that this roadmap is subject to change.

What is the R^{7}RS-Large?

- R^{7}RS was released in 2013
- R^{7}RS-Large is an extension of R^{7}RS
- Intended to address the practical needs of software development
- Encompass the R^{6}RS (as much as we can)
- Fix errata and other inconsistencies in the R^{7}RS

What is the R^{7}RS-Large?
- R^{7}RS was released in 2013
- R^{7}RS-Large is an extension of R^{7}RS
- Intended to address the practical needs of software development
- Encompass the R^{6}RS (as much as we can)
- Fix errata and other inconsistencies in the R^{7}RS

1. So just to review — what is the R^{7}RS-Large?
2. It is a downwards compatible version of the R^{7}RS (the small report), intended to address the practical needs of software development.
3. The R^{7}RS-Large is intended to encompass the R^{6}RS to at least some extent.
4. In the process, we will also be collecting and fixing parts of the R^{7}RS that have come up after its ratification.

How is the R^{7}RS-Large organized?

Volume I: Foundations

Volume II: Batteries

Volume III: Environments

How is the R^{7}RS-Large organized?
Volume I: Foundations
Volume II: Batteries
Volume III: Environments

1. In order to accomplish that, we are going to have a very large standard. So we split it up into three volumes: the Foundations, the Batteries, and the Environments.

How is the R^{7}RS-Large organized?

Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.

Volume II: Batteries

Volume III: Environments

How is the R^{7}RS-Large organized?
Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.
Volume II: Batteries
Volume III: Environments

1. The Foundations is, roughly, a superset of the R^{7}RS. It contains new features that cannot be implemented portably in R^{7}RS, such as `syntax-case`, and number vectors, along with some extensions like bitwise operations and a record type system that benefit a lot from being implemented as primitive.

How is the R^{7}RS-Large organized?

Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.

Volume II: Batteries
Things implementable in terms of the Foundations (or R^{7}RS).

Volume III: Environments

How is the R^{7}RS-Large organized?
Volume I: Foundations Things in R^{7}RS, with extensions not implementable in it.
Volume II: Batteries Things implementable in terms of the Foundations (or R^{7}RS).
Volume III: Environments

1. The Batteries are libraries that can be implemented on top of the Foundations, or in many cases, atop R^{7}RS. These are many of the libraries that have already been advertised as being in the R^{7}RS-Large, such as list utility libraries, hash tables, and sorting algorithms.

How is the R^{7}RS-Large organized?

Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.

Volume II: Batteries
Things implementable in terms of the Foundations (or R^{7}RS).

Volume III: Environments
Things specific to the operating environment, such as I/O or network access.

How is the R^{7}RS-Large organized?

Volume I: Foundations Things in R^{7}RS, with extensions not implementable in it.

Volume II: Batteries Things implementable in terms of the Foundations (or R^{7}RS).

Volume III: Environments Things specific to the operating environment, such as I/O or network access.

1. The Environments are libraries that are optional but strongly recommended. They allow one to portably interact with the outside environment. The scope of this volume is file I/O, POSIX APIs, network access, and foreign function interfaces. This volume is the one that we are the most uncertain about what will be inside of it, so there is not much concrete I can say about it at this time.

How is the R^{7}RS-Large organized?

Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.
This talk: 90%

Volume II: Batteries
Things implementable in terms of the Foundations (or R^{7}RS).
This talk: 5%

Volume III: Environments
Things specific to the operating environment, such as I/O or network access.
This talk: 5%

How is the R^{7}RS-Large organized?

Volume I: Foundations
Things in R^{7}RS, with extensions not implementable in it.
This talk: 90%

Volume II: Batteries
Things implementable in terms of the Foundations (or R^{7}RS).
This talk: 5%

Volume III: Environments
Things specific to the operating environment, such as I/O or network access.
This talk: 5%

1. This talk is going to focus mostly on the Foundations. This is because this is where most of the current work is going. I will quickly go over what is in the Batteries right now, and what we plan to put in the Environments. But major work on those will wait until later.

How is the R^{7}RS-Large Foundations organized?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

How is the R^{7}RS-Large Foundations organized?
Split into 7 fascicles:
- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. Since the Foundations is very large, we have split it up into seven fascicles, which will eventually be edited together into a final volume. Each fascicle has a codename that corresponds roughly to what topics that portion will cover.
2. The fascicles cover macros, procedural programming forms, data types, the error system, the library system, continuations, and the record type system. I'm going to go through each one and talk about what's in them, or what we plan to put in each one.

What is in the R^{7}RS-Large? — Macros

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. The first fascicle describes the macro system. Much of it comes from the R^{6}RS, with some innovations from Racket. It was released in October 2024.

What is in the R^{7}RS-Large? — Macros

- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions

What is in the R^{7}RS-Large? — Macros
- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions

1. The macro system of the R^{7}RS-Large will encompass all of the R^{6}RS system. This includes `syntax-case` and identifier macros. This will be combined with R^{7}RS extensions, such as custom ellipses in transformers.

What is in the R^{7}RS-Large? — Macros

- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (`unwrap-syntax`, `quote-syntax`)

What is in the R^{7}RS-Large? — Macros
- Extension of R^{6}RS (syntax-case, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (unwrap-syntax, quote-syntax)

1. One of the criticisms of `syntax-case` was that it was pretty complicated for a "low-level" macro system. To address this, the macro system now includes a minimalistic system based off of the `unwrap-syntax` procedure and the `quote-syntax` primitive. This system is similar to the low-level macro system in the R^{4}RS appendix. They allow one to introduce known identifiers and destructure syntax objects manually.

What is in the R^{7}RS-Large? — Macros

- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (`unwrap-syntax`, `quote-syntax`)
- Syntax parameters

What is in the R^{7}RS-Large? — Macros
- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (`unwrap-syntax`, `quote-syntax`)
- Syntax parameters

1. Syntax parameters are a feature originally in Racket, where one can implement a seemingly unhygienic macro in a hygienic manner, analogous with the dynamic extent of `parameterized` variables.

What is in the R^{7}RS-Large? — Macros

- Extension of R^{6}RS (`syntax-case`, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (`unwrap-syntax`, `quote-syntax`)
- Syntax parameters
- Identifier properties

What is in the R^{7}RS-Large? — Macros
- Extension of R^{6}RS (syntax-case, etc)
- Combined with R^{7}RS extensions
- Procedural macro system (unwrap-syntax, quote-syntax)
- Syntax parameters
- Identifier properties

1. Identifier properties are a feature we have adapted from Chez Scheme, where one can attach expand-time properties to identifiers that can then be inspected in a macro transformer. This can be used to define powerful macros like an extensible pattern matcher.

What is in the R^{7}RS-Large? — unwrap-syntax

```
(define (scar stx)
    (car (unwrap-syntax stx)))

(define (scdr stx)
    (cdr (unwrap-syntax stx)))

(define (scadr stx)
    (scar (scdr stx)))

(define (scddr stx)
    (scdr (scdr stx)))

(identifier? (scadr #'(x y))) => #t
```


What is in the R^{7}RS-Large? — unwrap-syntax

```
(define (scar stx)
    (car (unwrap-syntax stx)))

(define (scdr stx)
    (cdr (unwrap-syntax stx)))

(define (scadr stx)
    (scar (scdr stx)))

(define (scddr stx)
    (scdr (scdr stx)))

(identifier? (scadr #'(x y))) => #t
```

1. Here is an example of the procedural macro system, which is similar to the one described in the appendix of the R^{4}RS.
2. A syntax object is a wrapped syntax object, an identifier, a pair containing syntax objects, or a vector containing syntax objects. Hence, in order to access the underlying object in the syntax object, one has to use `unwrap-syntax`.
3. The procedure `unwrap-syntax` will destructure the syntax object one level while propagating the syntactic context downwards. So a wrapped syntax object pair will be turned into a pair with wrapped syntax objects as its car and cdr. Syntactic context propagation is lazy: a syntax object representing a list may be an improper list which is converted into a proper list by multiple invocations of `unwrap-syntax`.

What is in the R^{7}RS-Large? — unwrap-syntax

```
(define (scar stx)
    (car (unwrap-syntax stx)))

(define (scdr stx)
    (cdr (unwrap-syntax stx)))

(define (scadr stx)
    (scar (scdr stx)))

(define (scddr stx)
    (scdr (scdr stx)))

(identifier? (scadr #'(x y))) => #t
```


What is in the R^{7}RS-Large? — unwrap-syntax

(define (scar stx)
(car (unwrap-syntax stx)))

(define (scdr stx)
(cdr (unwrap-syntax stx)))

(define (scadr stx)
(scar (scdr stx)))

(define (scddr stx)
(scdr (scdr stx)))

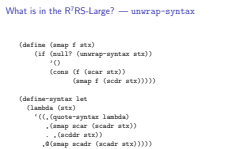
(identifier? (scadr #'(x y))) => #t

1. You can see this in the definitions of `scar` and `scdr`. Each procedure must call `unwrap-syntax` on the input syntax object in order to get the underlying pair, and then call the corresponding list accessor. Note that `unwrap-syntax` will not do anything to an already unwrapped syntax object: an unwrapped pair is returned unchanged. With these procedures, we can then build up analogous list accessors, such as `scadr`, as you can see from the evaluation example below.

What is in the R^{7}RS-Large? — unwrap-syntax

```
(define (smap f stx)
    (if (null? (unwrap-syntax stx))
        '()
        (cons (f (scar stx))
              (smap f (scdr stx)))))

(define-syntax let
  (lambda (stx)
    `((,(quote-syntax lambda)
       ,(smap scar (scadr stx))
       . ,(scddr stx))
      ,@(smap scadr (scadr stx)))))
```

1. Here is an example of `let` in a procedural style. We first need to define the helper function `smap`, which turns out to be very similar to regular `map`. The returned object is still a syntax object, even when it is composed of lists, because a pair containing syntax objects is still a syntax object.
2. The `let` macro itself is written in a style deliberately similar to how one would write such a macro in a system like Common Lisp's `defmacro` or the explicit renaming macro system.
3. The `quote-syntax` form, which is similar to the `syntax` form in the R^{4}RS, inserts an identifier that resolves to the binding of the identifier where the macro is written. In this example, it inserts an identifier that resolves to the `lambda` available to the definition of `let`.

