Let's agree that *serif* fonts do not always carry boring stuff. And have a taste of it:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. — Revised 7 Report on the Algorithmic Language Scheme, Introduction.

Otherwise said, Scheme offers a minimalist core of powerful primitives upon which one can build abstractions to solve (real world) problems.

The Scheme universe is vast and prolific. As programming languages, Scheme dialects target various niches and implement various paradigms. Some of them are part of the de facto standards (namely RnRS and SRFIs).

The best-known paradigm of agreed-upon practices revolve around Functional Programming.

Scheme might be a dynamically typed language, but it can compete with its scions and siblings when performance matters.

Few programming languages can compete with Scheme when it comes to computer science whether it is Programming Language Theory, or Artificial Intelligence.

That being said, Scheme implementations might be missing some love. That's a good opportunity for you to learn something useful and give something back.

Or, like others, to make it your secret sauce.

Discourse

Tutorial

Specification

Standard Libraries

- `(scheme base)` R7RS-small
- `(scheme bitwise)`
- `(scheme box)`
- `(scheme bytevector)`
- `(scheme case-lambda)` R7RS-small
- `(scheme char)` R7RS-small
- `(scheme charset)`
- `(scheme comparator)`
- `(scheme complex)` R7RS-small
- `(scheme cxr)` R7RS-small
- `(scheme division)`
- `(scheme ephemeron)`
- `(scheme eval)` R7RS-small

- `(scheme file)` R7RS-small
- `(scheme fixnum)`
- `(scheme flonum)`
- `(scheme generator)`
- `(scheme hash-table)`
- `(scheme idque)`
- `(scheme ilist)`
- `(scheme inexact)` R7RS-small
- `(scheme lazy)` R7RS-small
- `(scheme list)`
- `(scheme list-queue)`
- `(scheme load)` R7RS-small
- `(scheme lseq)`
- `(scheme mapping)`
- `(scheme mapping-hash)`
- `(scheme process-context)` R7RS-small
- `(scheme r5rs)` R7RS-small
- `(scheme read)` R7RS-small
- `(scheme regex)`
- `(scheme repl)` R7RS-small
- `(scheme rlist)`
- `(scheme set)`
- `(scheme show)`
- `(scheme sort)`
- `(scheme stream)`
- `(scheme text)`
- `(scheme time)` R7RS-small
- `(scheme vector)`
- `(scheme write)` R7RS-small

Source, and single page files

You can find the source over the rainbow. There is available a single markdown file, and a single html file and a pdf;

LICENSE

 # A cheatsheet on Discourse.

The three gates of speech

Before you speak, let your words pass through three gates.

- At the first gate, ask yourself, is it true.
- At the second gate ask, is it necessary.
- At the third gate ask, is it kind.

Rogerian rhetoric

> 1. You should attempt to re-express your target's position so clearly, vividly, and fairly that your target says, “Thanks, I wish I'd thought of putting it that way.”
> 2. You should list any points of agreement (especially if they are not matters of general or widespread agreement).
> 3. You should mention anything you have learned from your target.
> 4. Only then are you permitted to say so much as a word of rebuttal or criticism.
>
> — Dennett's version of Rapoport's Rules

Argument Ranking

- - High-level generators - Disagreements that remain when everyone understands exactly what's being argued, and agrees on what all the evidence says, but have vague and hard-to-define reasons for disagreeing.
- - Operationalizing - Where both parties understand they're in a cooperative effort to fix exactly what they're arguing about.
- - Survey of evidence - Not trying to devastate the other person with a mountain of facts and start looking at the studies and arguments on both sides and figuring out what kind of complex picture they paint.

- - Disputing definitions - Argument hinges on the meaning of words, or whether something counts as a member of a category or not.
- - Single Studies - Better than scattered facts, proving they at least looked into the issue formally.
- - Demands for rigor - Attempts to demand that an opposing argument be held to such strict standards that nothing could possibly clear the bar.
- - Single Facts- One fact, which admittedly does support their argument, but presented as if it solves the debate in and of itself.
- - Gotchas - Short claims that purport to be devastating proof that one side can't possibly be right.
- - Social shaming- A demand for listeners to place someone outside the boundary of whom deserve to be heard.

"How to apologize: Quickly, specifically, sincerely."

— Kevin Kelly

Arguments

- **Ad baculum** : Argument relying on an appeal to fear or a threat.
- **Ad ignorantiam** : Argument relying on people's ignorance.
- **Ad populum** : Argument relying on sentimental weakness.
- **Ad verecundiam** : Argument relying on the the words of an "expert", or authority.
- **Ex silentio** : Argument relying on ignorance.
- **Ex nihilo** : An argument that bears no relation to the previous topic of discussion.
- **Non sequitur** : An inference that does not follow from established premises or evidence.

Responses

- **Akrasia** : State of acting against one's better judgment.
- **Connotation** : Emotional association with a word.
- **Intransigence** : Refusal to change one's views or to agree about something.
- **Inferential distance** : Gap between the background knowledge and epistemology of a person trying to explain an idea, and the background knowledge and epistemology of the person trying to understand it.
- **Straw man** : Creating a false or made up scenario and then attacking it. Painting your opponent with false colors only deflects the purpose of the argument.
- **Steel man** : To steelman is to address the strongest possible variant or the most charitable interpretation of an idea, rather than the most available phrasings.

- **Red herring** : A diversion from the active topic.
- **Rationalization** : Starts from a conclusion, and then works backward to arrive at arguments apparently favouring that conclusion. Rationalization argues for a side already selected.
- **Dogpiling** : A disagreement wherein one person says something wrong or offensive, and a large number of people comment in response to tell them how wrong they are, and continue to disparage the original commenter beyond any reasonable time limit.
- **Grandstanding** : An action that is intended to make people notice and admire you, behaving in a way that makes people pay attention to you instead of thinking about more important matters.
- **Whataboutism** : An attempt to discredit an opponent's position by charging them with hypocrisy without directly refuting or disproving their argument.
- **Dissensus** : The deliberate avoidance of consensus.

Beliefs

- **Belief** : The mental state in which an individual holds a proposition to be true.
- **Priors** : The beliefs an agent holds regarding a fact, hypothesis or consequence, before being presented with evidence.
- **Alief** : An independent source of emotional reaction which can coexist with a contradictory belief. Example The fear felt when a monster jumps out of the darkness in a scary movie is based on the alief that the monster is about to attack you, even though you believe that it cannot.
- **Proper belief** : Requires observations, gets updated upon encountering new evidence, and provides practical benefit in anticipated experience.
- **Improper belief** : Is a belief that isn't concerned with describing the territory. Note that the fact that a belief just happens to be true doesn't mean you're right to have it. If you buy a lottery ticket, certain that it's a winning ticket (for no reason), and it happens to be, believing that was still a mistake.
- **Belief in belief** : Where it is difficult to believe a thing, it is often much easier to believe that you ought to believe it. Were you to really believe and not just believe in belief, the consequences of error would be much more severe. When someone makes up excuses in advance, it would seem to require that belief, and belief in belief, have become unsynchronized.
- **A Priori** : Knowledge which we can be sure of without any empirical evidence(evidence from our senses). So, knowledge that you could realize if you were just a mind floating in a void unconnected to a body.

> "A leader is best when people barely know they exists, when their work is done, their aim fulfilled, people will say: we did it ourselves."
>
> — (Lao Tse), (Dao De Jing)

The first principle of Wikipedia etiquette has been said to be **Assume Good Faith**, also they **Be Bold, but not Reckless**.

Wrong discourse

- Answer: Jumping into a conversation with endless unapplicable, unrealistic or unrelated answers to the question.
- Question: Spouting accusations while cowardly hiding behind the claim of just asking questions, and ignoring the answers. Asking loaded questions.