What is in the R^{7}RS-Large? — Syntax parameters

```
(define-syntax-parameter it
  (erroneous-syntax "only usable inside of aif"))

(define-syntax aif
  (syntax-rules ()
    ((_ condition on-true on-false)
     (let ((tmp condition))
       (syntax-parameterize
           ((it (identifier-syntax
                  (_ tmp)
                  ((set! _ value)
                   (set! tmp value)))))
         (if tmp on-true on-false))))))
(aif 10 (* 10 it) 20) => 100
```

(Barzilay, Culpepper, and Flatt. "Keeping it Clean with Syntax Parameters." (2011))

What is in the R^{7}RS-Large? — Syntax parameters
(define-syntax-parameter it
(erroneous-syntax "only usable inside of aif"))
(define-syntax aif
(syntax-rules ()
((_ condition on-true on-false)
(let ((tmp condition))
(syntax-parameterize
((it (identifier-syntax
(_ tmp)
((set! _ value)
(set! tmp value)))))
(if tmp on-true on-false))))))
(aif 10 (* 10 it) 20) => 100
(Barzilay, Culpepper, and Flatt. "Keeping it Clean with Syntax Parameters." (2011))

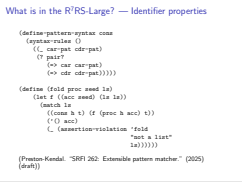
1. The next major feature is syntax parameters, which originate in Racket. Syntax parameters allow one to write seemingly unhygienic macros in a hygienic way. A common unhygienic macro is anaphoric `if`, which is implemented with syntax parameters here.
2. The macro uses `it` as a keyword. The `erroneous-syntax` macro is a transformer that raises a syntax error when the identifier is used if it is not parameterized.
3. This macro also shows off the `identifier-syntax` form from R^{6}RS which is included in the R^{7}RS-Large, which allows us to hide the `tmp` variable behind the syntax parameter.
4. An evaluation sample is shown below, where you can see that the value of the test expression is bound to `it`.

What is in the R^{7}RS-Large? — Identifier properties

```
(define-pattern-syntax cons
  (syntax-rules ()
    ((_ car-pat cdr-pat)
     (? pair?
        (=> car car-pat)
        (=> cdr cdr-pat)))))

(define (fold proc seed ls)
    (let f ((acc seed) (ls ls))
      (match ls
        ((cons h t) (f (proc h acc) t))
        ('() acc)
        (_ (assertion-violation 'fold
                                "not a list"
                                ls))))))
```

(Preston-Kendal. "SRFI 262: Extensible pattern matcher." (2025) (draft))

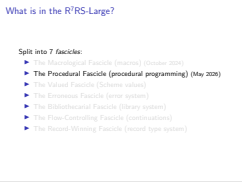


1. Identifier properties are another new feature that allow for the programmer to bind arbitrary data to an identifier. This allows for macros to communicate with each other, and allows for user extension of macros. The example in this slide is from Daphne Preston-Kendal's extensible pattern matcher. One can extend the pattern matcher by adding new patterns to it, in this case the `cons` pattern. After defining the new pattern syntax, one can then use it in the `match` macro below.
2. The `define-pattern-syntax` form is implemented with identifier properties. The extensible pattern matcher is based off of the one in Racket.
3. Beyond this example, one can implement the `syntax-case` pattern matcher using identifier properties. Generally, identifier properties are very useful for any form where identifiers inside of it are given special meanings in other macros.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)



1. Moving on to the second fascicle, which involves procedural programming forms like `lambda`, `if`, and `cond`. It was released in May of this year.

What is in the R^{7}RS-Large? — Procedure forms

- Mixing definitions and expressions

What is in the R^{7}RS-Large? — Procedure forms
- Mixing definitions and expressions

1. The most important feature of this fascicle is the ability to mix definitions and expressions in the bodies, like the body of a `lambda`.

What is in the R^{7}RS-Large? — Procedure forms

- Mixing definitions and expressions
- Aliases (`define-alias`)

What is in the R^{7}RS-Large? — Procedure forms
- Mixing definitions and expressions
- Aliases (`define-alias`)

1. There is now a way to define an alias to an identifier with the `define-alias` form.

What is in the R^{7}RS-Large? — Procedure forms

- Mixing definitions and expressions
- Aliases (`define-alias`)
- More multiple-values forms (`letrec*-values`, `set!-values`)

What is in the R^{7}RS-Large? — Procedure forms
- Mixing definitions and expressions
- Aliases (define-alias)
- More multiple-values forms (letrec*-values, set!-values)

1. More multiple values forms have been added to make all of the binding forms have a consistent single value and multiple value form.

What is in the R^{7}RS-Large? — Procedure forms

- Mixing definitions and expressions
- Aliases (`define-alias`)
- More multiple-values forms (`letrec*-values`, `set!-values`)
- Helpers for higher-order procedures (define higher-order lambda and anonymous recursive expressions)

What is in the R^{7}RS-Large? — Procedure forms
- Mixing definitions and expressions
- Aliases (`define-alias`)
- More multiple-values forms (`letrec*-values`, `set!-values`)
- Helpers for higher-order procedures (define higher-order lambda and anonymous recursive expressions)

1. Some forms that make higher-order procedures easier to write, such as higher-order definitions and anonymous recursive expressions.

What is in the R^{7}RS-Large? — Procedure forms

- Mixing definitions and expressions
- Aliases (`define-alias`)
- More multiple-values forms (`letrec*-values`, `set!-values`)
- Helpers for higher-order procedures (define higher-order lambda and anonymous recursive expressions)
- and more

What is in the R^{7}RS-Large? — Procedure forms
- Mixing definitions and expressions
- Aliases (define-alias)
- More multiple-values forms (letrec*-values, set!-values)
- Helpers for higher-order procedures (define higher-order lambda and anonymous recursive expressions)
- and more

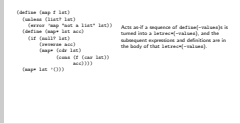
1. And some other changes which are too minor to have their own slide.

What is in the R^{7}RS-Large? — Mixing definitions and expressions

```
(define (map f lst)
  (unless (list? lst)
    (error 'map "not a list" lst))
  (define (map* lst acc)
    (if (null? lst)
        (reverse acc)
        (map* (cdr lst)
              (cons (f (car lst))
                    acc))))
  (map* lst '()))
```

Acts as-if a sequence of `define(-values)`s is turned into a `letrec*(-values)`, and the subsequent expressions and definitions are in the body of that `letrec*(-values)`.

What is in the R^{7}RS-Large? — Mixing definitions and expressions



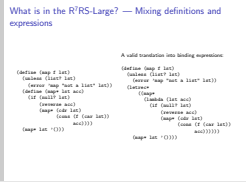
1. So let's start at mixing definitions and expressions. Here is a definition that is now valid in R^{7}RS-Large. The `unless` statement checks the validity of the `lst` argument before going into the actual loop for `map`.
2. In R^{6}RS and R^{7}RS, this is invalid, because the `define` comes after an expression. The R^{7}RS-Large relaxes this, by grouping every sequence of definitions into the equivalent of a `letrec*`.

What is in the R^{7}RS-Large? — Mixing definitions and expressions

```
(define (map f lst)
  (unless (list? lst)
    (error 'map "not a list" lst))
  (define (map* lst acc)
    (if (null? lst)
        (reverse acc)
        (map* (cdr lst)
              (cons (f (car lst))
                    acc))))
  (map* lst '()))
```

A valid translation into binding expressions:

```
(define (map f lst)
  (unless (list? lst)
    (error 'map "not a list" lst))
  (letrec*
      ((map*
        (lambda (lst acc)
          (if (null? lst)
              (reverse acc)
              (map* (cdr lst)
                    (cons (f (car lst))
                          acc))))))
    (map* lst '())))
```

1. Here is an example of a conforming transformation of the left hand side expression. Note that the `unless` statement is not evaluated as a binding in the `letrec*`, like `map*` is. This means that non-definition forms are allowed to return multiple times.

What is in the R^{7}RS-Large? — Mixing definitions and expressions

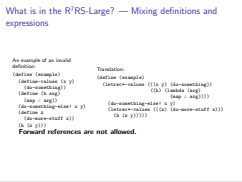
An example of an invalid definition:

```
(define (example)
  (define-values (x y)
    (do-something))
  (define (h arg)
    (map z arg))
  (do-something-else! x y)
  (define z
    (do-more-stuff x))
  (h (z y)))
```

Translation:

```
(define (example)
  (letrec*-values (((x y) (do-something))
                   ((h) (lambda (arg)
                          (map z arg))))
    (do-something-else! x y)
    (letrec*-values (((z) (do-more-stuff x)))
      (h (z y)))))
```

Forward references are not allowed.



1. To illustrate the consequences of this sort of transformation, here is an artificial example. The left hand side is invalid under the transformation in the previous slide. This is because of the forward reference of `z` in the definition of `h`. The first two definition forms are separated from the definition of `z` by the expression `do-something-else!`, and hence they get put in different recursive binding forms.

What is in the R^{7}RS-Large? — Mixing definitions and expressions

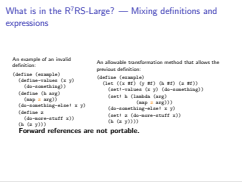
An example of an invalid definition:

```
(define (example)
  (define-values (x y)
    (do-something))
  (define (h arg)
    (map z arg))
  (do-something-else! x y)
  (define z
    (do-more-stuff x))
  (h (z y)))
```

An allowable transformation method that allows the previous definition:

```
(define (example)
  (let ((x #f) (y #f) (h #f) (z #f))
    (set!-values (x y) (do-something))
    (set! h (lambda (arg)
              (map z arg)))
    (do-something-else! x y)
    (set! z (do-more-stuff x))
    (h (z y))))
```

Forward references are not portable.