Good discourse

- Answer: A clear and honest response to the central point of a question without an aggressive attempt to convince.
- Question: question asked with the intention to be fair, open, and honest, regardless of the outcome of the interaction.

Social rules are expected to be broken from time to time, in that regard they are different from a code of conduct.

Response Ranking

- - Central point - Commit to refute explicitly the central point.
- - Refutation - Argue a conflicting passage, explain why it's mistaken.
- - Counterargument - Contradict with added reasoning or evidence.
- - Contradiction - State the opposing case, what.
- - Responding to Tone - Responding to the author's tone, how.
- - Ad Hominem - Attacking the author directly, who.

Interaction Ranking

Discussion

- - Release - Initiating a discussion on the lessons learnt from a project.
- - Update - Presenting the recent development of a personal experience, ongoing event or work in progress.
- - Soapbox - Spontaneous and or enthusiastic posts about a general topic of interest or finding.

Low-Effort

- - Rant- Venting frustration publicly without explicitly looking to have a conversation about the matter.

- - Shitpost - Aggressively or ironically looking for the biggest reaction with the least effort possible. Includes subtoots and vague-posting.

Emotional Reaction

- **Seduction** - You are led to feel that the fulfillment of your dreams depends on your doing what the other is encouraging you to do.
- **Alignment** - The interests of the system are presented as fulfilling your emotional needs. You are led to feel that your survival, your viability in society or your very identity depends on your doing what the other is requiring of you.
- **Reduction** - Complex subjects are reduced to a single, emotionally charged issue.
- **Polarization** - Issues are presented in such a way that you are either right or wrong. You are told that any dialogue between different perspectives is suspect, dangerous or simply not permissible.
- **Marginalization** - You are made to feel that your own interests (or interests that run counter to the interests of the other) are inconsequential.
- **Framing** - The terms of a debate are set so that issues that threaten the system cannot be articulated or discussed. You are led to ignore aspects of the issue that may be vitally important to your own interests but are contrary to the interests of the other that is seeking to make you act in their interests.

Quotes

> "Kings speak for the realm, governors for the state, popes for the church. Indeed, the titled, as titled, cannot speak **with** annyone."
>
> — James P. Carse, Finite and Infinite Games

> "Instead of trying to prove your opponent wrong, try to see in what sense he might be right." — Robert Nozick, Anarchy, State, and Utopia

> "I don't argue: I just say what I know or what I believe, as the case may be." — John W. Cohan

> "You should mention anything you have learned from your target."

LICENSE

with “wrong”) and “faith” (replaced with “discourse”), a few other changes, see history for complete log. # Tutorial

Basics

Continuation

After reading this section you will be able to write basic Scheme programs. In particular, you will study:

- How to comment code
- How to write literals for builtin types
- How to call a procedure
- How to define a variable
- How to compare objects
- How to define a procedure

How to comment code

You can comment code with the semi-colon, that is `;`. Idiomatic code use two semi-colons:

```
;; Everything after one semi-colon is a comment.
```

The following sections will use two semi-colons with followed by an arrow `=>` to describe the return value.

How to write literals for builtin types

number

- Integers can be written as usual `42`
- Inexact reals can be written as usual `3.1415`
- There is more number types. It is called the Numerical tower

boolean

- false: `#f`
- true: `#t`

characters Characters can be written with their natural representation prefixed with `#\\`, for instance the character `x` is represented in Scheme code as follow:

```
#\x
```

string A string is written with double quotes, that is `"`, for instance:

```
"hello world"
```

symbol A symbol is most of the time written with a simple quote prefix, that is `'`. For instance:

```
'unique
```

pair A pair of the symbol `'pi` and the value `3.1415` can be written as:

```
'(pi . 3.1415)
```

list A list can be written as literals separated by one space and enclosed by parenthesis. For instance, the following list has three items:

```
'(unique "hello world" (pi . 3.1415))`
```

The first item is the symbol `'unique`, the second item is a string, the third item is a pair.

The empty list is written `'()`.

vector A vector looks somewhat like a list but without the explicit simple quote. It use a hash prefix. For instance, the following vector has three items:

```
#(unique "hello world" 42)
```

The first item is the symbol `'unique`, the second item is a string, the third item is a number.

bytevector A bytevector is like vector but can contain only bytes. It looks like a list of integers, prefixed with `#vu8`. For instance, the following bytevector has three bytes:

```
#vu8(0 42 255)
```

How to call a procedure

A procedure call looks like a list without the simple quote prefix.

The following describe the addition 21 and 21:

```
(+ 21 21) ;; => 42
```

It returns `42`. So does the following multiplication:

```
(* 21 2) ;; => 42
```

The first item is a procedure object. Most of the time, procedure names are made of letters separated with dashes. That usually called `kebab-case`.

Here is another procedure call:

```
(string-append "hello" " " "world") ;; => "hello world"
```

It will return a string `"hello world"`.

How to define a variable

The first kind of variables that you encountered are procedures, things like `+`, `*` or `string-append`.

Variables can also contain constants. You can use `define`:

```
(define %thruth 42)
```

The above code will create a variable called `%thruth` that contains `42`.

Look at this very complicated computation:

```
(+ %thruth 1 (* 2 647)) ;; => 1337
```

How to compare objects

Identity equivalence To compare by identity, in pratice, whether two object represent the same memory location, you can use the procedure `eq?`.

In the case where you are comparing symbols you can use the procedure `eq?`:

```
(eq? 'unique 'unique) ;; => #t
(eq? 'unique 'singleton) ;; => #f
```

Equivalence

If you do not know the type of the compared objects, or the objects can be of different types, you can use the procedure `equal?`:

```
(equal? #t "true") ;; => #f
```

The string `"true"` is not equivalent to the boolean `#t`.

It is rare to use `equal?`, because, usually, you know the type of the compared objects and the compared object have the same type.

Equivalence predicates

The astute reader might have recognized a pattern in the naming of the equivalence procedures `eq?` and `equal?`: both end with a question mark. That is a convention that all procedures that can only return a boolean should end with a question mark. Those are called *predicates*.

They are predicates for every builtin types. For instance string type has a string equivalence predicate written `string=?`:

```
(string=? "hello" "hello world" "hello, world!") ;; => #f
```

The predicate procedure `string=?` will return `#t` if all arguments are the same string, in the sense they contain the same characters.

How to define a procedure

The simplest procedure ever, is the procedure that takes no argument and returns itself:

```
(define (ruse)
  ruse)
```

The above is sugar syntax for the following:

```
(define ruse (lambda () ruse))
```

A procedure that takes no arguments is called a *thunk*. Indentation and the newline are cosmetic conventions. If you call the procedure `ruse`, it will return `ruse`:

```
(eq? ruse (ruse))
```

One can define a procedure that adds one as follow:

```
(define (add1 number)
  (+ number 1))
```

The predicate to compare numbers is `=`. Hence, the following:

```
(= 2006 (add1 2005)) ;; => #t
```

Mind the fact that it returns a new number. It does not mutate the value even if it is passed as a variable.