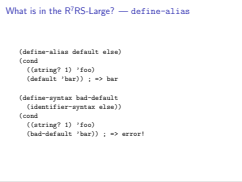


1. Intuitively, a forward reference inside of a `lambda` should work, just like it does at the REPL. The fascicle is written such that an implementation can support these forward declarations. Here is an example of a transformation that allows the forward declaration. It is a direct translation into a `let-and-set!` that still allows for `do-something-else!` to return multiple times.
2. This is allowed, but not required, because this sort of transformation may be difficult for implementations to optimize. This transformation does not allow for an implementation to "fix `letrec`" as easily.

What is in the R^{7}RS-Large? — `define-alias`

```
(define-alias default else)
(cond
  ((string? 1) 'foo)
  (default 'bar)) ; => bar

(define-syntax bad-default
  (identifier-syntax else))
(cond
  ((string? 1) 'foo)
  (bad-default 'bar)) ; => error!
```

1. The new `define-alias` form creates an identifier that is equivalent as a free identifier to another identifier. This means that the identifier points to the same syntactic binding, and is not merely a new binding that takes the old bindings value, or a new macro binding that expands to the other identifier.
2. As you can see in the first two statements on the left, when `default` is aliased to `else`, `default` behaves like `else` in a `cond` clause.
3. However, trying to do the same thing with an identifier macro fails, because the syntax transformer for the `cond` clause does not interpret `bad-default` as `else` until after it has been expanded, and an error results as the auxiliary syntax keyword `else` is being used as a variable.

What is in the R^{7}RS-Large — Anonymous recursive expressions

```
(map (rec (fact n)
       (if (<= n 1) 1
           (* n (fact (- n 1)))))
     '(1 2 3 4 5))
         ; => (1 2 6 24 120)
(car (force (cdr (rec s
                   (cons 1 (delay s))))))
         ; => 1
```


What is in the R^{7}RS-Large — Anonymous recursive expressions

```
(map (rec (fact n)
       (if (<= n 1) 1
           (* n (fact (- n 1)))))
     '(1 2 3 4 5))
         ; => (1 2 6 24 120)
(car (force (cdr (rec s
                   (cons 1 (delay s))))))
         ; => 1
```

1. The R^{7}RS-Large also adds a macro for anonymous recursive expressions: it's a thin wrapper over `letrec`.
2. The most obvious usage is for anonymous recursive procedures, like this factorial function. It can be useful when passed to higher order functions, like in the example here, which calculates the factorial of each number in the list.
3. Being a way to create recursive expressions, it can also construct complicated expressions that hide `lambda`s, like the `delay` expression in the second example. The recursive expression creates an infinite lazy list that is traversed by the `car`s, `cdr`s, and `force` invocations.

What is in the R^{7}RS-Large — Define higher-order `lambda`

```
(define ((addn n) x)
  (+ n x))

(define addn
  (lambda (n)
    (lambda (x)
      (+ n x))))

(map (addn 10) '(1 2 3 4 5))
  ; => (11 12 13 14 15)
```


What is in the R^{7}RS-Large — Define higher-order `lambda`

```
(define ((addn n) x)
  (+ n x))

(define addn
  (lambda (n)
    (lambda (x)
      (+ n x))))

(map (addn 10) '(1 2 3 4 5))
  ; => (11 12 13 14 15)
```

1. The R^{7}RS-Large also standardizes the popular define higher order `lambda` extension, which originates from MIT Scheme and was adopted by many other implementations. It allows one to write procedures that return procedures in a simple way. In the example here, the `addn` procedure returns a procedure that adds `n` to the input argument. It makes using higher-order procedures like `map` easier.

What is in the R^{7}RS-Large — Other extensions

- fixed `when` and `unless`
- aligned `letrec` and `letrec*`
- `(cond)` => unspecified
- `(case x)` => unspecified
- definitions in `when`, `unless`, `cond`, and `case`

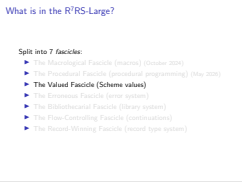
What is in the R^{7}RS-Large — Other extensions
- fixed when and unless
- aligned letrec and letrec*
- (cond) => unspecified
- (case x) => unspecified
- definitions in when, unless, cond, and case

1. The R^{7}RS-Large also does some cleanup work to make some of the forms more consistent, or to clean up issues with the original R^{7}RS.
2. The definition of when and unless in the R^{7}RS required it to tail-call its final expression and to always return a single unspecified value, even if the final expression returned multiple values. This contradicted the sample implementation, and no Scheme implementation I could find always returned a single unspecified value from these forms. The R^{7}RS-Large changes this so that the forms can return any number of unspecified values.
3. The definition of `letrec` did not ban the re-invocation of the continuation of a right-hand side, the way that `letrec*` does. The R^{7}RS-Large aligns both behaviors. Conditional forms with no matching clauses return an unspecified value.
4. Finally, the conditional forms have their sequence forms replaced with body forms. So one can use internal `defines` in the right hand side of a `cond` without wrapping it in `let`.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)



1. The third fascicle is almost done: hopefully it will be released before the end of the year. It will encompass the basic Scheme value types, such as pairs and strings. Much of this material has been discussed and approved by the Working Group, but has not been ratified by the community at large in a ballot.

What is in the R^{7}RS-Large?

Number features

String and bytevector features

What is in the R^{7}RS-Large?
Number features
String and bytevector features

1. This fascicle has lots of small changes. The changes we will focus on are the changes to strings and bytevectors, and the changes to numbers.

What is in the R^{7}RS-Large?

Number features

String and bytevector features

- Full Unicode support
- Raw string literals
- String-notated bytevectors

What is in the R^{7}RS-Large?
Number features
String and bytevector features
- Full Unicode support
- Raw string literals
- String-notated bytevectors

1. The R^{7}RS-Large will include the full Unicode support that the R^{6}RS required, and also adds some new reader syntax that makes it easier to write some strings and bytevectors without escapes.

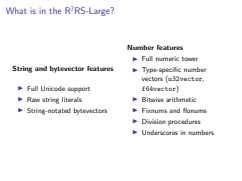
What is in the R^{7}RS-Large?

String and bytevector features

- Full Unicode support
- Raw string literals
- String-notated bytevectors

Number features

- Full numeric tower
- Type-specific number vectors (`u32vector`, `f64vector`)
- Bitwise arithmetic
- Fixnums and flonums
- Division procedures
- Underscores in numbers



1. The R^{7}RS-Large will include the full numeric tower of the R^{6}RS, type specific number vectors such as `u32vector`, an expanded version of the fixnum and flonum procedures from the R^{6}RS, a large set of integer division procedures, and an extension to support underscores in numbers.

What is in the R^{7}RS-Large? — Unicode

- Implementations must support all and only the Unicode scalar values as characters and in strings
  - R^{7}RS only requires a subset of ASCII
  - R^{7}RS allows for characters with values above `#x10FFFF`
  - Excludes surrogate codepoints

What is in the R^{7}RS-Large? — Unicode
- Implementations must support all and only the Unicode scalar values as characters and in strings
  - R^{7}RS only requires a subset of ASCII
  - R^{7}RS allows for characters with values above #x10FFFF
  - Excludes surrogate codepoints

1. Implementations must support all of and only the Unicode scalar values, which are the non-surrogate codepoints. The R^{7}RS only required implementations to support the non-NUL ASCII characters and also allowed non-Unicode characters by allowing characters to have numerical values above any Unicode codepoint. These features were seldom used, so support for them was dropped in the R^{7}RS-Large.
2. This also means that surrogate codepoints are not characters. This is the case in the R^{6}RS. Surrogate codepoints will have subtly different behavior on different implementations, as some use them for escaping. In addition, requiring lone surrogate handling would complicate implementing the R^{7}RS-Large on systems that use UTF-16.

What is in the R^{7}RS-Large? — Unicode

- Implementations must support all and only the Unicode scalar values as characters and in strings
  - R^{7}RS only requires a subset of ASCII
  - R^{7}RS allows for characters with values above `#x10FFFF`
  - Excludes surrogate codepoints
- Implementations must case-fold according to the default Unicode case-folding rules
  - `(string-foldcase "StRaße") => "strasse"`
- Implementations will have to allow Unicode in identifiers
  - `(define √2 (sqrt 2))`

What is in the R^{7}RS-Large? — Unicode
- Implementations must support all and only the Unicode scalar values as characters and in strings
  - R^{7}RS only requires a subset of ASCII
  - R^{7}RS allows for characters with values above #x10FFFF
  - Excludes surrogate codepoints
- Implementations must case-fold according to the default Unicode case-folding rules
  - (string-foldcase "StRaße") => "strasse"
- Implementations will have to allow Unicode in identifiers
  - (define √2 (sqrt 2))

1. Another addition, carried over from the R^{6}RS, is that implementations must implement the default Unicode case-folding rules and implement Unicode for identifiers in a specified way that was specified in the R^{6}RS and was optional in the R^{7}RS. As you can see here, this definition, which is valid in R^{6}RS, is now valid in R^{7}RS-Large.

What is in the R^{7}RS-Large? — Raw String Literals

- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)

What is in the R^{7}RS-Large? — Raw String Literals
- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)

1. Another extension we have introduced is raw string literals. Raw string literals allow one to write strings that contain a lot of double quotes and backslashes without escaping them. They are useful when embedding another syntax inside of a string, such as JSON or SQL. They are a popular extension in other languages, like C++, Python, and Rust.

What is in the R^{7}RS-Large? — Raw String Literals

- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)
- Syntax: #" *delim* " *text* " *delim*"

What is in the R^{7}RS-Large? — Raw String Literals
- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)
- Syntax: #" *delim* " *text* " *delim*"

1. The syntax of a raw string literal starts with a hash followed by the delimiter sequence. The delimiter sequence is zero or more characters surrounded by double quotes. Then after the delimiter sequence is the string data interpreted without escapes, until the delimiter sequence is read again.

What is in the R^{7}RS-Large? — Raw String Literals

- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)
- Syntax: `#"` *delim* `"` *text* `"` *delim*`"`
- Examples:
```
#""This is my "string\n"."
 => "This is my \"string\\\n\"."
#"delim"#""string""delim"  => "#\"\"string\"\""
```
- **An arbitrary string can be embedded as-is with a suitable choice of delimiter.**

What is in the R^{7}RS-Large? — Raw String Literals
- Write strings with backslashes and double quotes inside of them without escapes
- Popular extension in other languages (C++, Python, Rust)
- Syntax: #" delim " text " delim"
- Examples: #""This is my "string\n"."
  => "This is my \"string\\\n\"."
  #"delim"#""string""delim" => "#\"\"string\"\""
- An arbitrary string can be embedded as-is with a suitable choice of delimiter.

1. Here are some examples to demonstrate. The first one has the smallest delimiter sequence, which is two double quotes. Hence the appearance of two double quotes terminates the raw string. The double quotes and the backslashes that do not terminate the string are kept as-is.
2. The second example shows a raw string with a custom delimiter. This allows us to write a raw string in a raw string without escaping. The moral is, raw strings allow us to include a string verbatim in the source code of a program, given a suitable choice for a delimiter.

What is in the R^{7}RS-Large? — String-notated bytevectors

- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components

What is in the R^{7}RS-Large? — String-notated bytevectors
- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components

1. The R^{7}RS-Large also adds string-notated bytevectors. These are useful when one wants to write a bytevector that includes human-readable sections in ASCII.

What is in the R^{7}RS-Large? — String-notated bytevectors

- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: `#u8"` *string bytevector elements* `"`
  - *string bytevector element* is either an ASCII character or an octet

What is in the R^{7}RS-Large? — String-notated bytevectors
- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: #u8" *string bytevector elements* "
  - *string bytevector element* is either an ASCII character or an octet

1. The syntax for string-notated bytevectors is very similar to strings. The only difference is that they start with `#u8` instead of just a double quote. Inside of them, the `\x` escape sequence means octet, not Unicode codepoint.

What is in the R^{7}RS-Large? — String-notated bytevectors

- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: `#u8"` *string bytevector elements* `"`
  - *string bytevector element* is either an ASCII character or an octet
- ELF header: `#u8"\x7F;ELF"`
  - Equivalent to `#u8(#x7F #x45 #x4C #x46)`
- PNG header: `#u8"\x89;PNG\r\n\x1A;\n"`
  - `#u8(#x89 #x50 #x4E #x47 #x0D #x0A #x1A #x0A)`

What is in the R^{7}RS-Large? — String-notated bytevectors
- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: #u8" string bytevector elements "
  - string bytevector element is either an ASCII character or an octet
- ELF header: #u8"\x7F;ELF"
  - Equivalent to #u8(#x7F #x45 #x4C #x46)
- PNG header: #u8"\x89;PNG\r\n\x1A;\n"
  - #u8(#x89 #x50 #x4E #x47 #x0D #x0A #x1A #x0A)

1. Here are some examples of string-notated bytevectors. The first one is the ELF header. Note the use of `\x7F;` to mean the octet `\x7F;`. In this case, the octet and the UTF-8 encoded character U+7F coincide, but this is not the case in general. You can see this in the next example, the PNG header. Here the `\x89;` inserts the literal byte `\x89;`. Escapes that correspond to ASCII escapes are allowed, so `\n` is the ASCII newline.

What is in the R^{7}RS-Large? — String-notated bytevectors

- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: `#u8"` *string bytevector elements* `"`
  - *string bytevector element* is either an ASCII character or an octet
- ELF header: `#u8"\x7F;ELF"`
  - Equivalent to `#u8(#x7F #x45 #x4C #x46)`
- PNG header: `#u8"\x89;PNG\r\n\x1A;\n"`
  - `#u8(#x89 #x50 #x4E #x47 #x0D #x0A #x1A #x0A)`
- R^{7}RS-Large does not allow Unicode characters in string-notated bytevectors

What is in the R^{7}RS-Large? — String-notated bytevectors
- Write bytevectors like strings
- Useful for dealing with binary data that includes ASCII components
- Syntax: #u8" string bytevector elements "
  - string bytevector element is either an ASCII character or an octet
- ELF header: #u8"\x7F;ELF"
  - Equivalent to #u8(#x7F #x45 #x4C #x46)
- PNG header: #u8"\x89;PNG\r\n\x1A;\n"
  - #u8(#x89 #x50 #x4E #x47 #x0D #x0A #x1A #x0A)
- R7RS-Large does not allow Unicode characters in string-notated bytevectors

1. In general, the R^{7}RS-Large does not allow Unicode characters in string-notated bytevectors.

What is in the R^{7}RS-Large? — Full numeric tower

- Arbitrarily large exact integers and rationals
- Exact and inexact complex numbers
- Signed inexact zero, infinities, and NaNs
- `(atan +0.0+1.0i)` => $\pi/4$`+inf.0i`
- `(atan -0.0+1.0i)` => $-\pi/4$`+inf.0i`

What is in the R^{7}RS-Large? — Full numeric tower
- Arbitrarily large exact integers and rationals
- Exact and inexact complex numbers
- Signed inexact zero, infinities, and NaNs
- `(atan +0.0+1.0i)` => $\pi/4$`+inf.0i`
- `(atan -0.0+1.0i)` => $-\pi/4$`+inf.0i`

1. The R^{6}RS numeric tower is adopted in full. This means that bignums, arbitrarily large exact rationals, and exact and inexact complex numbers must be supported. In addition, optional features like signed zero, infinities, and NaNs are now required. This allows code to be more precise and more portable.
2. This means that the behavior of functions sensitive to the sign of zero and that return infinities will now behave more portably across implementations, such as complex `atan` here.

What is in the R^{7}RS-Large? — Type-specific number vectors

- Optimized vectors for storing certain types of numbers

What is in the R^{7}RS-Large? — Type-specific number vectors
- Optimized vectors for storing certain types of numbers

1. The R^{7}RS-Large also adds type specific number vectors. These could be implemented atop bytevectors in the R^{6}RS, but the dedicated type specific vectors are a popular extension.

What is in the R^{7}RS-Large? — Type-specific number vectors

- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- `#s32(1 -2 3 -4)`

What is in the R^{7}RS-Large? — Type-specific number vectors
- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- #s32(1 -2 3 -4)

1. The integer number vectors come in signed and unsigned varieties, and are sized to contain 8, 16, 32, and 64 bit numbers. The 8-bit unsigned number vector is specified to be the same as a bytevector. Here is an example of a signed 32-bit number vector.

What is in the R^{7}RS-Large? — Type-specific number vectors

- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- `#s32(1 -2 3 -4)`
- binary32 and binary64 floating-point numbers
- `#f64(1.0 2.0)`

What is in the R^{7}RS-Large? — Type-specific number vectors
- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- #s32(1 -2 3 -4)
- binary32 and binary64 floating-point numbers
- #f64(1.0 2.0)

1. The inexact real number vectors come in binary32 and binary64 variants. The R^{7}RS-Large is slightly stricter than the SRFIs, as we now mandate the type stored in the number vector. Here is an example of a binary64 number vector.

What is in the R^{7}RS-Large? — Type-specific number vectors

- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- `#s32(1 -2 3 -4)`
- binary32 and binary64 floating-point numbers
- `#f64(1.0 2.0)`
- inexact complex numbers comprised of binary32 and binary64 numbers respectively
- `#c128(1.0+2.0i 3.0+4.0i)`

1. There are also complex number vectors, comprised of binary32 and binary64 numbers respectively.

What is in the R^{7}RS-Large? — Type-specific number vectors
- Optimized vectors for storing certain types of numbers
- two's-complement and unsigned 8, 16, 32, and 64 bit integers
- #s32(1 -2 3 -4)
- binary32 and binary64 floating-point numbers
- #f64(1.0 2.0)
- inexact complex numbers comprised of binary32 and binary64 numbers respectively
- #c128(1.0+2.0i 3.0+4.0i)

What is in the R^{7}RS-Large? — Bitwise arithmetic

- In the Foundations, a core set of bitwise operators for infinite bitstrings
  - `bitwise-not`, `bitwise-and`, `bitwise-ior`, `bitwise-xor`, `arithmetic-shift`, `integer-length`, and `bit-count`
- In the whole R^{7}RS-Large, a superset of the R^{6}RS procedures

What is in the R^{7}RS-Large? — Bitwise arithmetic
- In the Foundations, a core set of bitwise operators for infinite bitstrings
  - bitwise-not, bitwise-and, bitwise-ior, bitwise-xor, arithmetic-shift, integer-length, and bit-count
- In the whole R^{7}RS-Large, a superset of the R^{6}RS procedures

1. The R^{7}RS-Large will include a facility for bitwise arithmetic over bignums. The Foundations will only include a core set of operators that are sufficient to implement the rest in an efficient way. The whole report will include a superset of the R^{6}RS procedures.

What is in the R^{7}RS-Large? — Flonums

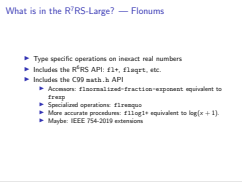
- Type specific operations on inexact real numbers
- Includes the R^{6}RS API: `fl+`, `flsqrt`, etc.

What is in the R^{7}RS-Large? — Flonums
- Type specific operations on inexact real numbers
- Includes the R^{6}RS API: `fl+`, `flsqrt`, etc.

1. The R^{7}RS-Large will include a large superset of the R^{6}RS's flonum library. Included will be equivalents to many of the generic arithmetic procedures, like square root, exponential and logarithm.

What is in the R^{7}RS-Large? — Flonums

- Type specific operations on inexact real numbers
- Includes the R^{6}RS API: `fl+`, `flsqrt`, etc.
- Includes the C99 `math.h` API
  - Accessors: `flnormalized-fraction-exponent` equivalent to `frexp`
  - Specialized operations: `flremquo`
  - More accurate procedures: `fllog1+` equivalent to $\log(x+1)$.
  - Maybe: IEEE 754-2019 extensions



1. The flonum library will also include functions from the C99 `math.h` library. These include flonum accessors, such as an equivalent to `frexp`, specialized operations such as `flremquo`, and procedures that can compute certain operations more accurately, such as $\log(x+1)$. We might look into including more IEEE 754-2019 recommended operations, which appear in C23.