Let's imagine a procedure that appends a name to the string `"Hello"`. For instance, given `"Aziz"` or a variable containing `"Aziz"`, it will return `"Hello Aziz"`.

```
(define name "Aziz")

(define (say-hello name)
  (string-append "Hello " name))

(string=? "Hello Aziz" (say-hello name)) ;; => #t

;; XXX: the variable name still contains "Aziz"

(string=? name "Aziz")) ;; => #t
```

It does not matter for the callee whether the arguments are passed as variables or literals:

```
(string=? "Hello John"  (say-hello "John")) ;; => #t
```

Backtrack

In this section you learned:

- How to comment code using a semi-colon character `;`
- How to write literals for builtin types
  - integer: `42`
  - float: `3.1415`
  - symbol: `'unique`
  - string: `"hello world"`
  - pair: `(pi . 3.1415)`
  - list: `'(42 "hello world" (pi . 3.1415))`
  - vector: `#(42 "hello world" (pi . 3.1415))`
  - bytevector: `#vu8(1 42 255)`
- How to call a procedure `(string-append "hello " "Aziz")`
- How to define a variable `(define %thruth 42)`
- How to compare objects using their type specific predicates. For instance: `(string=? "hello" "hello")`
- How to define a procedure again using `define` with slightly different syntax `(define (add1 number) (+ number 1))`

Forward

Continuation

After reading this section you will be able to write more complex Scheme code. In particular you will study:

- How to create lexical bindings
- How to set a variable
- How to do a branch `if`
- How to create a new type
- How to write a named-let

How to create lexical bindings

Lexical bindings can be created with `let`, `let*`, `letrec` and `letrec*`. They have slightly different behaviors, but the same syntax:

```
(let (<binding> ...) <expression> ...)
```

Where `<binding>` looks like an association of a variable name with the initial value it is holding. For instance:

```
(let ((a 1)
      (b 2))
  (+ a b 3)) ;; => 6
```

The above `let` form will bind `a` to `1`, `b` to `2` and return the output of `(+ a b 3)` that is `6`.

How to set a variable

To change what a variable holds without overriding it or mutating the object contained in the varialbe, you can use `set!`. Mind the exclamation mark, it is a convention that forms that have a side-effect ends with a exclamation mark. For instance:

```
(define %thruth 42)

(display %truth)
(newline)

(set! %thruth 101)

(display %truth)
(newline)
```

How to do a branch `if`

Scheme `if` will consider false, only the object `#f`. Hence, one can do the following:

```
(if #t
  (display "true")
  (display "never executed"))
```

Similarly:

```
(if #f
  (display "never executed")
  (display "false"))
```

In particular, the number zero is true according to scheme `if`:

```
(if 0
  (display "zero is true")
  (display "never executed"))
```

If you want to check whether a value is zero you can use the predicate `zero?` like so:

```
(if (zero? %thruth)
   (display "%thruth is zero")
   (display "%thruth is not zero"))
```

Or the less idiomatic predicate `=`:

```
(if (= %truth 0)
  (display "%thruth is zero")
  (display "%thruth is not zero"))
```

How to create a new type

To create a new type you can use the macro `define-record-type`. For instance, in a todo list application, we will need an `<item>` type that can be defined as:

```
(define-record-type <item>
  (make-item title body status)
  item?
  (title item-title item-title!)
  (body item-body item-body!)
  (status item-status item-status!))
```

Where:

- `<item>` is the record name,
- `make-item` is the constructor of record instances,
- `item?` is the predicate that allows to tell whether an object is a `<item>` type,
- `title`, `body` and `status` are fields with their associated getters and setters. Setters ends with an exclamation mark. They will mutate the object. Setters are optional.

Here is an example use of the above `<item>` definition:

```
(define item (make-item "Learn Scheme" "The Scheme programming language is awesome, I should

;; To change the status, one can do the following:

(item-status! item 'wip)

;; to get the title, one can do the following:

(display (item-title item))
(newline)
```

How to write a named-let

A named-let allows to do recursion without going through the ceremony of defining a separate procedure. In pratice, it used in similar contexts such as `for` or `while` loop in other languages. Given the procedure `(cons item items)` that will return a new list with `ITEMS` as tail and `ITEM` as first item, study the following code:

```
(let loop ((index 0)
           (out '()))
  (if (= index 10)
      (display out)
      (loop (+ index 1) (cons index out))))
```