What is in the R^{7}RS-Large? — Division procedures

- There are many ways to do integer division, depending on the desired properties on the quotient/remainder
- Given numerator n and denominator d, compute integers q and r such that $n = qd + r$ and $|r| < |d|$
- R^{7}RS-Large will include multiple integer division procedures
  - `floor/` (in R^{7}RS)
  - `truncate/` (in R^{7}RS)
  - `euclidean/` (similar to R^{6}RS `div-and-mod`)
  - `balanced/` (similar to R^{6}RS `div0-and-mod0`)
  - `ceiling/`
  - `round/`

What is in the R^{7}RS-Large? — Division procedures
- There are many ways to do integer division, depending on the desired properties on the quotient/remainder
- Given numerator n and denominator d, compute integers q and r such that $n = qd + r$ and $|r| < |d|$
- R^{7}RS-Large will include multiple integer division procedures
  - `floor/` (in R^{7}RS)
  - `truncate/` (in R^{7}RS)
  - `euclidean/` (similar to R^{6}RS `div-and-mod`)
  - `balanced/` (similar to R^{6}RS `div0-and-mod0`)
  - `ceiling/`
  - `round/`

1. In many languages there is only one integer division pair. The R^{7}RS-Large will include a very large selection of integer division operators.
2. This is the complete list of the integer division operators. Two were included in the R^{7}RS, and two were included in the R^{6}RS. The R^{7}RS operators are based off of two integer rounding procedures, and the R^{6}RS procedures are based off of two constraints on the remainder. The other procedures, `ceiling/` and `round/`, implement the pairs based off of the other IEEE 754 integer rounding procedures.

What is in the R^{7}RS-Large? — Underscores in numbers

- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: 1'000'000
  - Python: 10_00_00
- The R^{7}RS-Large allows an underscore between digits.

What is in the R^{7}RS-Large? — Underscores in numbers
- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: 1'000'000
  - Python: 10_00_00
- The R^{7}RS-Large allows an underscore between digits.

1. To round off the major new features for numbers is the addition of underscores in numbers. Underscores in numbers have no effect on the value of the number: they make the number easier for humans to read. Such a feature is a very common extension, as seen in C++ and Python.

What is in the R^{7}RS-Large? — Underscores in numbers

- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: 1'000'000
  - Python: 10_00_00
- The R^{7}RS-Large allows an underscore between digits.
  - `#b10110000_11011101`
  - `#x7FFF_FFFF_FFFF_FFFF`

What is in the R^{7}RS-Large? — Underscores in numbers
- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: 1'000'000
  - Python: 10_00_00
- The R^{7}RS-Large allows an underscore between digits.
  - #b10110000_11011101
  - #x7FFF_FFFF_FFFF_FFFF

1. Here are some examples of the new syntax. As you can see, it's easier to tell that the first number is 16 bits, and that the second number is 64 bits.

What is in the R^{7}RS-Large? — Underscores in numbers

- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: `1'000'000`
  - Python: `10_00_00`
- The R^{7}RS-Large allows an underscore between digits.
  - `#b10110000_11011101`
  - `#x7FFF_FFFF_FFFF_FFFF`
- Only one underscore is portable: implementations may extend the syntax to more locations.

What is in the R^{7}RS-Large? — Underscores in numbers
- Being able to add digit separators to numbers is a common and useful feature in other languages
  - C++: 1'000'000
  - Python: 10_00_00
- The R^{7}RS-Large allows an underscore between digits.
  - #b10110000_11011101
  - #x7FFF_FFFF_FFFF_FFFF
- Only one underscore is portable: implementations may extend the syntax to more locations.

1. Only one underscore between two digits is portable, although implementations may allow more underscores. This is because the Scheme number syntax is complicated, and doing things like prefixing or suffixing something with underscores may be intended by the programmer to be an identifier instead of a number.

What is in the R^{7}RS-Large? — Much more

- Fixnums (superset of R^{6}RS library)
- Hyperbolic and inverse hyperbolic functions
- `conjugate` procedure
- Extensions to many procedures (`utf8->string`, `assoc`, etc.)

What is in the R^{7}RS-Large? — Much more
- Fixnums (superset of R^{6}RS library)
- Hyperbolic and inverse hyperbolic functions
- `conjugate` procedure
- Extensions to many procedures (`utf8->string`, `assoc`, etc.)

1. There are many comparatively small things that we have added to the R^{7}RS-Large in this fascicle that did not merit their own slide. However, we think that adding them will be a benefit for the language.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)

The following are in the planning stages:

- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

What is in the R^{7}RS-Large?
Split into 7 *fascicles*:
- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)

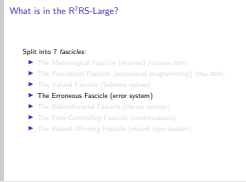
The following are in the planning stages:
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. The final four fascicles are in the early planning stages. However, the bulk of the procedures were defined in the third fascicle, so hopefully drafting these should be simpler. As such, I will have less to say about them.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)



1. The fourth fascicle will describe the error raising requirements and the condition system.

What is in the R^{7}RS-Large? — Error system

- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - `with-exception-handler`, `raise`, `guard`
  - "it is an error" (undefined behavior)

What is in the R^{7}RS-Large? — Error system
- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - with-exception-handler, raise, guard
  - "it is an error" (undefined behavior)

1. The R^{7}RS had a very small exception handling and error raising system. The exception handling system is a subset of the R^{6}RS's system.
2. The R^{7}RS specifies that "it is an error" when the language is used incorrectly, such as trying to take the cdr of the empty list. This jargon is basically undefined behavior: an implementation can take any action whatsoever.

What is in the R^{7}RS-Large? — Error system

- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - `with-exception-handler`, `raise`, `guard`
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - `(cdr '()) =>` *exception*
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...

What is in the R^{7}RS-Large? — Error system
- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - with-exception-handler, raise, guard
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - (cdr '()) => exception
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...

1. In practice, errorful situations don't cause segmentation faults or memory corruption. Most implementations raise exceptions and implement some sort of way of inspecting those exceptions.

What is in the R^{7}RS-Large? — Error system

- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - `with-exception-handler`, `raise`, `guard`
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - `(cdr '()) =>` *exception*
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions

What is in the R^{7}RS-Large? — Error system
- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - with-exception-handler, raise, guard
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - (cdr '()) => exception
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions

1. The R^{6}RS has a complete exception object system, with a standard set of exception types that one can subclass. In the R^{6}RS, most misuses of the language raise "assertion violations."

What is in the R^{7}RS-Large? — Error system

- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - `with-exception-handler`, `raise`, `guard`
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - `(cdr '()) =>` *exception*
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions
- **What condition system to use?**
  - The R^{6}RS condition system, likely

What is in the R^{7}RS-Large? — Error system
- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - with-exception-handler, raise, guard
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - (cdr '()) => exception
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions
- **What condition system to use?**
  - The R^{6}RS condition system, likely

1. So what condition object system will we use for the R^{7}RS-Large? We will likely use the R^{6}RS condition hierarchy. However, we don't know how much of the R^{6}RS condition object system we will use.

What is in the R^{7}RS-Large? — Error system

- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - `with-exception-handler`, `raise`, `guard`
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - `(cdr '()) =>` *exception*
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions
- **What condition system to use?**
  - The R^{6}RS condition system, likely
- **What error raising behavior to use?**
  - R^{6}RS style? (No extensions allowed)
  - Extensions allowed with restrictions?
- We will look to the community for guidance.

What is in the R^{7}RS-Large? — Error system
- The R^{7}RS has an exception system that is a subset of R^{6}RS
  - with-exception-handler, raise, guard
  - "it is an error" (undefined behavior)
- Most R^{7}RS implementations raise exceptions on type errors
  - (cdr '()) => exception
  - CHICKEN 6, Chibi Scheme, Gauche, Gambit, MIT-Scheme, ...
- The R^{6}RS specifies exception object system, and condition hierarchy
  - Most errors raise exceptions
- What condition system to use?
  - The R^{6}RS condition system, likely
- What error raising behavior to use?
  - R^{6}RS style? (No extensions allowed)
  - Extensions allowed with restrictions?
- We will look to the community for guidance.

1. What are the requirements for raising exceptions? Should we take the R^{6}RS position of no extensions to the standard procedures, or should we be more flexible? We will try to poll people in the community to figure out the answer.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. The fifth fascicle will describe libraries and programs.

What is in the R^{7}RS-Large? — Library syntax

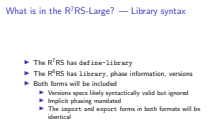
- The R^{7}RS has `define-library`
- The R^{6}RS has `library`, phase information, versions

What is in the R^{7}RS-Large? — Library syntax
- The R^{7}RS has define-library
- The R^{6}RS has library, phase information, versions

1. The R^{6}RS and R^{7}RS have different forms for describing the contents of a library. The R^{7}RS `define-library` has provisions for conditional imports and including text files, while the R^{6}RS `library` form has provisions for import phases and versions. The R^{7}RS-Large will have to deal with this.

What is in the R^{7}RS-Large? — Library syntax

- The R^{7}RS has `define-library`
- The R^{6}RS has `library`, phase information, versions
- Both forms will be included
  - Versions specs likely syntactically valid but ignored
  - Implicit phasing mandated
  - The `import` and `export` forms in both formats will be identical



1. What we plan to do is to offer both syntactic forms. Library version specs will likely be syntactically valid but ignored, and we will mandate an implicit phasing model which means that phase information will also be syntactically valid but ignored.

What is in the R^{7}RS-Large? — Top-level expansion order

The R^{6}RS and R^{7}RS have different expansion orders.

What is in the R^{7}RS-Large? — Top-level expansion order
The R^{6}RS and R^{7}RS have different expansion orders.

1. One major issue we will have to reconcile at this point is the different expansion orders of the R^{6}RS and R^{7}RS.

What is in the R^{7}RS-Large? — Top-level expansion order

The R^{6}RS and R^{7}RS have different expansion orders.