It is equivalent to the following:

```
(define (loop index out)
  (if (= index 10)
      (display out)
      (loop (+ index 1) (cons index out))))

(loop 0 '())
```

A named-let, look like a `let` form that can be used to bind variables prefixed with a name. Here is some pseudo-code that describe the syntax of the named-let form:

```
(let <name> (<binding> ...) expression ...))
```

So `<binding>` and `<expression>` are very similar to a `let`. `<name>` will be bound to a procedure that takes as many argument as there is `<binding>` and its body will be `<expression> ...`. It will be called with the associated objects in `<binding> ...`. `expression` can call `<name>` most likely in tail call position but not necessarly. If the named-let is not tail-recursive, it is also known to be a *grow the stack recursive call*. Another way to see the named-let is pseudo-code:

```
(define <name> (lambda <formals> <expression> ...))

(<name> <arguments> ...)
```

Where:

- `<formals>` are the variable names from `<binding> ...`
- `<arguments>` are the initial object bound in `<binding> ...`

That is all.

Backtrack

- How to create lexical bindings with `let`, `let*`, `letrec` and `letrec*`,
- How to set a variable using `(set! %thruth 42)`,
- How to do a `if` with `(if %thruth (display "That is true") (display "That is false"))`,
- How to create a new type using `define-record-type` that can look like:

```
(define-record-type <record-name>
  (make-record-name field0 ...)
```

```
  record-name?
  (field0 record-name-field0 record-name-field0!))
```

- How to write a named-let, for instance an infinite loop will look like:

```
(let loop ((index 0))
  (display index)
  (loop (+ index 1)))
```

Beyond

Continuation

After reading this section you will be able to create libraries.

Backtrack

Elements of Style

R7RS small specification

Note: This is a port of R7RS specification from tex to markdown that is rendered to html. It does not include formal semantics.

Summary

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and object-oriented styles, find convenient expression in Scheme.

The introduction offers a brief history of the language and of the report.

The first three chapters present the fundamental ideas of the language and describe the notational conventions used for describing the language and for writing programs in the language.

Chapters [expressionchapter] and [programchapter] describe the syntax and semantics of expressions, definitions, programs, and libraries.

Chapter [builtinchapter] describes Scheme's built-in procedures, which include all of the language's data manipulation and input/output primitives.

Chapter [formalchapter] provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics. An example of the use of the language follows the formal syntax and semantics.

Appendix [stdlibraries] provides a list of the standard libraries and the identifiers that they export.

Appendix [stdfeatures] provides a list of optional but standardized implementation feature names.

The report concludes with a list of references and an alphabetic index.

Note: The editors of the R5RS and R6RS reports are listed as authors of this report in recognition of the substantial portions of this report that are copied directly from R5RS and R6RS. There is no intended implication that those editors, individually or collectively, support or do not support this report.

Contents

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Scheme was one of the first programming languages to incorporate first-class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially GOTOs that pass arguments, thus allowing a programming style that is both coherent and efficient. Scheme was the first widely used programming language to embrace first-class escape procedures, from which all previously known sequential control structures can be synthesized. A subsequent version of Scheme introduced the concept of exact and inexact numbers, an extension of Common Lisp's generic arithmetic. More recently, Scheme became the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended in a consistent and reliable manner.

Background

The first description of Scheme was written in 1975 . A revised report appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler . Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University . An introductory computer science textbook using Scheme was published in 1984 .

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Their report, the RRRS , was published at MIT and Indiana University in the summer of 1985. Further revision took place in the spring of 1986, resulting in the R3RS . Work in the spring of 1988 resulted in R4RS , which became the basis for the IEEE Standard for the Scheme Programming Language in 1991 . In 1998, several additions to the IEEE standard, including high-level hygienic macros, multiple return values, and eval, were finalized as the R5RS .

In the fall of 2006, work began on a more ambitious standard, including many new improvements and stricter requirements made in the interest of improved portability. The resulting standard, the R6RS, was completed in August 2007 , and was organized as a core language and set of mandatory standard libraries. Several new implementations of Scheme conforming to it were created. However, most existing R5RS implementations (even excluding those which are essentially unmaintained) did not adopt R6RS, or adopted only selected parts of it.

In consequence, the Scheme Steering Committee decided in August 2009 to divide the standard into two separate but compatible languages — a "small" language, suitable for educators, researchers, and users of embedded languages, focused on R5RS compatibility, and a "large" language focused on the practical needs of mainstream software development, intended to become a replacement for R6RS. The present report describes the "small" language of that effort: therefore it cannot be considered in isolation as the successor to R6RS.

We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementers of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

Acknowledgments

We would like to thank the members of the Steering Committee, William Clinger, Marc Feeley, Chris Hanson, Jonathan Rees, and Olin Shivers, for their support and guidance.

This report is very much a community effort, and we'd like to thank everyone who provided comments and feedback, including the following people: David Adler, Eli Barzilay, Taylan Ulrich Bayırlı/Kammer, Marco Benelli, Pierpaolo Bernardi, Peter Bex, Per Bothner, John Boyle, Taylor Campbell, Raffael Cavallaro, Ray Dillinger, Biep Durieux, Sztefan Edwards, Helmut Eller, Justin Ethier, Jay Reynolds Freeman, Tony Garnock-Jones, Alan Manuel Gloria, Steve Hafner, Sven Hartrumpf, Brian Harvey, Moritz Heidkamp, Jean-Michel Hufflen, Aubrey Jaffer, Takashi Kato, Shiro Kawai, Richard Kelsey, Oleg Kiselyov, Pjotr Kourzanov, Jonathan Kraut, Daniel Krueger, Christian Stigen Larsen, Noah

Lavine, Stephen Leach, Larry D. Lee, Kun Liang, Thomas Lord, Vincent Stewart Manis, Perry Metzger, Michael Montague, Mikael More, Vitaly Magerya, Vincent Manis, Vassil Nikolov, Joseph Wayne Norton, Yuki Okumura, Daichi Oohashi, Jeronimo Pellegrini, Jussi Piitulainen, Alex Queiroz, Jim Rees, Grant Rettke, Andrew Robbins, Devon Schudy, Bakul Shah, Robert Smith, Arthur Smyles, Michael Sperber, John David Stone, Jay Sulzberger, Malcolm Tredinnick, Sam Tobin-Hochstadt, Andre van Tonder, Daniel Villeneuve, Denis Washington, Alan Watson, Mark H. Weaver, Göran Weinholt, David A. Wheeler, Andy Wingo, James Wise, Jörg F. Wittenberger, Kevin A. Wortman, Sascha Ziemann.

In addition we would like to thank all the past editors, and the people who helped them in turn: Hal Abelson, Norman Adams, David Bartley, Alan Bawden, Michael Blair, Gary Brooks, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Robert Findler, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Halstead, Robert Hieb, Paul Hudak, Morry Katz, Eugene Kohlbecker, Chris Lindblad, Jacob Matthews, Mark Meyer, Jim Miller, Don Oxley, Jim Philbin, Kent Pitman, John Ramsdell, Guillermo Rozas, Mike Shaff, Jonathan Shapiro, Guy Steele, Julie Sussman, Perry Wagle, Mitchel Wand, Daniel Weise, Henry Wu, and Ozan Yigit. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual* . We gladly acknowledge the influence of manuals for MIT Scheme , T , Scheme 84 , Common Lisp , and Algol 60 , as well as the following SRFIs: 0, 1, 4, 6, 9, 11, 13, 16, 30, 34, 39, 43, 46, 62, and 87, all of which are available at http://srfi.schemers.org.

Overview of Scheme

Semantics

This section gives an overview of Scheme's semantics. A detailed informal semantics is the subject of chapters [basicchapter] through [builtinchapter]. For reference purposes, section [formalsemanticssection] provides a formal semantics of Scheme.