R^{6}RS: Mutual expansion

```
(define (f x) (macro 10 x))
(define-syntax macro
  (syntax-rules ()
    ((_ a b) (+ a b))))
(display (f 100)) ; "110"
```

This program is *invalid* in R^{7}RS, because `macro` is used before definition.

What is in the R^{7}RS-Large? — Top-level expansion order
The R^{6}RS and R^{7}RS have different expansion orders.
R^{6}RS: Mutual expansion
(define (f x) (macro 10 x))
(define-syntax macro
(syntax-rules ()
((_ a b) (+ a b))))
(display (f 100)) ; "110"
This program is invalid in R^{7}RS, because macro is used before definition.

1. The R^{6}RS has what I will call a mutual expansion order. Macros defined later on in a top-level program are usable earlier on in the program. In the example on the screen, the orange text denotes a macro use that appears before the macro definition later on in blue. Note that this program acts differently than if one input the text of the program into a REPL.

What is in the R^{7}RS-Large? — Top-level expansion order

The R^{6}RS and R^{7}RS have different expansion orders.

R^{6}RS: Mutual expansion

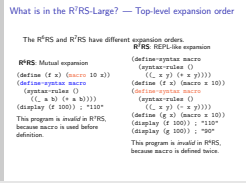
```
(define (f x) (macro 10 x))
(define-syntax macro
  (syntax-rules ()
    ((_ a b) (+ a b))))
(display (f 100)) ; "110"
```

This program is *invalid* in R^{7}RS, because `macro` is used before definition.

R^{7}RS: REPL-like expansion

```
(define-syntax macro
  (syntax-rules ()
    ((_ x y) (+ x y))))
(define (f x) (macro x 10))
(define-syntax macro
  (syntax-rules ()
    ((_ x y) (- x y))))
(define (g x) (macro x 10))
(display (f 100)) ; "110"
(display (g 100)) ; "90"
```

This program is *invalid* in R^{6}RS, because `macro` is defined twice.



1. On the other hand, R^{7}RS has what I call a REPL-like expansion order. Definitions are executed sequentially, and new definitions with the same name overwrite previous definitions. Here, the first definition of `macro` is used in the definition of the procedure `f`, and the second definition of `macro` (in orange) is used in the definition of the procedure `g`. This is invalid in the R^{6}RS, because the same identifier is defined twice.
2. We don't have a solution for any of this yet.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

What is in the R^{7}RS-Large?
Split into 7 *fascicles*:
- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. Moving on to the next fascicle, which will cover continuations and continuation-related matters.

Delimited and composable continuations

- Delimited continuations capture part of a computation, not the whole computation (`call/cc`)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile

Delimited and composable continuations
- Delimited continuations capture part of a computation, not the whole computation (call/cc)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile

1. Scheme has had reified continuations in the form of `call/cc` for a long time, but over time people have developed the concept of delimited continuations. Delimited continuations capture part of a computation, not the whole computation like `call/cc` does. Delimited continuations as a primitive operation are included in more Scheme implementations: Racket, Gauche, Sagittarius, and Guile.

Delimited and composable continuations

- Delimited continuations capture part of a computation, not the whole computation (`call/cc`)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile
- Racket-inspired proposal for Scheme: SRFI 226 (followed by Gauche and Sagittarius)

Delimited and composable continuations
- Delimited continuations capture part of a computation, not the whole computation (call/cc)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile
- Racket-inspired proposal for Scheme: SRFI 226 (followed by Gauche and Sagittarius)

1. For R7RS-Large, Marc Nieper-Wißkirchen has written a very comprehensive proposal, SRFI 226. Sagittarius implements the entire proposal, and Gauche is in the process of implementing it.

Delimited and composable continuations

- Delimited continuations capture part of a computation, not the whole computation (`call/cc`)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile
- Racket-inspired proposal for Scheme: SRFI 226 (followed by Gauche and Sagittarius)
  - Prompts: `call-with-continuation-prompt`, `abort-current-continuation`
  - Composable continuations: `call-with-composable-continuation`
  - Continuation barriers: `call-with-continuation-barrier`
  - Continuation marks: `with-continuation-marks`
  - Integration with threads, interrupts, promises, and fluids

Delimited and composable continuations
- Delimited continuations capture part of a computation, not the whole computation (call/cc)
- Delimited continuations are popular
  - Gauche, Sagittarius, Racket, and Guile
- Racket-inspired proposal for Scheme: SRFI 226 (followed by Gauche and Sagittarius)
  - Prompts: call-with-continuation-prompt, abort-current-continuation
  - Composable continuations: call-with-composable-continuation
  - Continuation barriers: call-with-continuation-barrier
  - Continuation marks: with-continuation-marks
  - Integration with threads, interrupts, promises, and fluids

1. The proposal covers prompts, composable continuations, continuation barriers, continuation marks, and integration of these things with threads, interrupts, promises, and fluids.

Delimited and composable continuations — Example

Implementation of `shift` and `reset`.

(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

Delimited and composable continuations — Example
Implementation of shift and reset.
(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

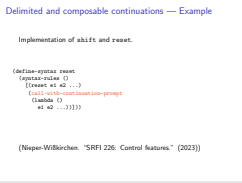
1. There is a lot here that has been covered in other places in more detail, so I will not go into many details here. However, I will go over an example of how the common operators `shift` and reset are implemented in terms of SRFI 226 continuations.

Delimited and composable continuations — Example

Implementation of `shift` and `reset`.

```
(define-syntax reset
  (syntax-rules ()
    [(reset e1 e2 ...)
     (call-with-continuation-prompt
      (lambda ()
        e1 e2 ...))]))
```

(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))



1. `Reset` is implemented by installing a continuation prompt. Prompts have tags that identify the prompt and an abort handler. The default tag is a fixed value that is returned by default-continuation-prompt-tag. The default abort handler re-instates the prompt and then calls a thunk.

Delimited and composable continuations — Example

Implementation of `shift` and `reset`.

```
(define-syntax reset
  (syntax-rules ()
    [(reset e1 e2 ...)
     (call-with-continuation-prompt
      (lambda ()
        e1 e2 ...))]))
```

```
(define-syntax shift
  (syntax-rules ()
    [(shift k e1 e2 ...)
     (call-with-composable-continuation
      (lambda (c)
        (define (k . args)
          (reset (apply c args)))
        (abort-current-continuation
         (default-continuation-prompt-tag)
         (lambda ()
           e1 e2 ...))))]))
```

(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

Delimited and composable continuations — Example
Implementation of shift and reset.
(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

1. `Shift` is implemented by capturing a composable continuation up to, but not including a given prompt tag. The default prompt tag is the same tag as the one installed by `reset`. The composable continuation here is called `c`.

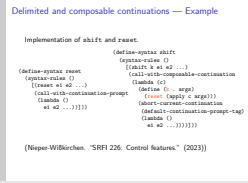
Delimited and composable continuations — Example

Implementation of `shift` and `reset`.

```
(define-syntax reset
  (syntax-rules ()
    [(reset e1 e2 ...)
     (call-with-continuation-prompt
      (lambda ()
        e1 e2 ...))]))
```

```
(define-syntax shift
  (syntax-rules ()
    [(shift k e1 e2 ...)
     (call-with-composable-continuation
      (lambda (c)
        (define (k . args)
          (reset (apply c args)))
        (abort-current-continuation
         (default-continuation-prompt-tag)
         (lambda ()
           e1 e2 ...))))]))
```

(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))



1. We want the continuation to have the prompt available when we call it, so we make a wrapper `k` that wraps the captured composable continuation `c` with a `reset` to accomplish that.

Delimited and composable continuations — Example

Implementation of `shift` and `reset`.

```
(define-syntax reset
  (syntax-rules ()
    [(reset e1 e2 ...)
     (call-with-continuation-prompt
      (lambda ()
        e1 e2 ...))]))
```

```
(define-syntax shift
  (syntax-rules ()
    [(shift k e1 e2 ...)
     (call-with-composable-continuation
      (lambda (c)
        (define (k . args)
          (reset (apply c args)))
        (abort-current-continuation
         (default-continuation-prompt-tag)
         (lambda ()
           e1 e2 ...))))]))
```

(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

Delimited and composable continuations — Example
Implementation of shift and reset.
(Nieper-Wißkirchen. "SRFI 226: Control features." (2023))

1. Then we abort the continuation to the prompt that we instated using `reset`. The default abort handler will re-instate the continuation, and then execute the thunk we passed it, which contains the body of the `shift` statement.

Multithreading

- Many implementations have preemptive multithreading
  - Chibi, Capy, Chez, CHICKEN, Cyclone, Gambit and more
- Specification: SRFI 18
  - `make-thread`, `thread-specific`, `thread-start!`, `thread-yield!`, `thread-sleep!`, `mutex-lock!`, ...

Multithreading
- Many implementations have preemptive multithreading
  - Chibi, Capy, Chez, CHICKEN, Cyclone, Gambit and more
- Specification: SRFI 18
  - make-thread, thread-specific, thread-start!, thread-yield!, thread-sleep!, mutex-lock!, ...

1. Another thing we want to introduce at this stage is a standard multithreading library. Multithreading is a part of multiple popular implementations, and there is already a mature specification for such a thing written by Marc Feeley all the way back in 2001.
2. The proposal covers many things that one expects in a multithreading system, like mutexes and thread-local storage. So I won't go over it here.

What is in the R^{7}RS-Large?

Split into 7 *fascicles*:

- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

What is in the R^{7}RS-Large?
Split into 7 *fascicles*:
- The Macrological Fascicle (macros) (October 2024)
- The Procedural Fascicle (procedural programming) (May 2026)
- The Valued Fascicle (Scheme values)
- The Erroneous Fascicle (error system)
- The Bibliothecarial Fascicle (library system)
- The Flow-Controlling Fascicle (continuations)
- The Record-Winning Fascicle (record type system)

1. Moving on to the the final fascicle, which is the fascicle on record types.

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity

1. The R^{7}RS has a minimalistic record-type system, useful for defining disjoint types. The R^{6}RS in contrast has a very featureful system that was criticized for its complexity and some of its design choices, particularly regarding the interaction of the syntactic and procedural layers.

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?

1. So what do we do to reconcile this?

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)

1. Should we use a simplified but compatible form of the R^{6}RS record system?

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)

1. Should we use a non-compatible record system that has some of the features?

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)
  - Just keep R^{7}RS's `define-record-type` as-is?

1. Just keep SRFI 9 as-is?

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)
  - Just keep R^{7}RS's `define-record-type` as-is?

The record-type system

- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)
  - Just keep R^{7}RS's `define-record-type` as-is?
  - "The only record I expect [the Record-Winning Fascicle] to win is for how heated the discussion will be." — Daphne Preston-Kendal

The record-type system
- The R^{7}RS has SRFI 9 `define-record-type`, a simple system for defining disjoint types
- The R^{6}RS included a featureful record type system (single inheritance, procedural interface, inspection) that was criticized for its complexity
- What do we do here?
  - A simplified version of the R^{6}RS record system? (SRFI 237 + 240)
  - A non-compatible interface with a syntactic, procedural, and inspection interface? (SRFI 99/136/150)
  - Just keep R^{7}RS's `define-record-type` as-is?
  - "The only record I expect [the Record-Winning Fascicle] to win is for how heated the discussion will be." — Daphne Preston-Kendal

1. Here's a great quote from Daphne Preston-Kendal: "The only record I expect this to win is for how heated the discussion will be." People care a lot about this, and we don't have a definitive answer on what to do here yet.

SRFI 237: R^{6}RS Records (refined)

- Previous R^{6}RS code will work with this proposal
- `define-record-type` can be made backwards-compatible with R^{7}RS (SRFI 240)

Nieper-Wißkirchen. "SRFI 237: R^{6}RS Records (refined)." (2023)

SRFI 237: R^{6}RS Records (refined)
- Previous R^{6}RS code will work with this proposal
- `define-record-type` can be made backwards-compatible with R^{7}RS (SRFI 240)

Nieper-Wißkirchen. "SRFI 237: R^{6}RS Records (refined)." (2023)

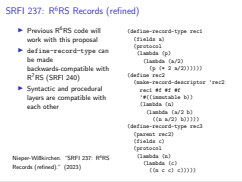
1. I will quickly go over the two major proposals that go beyond the R^{7}RS. The first is Marc Nieper-Wißkirchen's proposal that is backwards-compatible with the R^{6}RS record system and the R^{7}RS record system.

SRFI 237: R^{6}RS Records (refined)

- Previous R^{6}RS code will work with this proposal
- `define-record-type` can be made backwards-compatible with R^{7}RS (SRFI 240)
- Syntactic and procedural layers are compatible with each other

```
(define-record-type rec1
  (fields a)
  (protocol
   (lambda (p)
     (lambda (a/2)
       (p (* 2 a/2))))))
(define rec2
  (make-record-descriptor 'rec2
    rec1 #f #f #f
    '#((immutable b))
    (lambda (n)
      (lambda (a/2 b)
        ((n a/2) b)))))
(define-record-type rec3
  (parent rec2)
  (fields c)
  (protocol
   (lambda (n)
     (lambda (c)
       ((n c c) c)))))
```

Nieper-Wißkirchen. "SRFI 237: R^{6}RS Records (refined)." (2023)



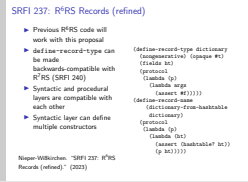
1. This proposal makes the syntactic and procedural layers compatible with each other: the fact that they were not was a criticism of the R^{6}RS. On the right half, you can see that `rec1` is defined syntactically, and `rec2` is defined with the procedural interface and inherits from `rec1`. `Rec3` inherits from `rec2` using the syntactic interface.

SRFI 237: R^{6}RS Records (refined)

- Previous R^{6}RS code will work with this proposal
- `define-record-type` can be made backwards-compatible with R^{7}RS (SRFI 240)
- Syntactic and procedural layers are compatible with each other
- Syntactic layer can define multiple constructors

```
(define-record-type dictionary
  (nongenerative) (opaque #t)
  (fields ht)
  (protocol
   (lambda (p)
     (lambda args
       (assert #f)))))
(define-record-name
    (dictionary-from-hashtable
     dictionary)
  (protocol
   (lambda (p)
     (lambda (ht)
       (assert (hashtable? ht))
       (p ht)))))
```

Nieper-Wißkirchen. "SRFI 237: R^{6}RS Records (refined)." (2023)



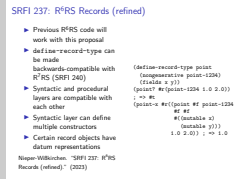
1. This proposal also makes the syntactic and procedural layers equal by allowing the syntactic layer to define multiple constructors for a single record type. Here you can see the `dictionary` record type is defined so that one cannot construct it directly, but it can be constructed with the `dictionary-from-hashtable` constructor. Here `dictionary` acts like a virtual class.

SRFI 237: R^{6}RS Records (refined)

- Previous R^{6}RS code will work with this proposal
- `define-record-type` can be made backwards-compatible with R^{7}RS (SRFI 240)
- Syntactic and procedural layers are compatible with each other
- Syntactic layer can define multiple constructors
- Certain record objects have datum representations

```
(define-record-type point
  (nongenerative point-1234)
  (fields x y))
(point? #r(point-1234 1.0 2.0))
; => #t
(point-x #r((point #f point-1234
              #f #f
              #((mutable x)
                (mutable y)))
            1.0 2.0)) ; => 1.0
```

Nieper-Wißkirchen. "SRFI 237: R^{6}RS Records (refined)." (2023)

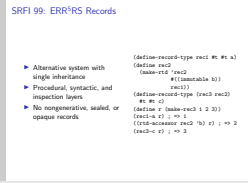


1. The proposal also adds datum representations for non-generative, non-opaque records whose fields have Scheme data in them. This allows for record types to be written and read. Displayed are two different representations of a record object, where one is more detailed than the other.

SRFI 99: ERR^{5}RS Records

- Alternative system with single inheritance
- Procedural, syntactic, and inspection layers
- No nongenerative, sealed, or opaque records

```
(define-record-type rec1 #t #t a)
(define rec2
  (make-rtd 'rec2
            #((immutable b))
            rec1))
(define-record-type (rec3 rec2)
  #t #t c)
(define r (make-rec3 1 2 3))
(rec1-a r) ; => 1
((rtd-accessor rec2 'b) r) ; => 2
(rec3-c r) ; => 3
```

1. An alternative proposal is SRFI 99, written by Will Clinger. It is a single-inheritance record type system that has a procedural, syntactic, and inspection layer, like the R^{6}RS.
2. An example is shown on the right which is similar to the first example on the previous slide. The first definition makes a record with no parent using the syntactic interface. The second creates a record type inheriting from the first using the procedural interface, and the third has a record type defined with the syntactic interface inheriting from the procedural interface.
3. Note that this system does not have nongenerative, sealed, or opaque records, and it does not have custom constructors like the R^{6}RS.

Other record-related matters

1. Outside of the big-ticket items, there are also some matters that are relevant to containers of Scheme objects that we will deal with here.

Other record-related matters

- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety

Other record-related matters
- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety

1. One of these is tagged procedures. These are procedures that have data associated with them, where that data can be accessed without invoking the procedure itself. This is useful when using procedures as a part of an object system, where you want to know if you can communicate with a procedure by passing a message before actually passing that message.

Other record-related matters

- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety
- Custom ports
  - R^{6}RS-style ports, or something more advanced?

Other record-related matters
- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety
- Custom ports
  - R^{6}RS-style ports, or something more advanced?

1. We will also consider custom ports. Custom ports already exist in the R^{6}RS, although there are some ideas on extending this system. The R^{6}RS custom port system is unable to implement the string and bytevector ports of the R^{7}RS, for example. It would be nice if the R^{7}RS-Large's custom port system could handle that.

Other record-related matters

- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety
- Custom ports
  - R^{6}RS-style ports, or something more advanced?
- Storage management (SRFI 254)
  - Ephemerons (holders for weak values)
  - Guardians (for cleanup of external resources)
  - Transport cell guardians (for implementing e.g. eq?-based hash tables)

Other record-related matters
- Tagged procedures
  - MIT Scheme, CHICKEN, Gauche, Chibi, ...
  - SRFI 259: Tagged procedures with type safety
- Custom ports
  - R^{6}RS-style ports, or something more advanced?
- Storage management (SRFI 254)
  - Ephemerons (holders for weak values)
  - Guardians (for cleanup of external resources)
  - Transport cell guardians (for implementing e.g. eq?-based hash tables)

1. We will also consider some systems related to garbage collection, such as supporting weakly held values and finalization. One can read the proposal and the literature cited in it for more information about these.

Volume II: Batteries

1. OK, that's all of the Foundations. It is quite a bit. Next I will cover the "Batteries," which is the much more well known section of the R^{7}RS-Large. This consists of things that can be portably implemented in terms of the Foundations, and in many cases, in terms of the R^{6}RS or R^{7}RS.

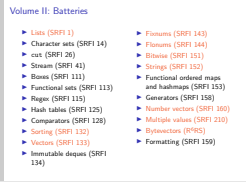
Volume II: Batteries

- Lists (SRFI 1)
- Character sets (SRFI 14)
- `cut` (SRFI 26)
- Stream (SRFI 41)
- Boxes (SRFI 111)
- Functional sets (SRFI 113)
- Regex (SRFI 115)
- Hash tables (SRFI 125)
- Comparators (SRFI 128)
- Sorting (SRFI 132)
- Vectors (SRFI 133)
- Immutable deques (SRFI 134)
- Fixnums (SRFI 143)
- Flonums (SRFI 144)
- Bitwise (SRFI 151)
- Strings (SRFI 152)
- Functional ordered maps and hashmaps (SRFI 153)
- Generators (SRFI 158)
- Number vectors (SRFI 160)
- Multiple values (SRFI 210)
- Bytevectors (R^6RS)
- Formatting (SRFI 159)

1. Here is the list of libraries that are currently voted into the R^{7}RS-Large. They are derived either from finalized SRFIs, or from other Scheme standards.

Volume II: Batteries

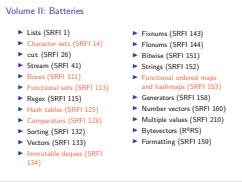
- Lists (SRFI 1)
- Character sets (SRFI 14)
- `cut` (SRFI 26)
- Stream (SRFI 41)
- Boxes (SRFI 111)
- Functional sets (SRFI 113)
- Regex (SRFI 115)
- Hash tables (SRFI 125)
- Comparators (SRFI 128)
- Sorting (SRFI 132)
- Vectors (SRFI 133)
- Immutable deques (SRFI 134)
- Fixnums (SRFI 143)
- Flonums (SRFI 144)
- Bitwise (SRFI 151)
- Strings (SRFI 152)
- Functional ordered maps and hashmaps (SRFI 153)
- Generators (SRFI 158)
- Number vectors (SRFI 160)
- Multiple values (SRFI 210)
- Bytevectors (R^6RS)
- Formatting (SRFI 159)



1. This includes libraries that add operations to the ones in the Foundations, such as libraries for lists, vectors, strings, and new types such as number vectors and flonums,

Volume II: Batteries

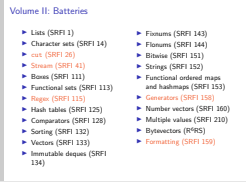
- Lists (SRFI 1)
- Character sets (SRFI 14)
- `cut` (SRFI 26)
- Stream (SRFI 41)
- Boxes (SRFI 111)
- Functional sets (SRFI 113)
- Regex (SRFI 115)
- Hash tables (SRFI 125)
- Comparators (SRFI 128)
- Sorting (SRFI 132)
- Vectors (SRFI 133)
- Immutable deques (SRFI 134)
- Fixnums (SRFI 143)
- Flonums (SRFI 144)
- Bitwise (SRFI 151)
- Strings (SRFI 152)
- Functional ordered maps and hashmaps (SRFI 153)
- Generators (SRFI 158)
- Number vectors (SRFI 160)
- Multiple values (SRFI 210)
- Bytevectors (R^{6}RS)
- Formatting (SRFI 159)



1. common data structures such as hash tables and persistent maps,

Volume II: Batteries

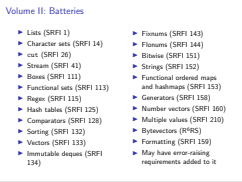
- Lists (SRFI 1)
- Character sets (SRFI 14)
- cut (SRFI 26)
- Stream (SRFI 41)
- Boxes (SRFI 111)
- Functional sets (SRFI 113)
- Regex (SRFI 115)
- Hash tables (SRFI 125)
- Comparators (SRFI 128)
- Sorting (SRFI 132)
- Vectors (SRFI 133)
- Immutable deques (SRFI 134)
- Fixnums (SRFI 143)
- Flonums (SRFI 144)
- Bitwise (SRFI 151)
- Strings (SRFI 152)
- Functional ordered maps and hashmaps (SRFI 153)
- Generators (SRFI 158)
- Number vectors (SRFI 160)
- Multiple values (SRFI 210)
- Bytevectors (R^{6}RS)
- Formatting (SRFI 159)



1. and other libraries, such as lazy streams and regular expressions.

Volume II: Batteries

- Lists (SRFI 1)
- Character sets (SRFI 14)
- `cut` (SRFI 26)
- Stream (SRFI 41)
- Boxes (SRFI 111)
- Functional sets (SRFI 113)
- Regex (SRFI 115)
- Hash tables (SRFI 125)
- Comparators (SRFI 128)
- Sorting (SRFI 132)
- Vectors (SRFI 133)
- Immutable deques (SRFI 134)
- Fixnums (SRFI 143)
- Flonums (SRFI 144)
- Bitwise (SRFI 151)
- Strings (SRFI 152)
- Functional ordered maps and hashmaps (SRFI 153)
- Generators (SRFI 158)
- Number vectors (SRFI 160)
- Multiple values (SRFI 210)
- Bytevectors (R^{6}RS)
- Formatting (SRFI 159)
- May have error-raising requirements added to it



1. These proposals are mostly written with the R^{7}RS in mind. When we decide on the exception and condition system of the R^{7}RS-Large, we may extend requirements to raise exceptions on type errors and other errors on the procedures in the Batteries.

Batteries: What else?

1. So what other things are we thinking of adding to the Batteries?

Batteries: What else?

- Bitvectors (SRFI 178)
- Date and time manipulation
- Multidimensional arrays (SRFI 231, 268)
- Random number generation (SRFI 27, 194, 271)
- JSON (SRFI 180)
- Integer sets and maps (SRFI 217, 224)
- Pipeline operators (SRFI 197)
- Flexvectors (SRFI 214)
- Argument parsing (SRFI 37)
- A loop macro
- A pattern matcher
- R^{6}RS compatibility libraries
- more?

Batteries: What else?
- Bitvectors (SRFI 178)
- Date and time manipulation
- Multidimensional arrays (SRFI 231, 268)
- Random number generation (SRFI 27, 194, 271)
- JSON (SRFI 180)
- Integer sets and maps (SRFI 217, 224)
- Pipeline operators (SRFI 197)
- Flexvectors (SRFI 214)
- Argument parsing (SRFI 37)
- A loop macro
- A pattern matcher
- R^{6}RS compatibility libraries
- more?

1. Here is a list of things, some with SRFIs, and some without, that might make it in. Some of these are more likely than others.

Volume III: Environments

1. The final volume of the R^{7}RS-Large will be the Environments, a set of optional standard libraries for interacting with the operating environment.

Volume III: Environments

- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)

Volume III: Environments
- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)

1. The environments will contain the procedures for opening files, closing files, and operating on the process that existed in the R^6RS and R^7RS.

Volume III: Environments

- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)

Volume III: Environments
- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)

1. Another thing we might want to add is a standard interface to the POSIX API. Then we can extend our file I/O operations to operations on the filesystem, the terminal, pipes, and everything one could need (on POSIX).

Volume III: Environments

- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)
- C FFI
  - Prior art: `pffi` and `foreign-c`

Volume III: Environments
- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)
- C FFI
  - Prior art: `pffi` and `foreign-c`

1. Another good thing to have would be a standard C FFI. As it turns out, figuring out a good set of FFI operations for implementations to support is difficult. For example, some implementations currently do not support calling Scheme callbacks from C code. There have been some attempts at making a standard FFI, such as Takahashi Kato's R^6RS `pffi`, and Retropikzel's `foreign-c` for R^7RS.

Volume III: Environments

- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)
- C FFI
  - Prior art: `pffi` and `foreign-c`
- Sockets, TLS, HTTP, and the modern networking kitchen sink

Volume III: Environments
- File I/O (R^6RS and R^7RS libraries)
- Basic filesystem operations (`delete-file`, `file-exists?`)
- Process context (`command-line`, `exit`)
- POSIX API (filesystem, termios, pipes, process control, signals, etc...)
- C FFI
  - Prior art: `pffi` and `foreign-c`
- Sockets, TLS, HTTP, and the modern networking kitchen sink

1. And a final thing to consider for all of this business is the wild world of networking, including sockets, TLS, and more. These are very important for modern programs that involve big servers and clients talking to those servers.

Conclusion

- Volume I (Foundations): Required features, not portably implementable
  - Macro system
  - Extended lexical syntax
  - Delimited continuations
  - Record type system
- Volume II (Batteries): Required features, portably implementable in terms of Volume I
  - Utilities for data types
  - Data structures and containers
- Volume III (Environments): Optional features, not portably implementable
  - Foreign function interface
  - Operating system interface

Conclusion
- Volume I (Foundations): Required features, not portably implementable
  - Macro system
  - Extended lexical syntax
  - Delimited continuations
  - Record type system
- Volume II (Batteries): Required features, portably implementable in terms of Volume I
  - Utilities for data types
  - Data structures and containers
- Volume III (Environments): Optional features, not portably implementable
  - Foreign function interface
  - Operating system interface

1. That covers the R^{7}RS-Large, as we have currently planned it out. We are starting with required and non-portable features, before integrating the required and portable features, and then specifying the optional and non-portable features.
2. The first volume will encompass macro systems, more lexical syntax, delimited continuations, and the record type system.
3. The second volume will encompass utilities and data structures defined for the things in the first volume.
4. The third volume will encompass interfaces to outside systems, like foreign function interfaces and the operating system.

Acknowledgements

Working Group 2

- Alaric Snell-Pym
- Arthur A. Gleckler
- Jani Juhani Sinervo
- John Cowan
- Marc Nieper-Wißkirchen
- Peter McGoron
- Sudarshan S Chawathe
- Vincent Manis
- Wolfgang Corcoran-Mathe
- Zhu Zihao
- "Identity"

And the contributors to many SRFIs:

- Alex Shinn
- Daphne Preston-Kendal
- Marc Feeley
- Olin Shivers
- Philip Bewig
- Sebastian Egner
- Shiro Kawai
- Taylor Campbell
- Will Clinger
- ... and many more!

Acknowledgements
Working Group 2
Alaric Snell-Pym
Arthur A. Gleckler
Jani Juhani Sinervo
John Cowan
Marc Nieper-Wißkirchen
Peter McGoron
Sudarshan S Chawathe
Vincent Manis
Wolfgang Corcoran-Mathe
Zhu Zihao
"Identity"
And the contributors to many SRFIs:
Alex Shinn
Daphne Preston-Kendal
Marc Feeley
Olin Shivers
Philip Bewig
Sebastian Egner
Shiro Kawai
Taylor Campbell
Will Clinger
... and many more!

1. I'd like to end the talk by thanking the members of Working Group 2, who have spent a lot of time discussing things and writing proposals for this project. I want to thank all of them for their effort in working on making R^{7}RS-Large a reality by coming to meetings in the odd hours of their respective timezones. We would not be making the progress we are making now if it were not for them.
2. I also want to thank the contributors to the SRFIs that are going into the R^{7}RS-Large. This is only a fraction of the people who have contributed, as many people have discussed them on the mailing lists.