Scheme is a statically scoped programming language. Each use of a variable is associated with a lexically apparent binding of that variable.

Scheme is a dynamically typed language. Types are associated with values (also called objects) rather than with variables. Statically typed languages, by contrast, associate types with variables and expressions as well as with values.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they

can prove that the object cannot possibly matter to any future computation.

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar. See section [proper tail recursion].

Scheme procedures are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section [continuations].

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, regardless of whether the procedure needs the result of the evaluation.

Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex number. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. Exact arithmetic is not limited to integers.

Syntax

Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and other data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is that Scheme programs and data can easily be treated uniformly by other Scheme programs. For example, the eval procedure evaluates a Scheme program expressed as data.

The read procedure performs syntactic as well as lexical decomposition of the data it reads. The read procedure parses its input as data (section [datumsyntax]), not as program.

The formal syntax of Scheme is described in section [BNF].

Notation and terminology

Base and optional features Every identifier defined in this report appears in one or more of several *libraries*. Identifiers defined in the *base library* are not marked specially in the body of the report. This library includes the core syntax of Scheme and generally useful procedures that manipulate data. For example, the variable abs is bound to a procedure of one argument that computes the absolute value of a number, and the variable + is bound to a procedure that computes sums. The full list all the standard libraries and the identifiers they export is given in Appendix [stdlibraries].

All implementations of Scheme:

- Must provide the base library and all the identifiers exported from it.
- May provide or omit the other libraries given in this report, but each library must either be provided in its entirety, exporting no additional identifiers, or else omitted altogether.
- May provide other libraries not described in this report.
- May also extend the function of any identifier in this report, provided the extensions are not in conflict with the language reported here.
- Must support portable code by providing a mode of operation in which the lexical syntax does not conflict with the lexical syntax described in this report.

Error situations and unspecified behavior When speaking of an error situation, this report uses the phrase "an error is signaled" to indicate that implementations must detect and report the error. An error is signaled by raising a non-continuable exception, as if by the procedure raise as described in section [exceptionsection]. The object raised is implementation-dependent and need not be distinct from objects previously used for the same purpose. In addition to errors signaled in situations described in this report, programmers can signal their own errors and handle signaled errors.

The phrase "an error that satisfies *predicate* is signaled" means that an error is signaled as above. Furthermore, if the object that is signaled is passed to the specified predicate (such as file-error? or read-error?), the predicate returns #t.

If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so. Such a situation is sometimes, but not always, referred to with the phrase "an error." In such a situation, an implementation may or may not signal an error; if it does signal an error, the object that is signaled may or may not satisfy the predicates error-object?, file-error?, or read-error?. Alternatively, implementations may provide non-portable extensions.

For example, it is an error for a procedure to be passed an argument of a

type that the procedure is not explicitly specified to handle, even though such domain errors are seldom mentioned in this report. Implementations may signal an error, extend a procedure's domain of definition to include such arguments, or fail catastrophically.

This report uses the phrase "may report a violation of an implementation restriction" to indicate circumstances under which an implementation is permitted to report that it is unable to continue execution of a correct program because of some restriction imposed by the implementation. Implementation restrictions are discouraged, but implementations are encouraged to report violations of implementation restrictions.

For example, an implementation may report a violation of an implementation restriction if it does not have enough storage to run a program, or if an arithmetic operation would produce an exact number that is too large for the implementation to represent.

If the value of an expression is said to be "unspecified," then the expression must evaluate to some object without signaling an error, but the value depends on the implementation; this report explicitly does not say what value is returned.

Finally, the words and phrases "must," "must not," "shall," "shall not," "should," "should not," "may," "required," "recommended," and "optional," although not capitalized in this report, are to be interpreted as described in RFC 2119 . They are used only with reference to implementer or implementation behavior, not with reference to programmer or program behavior.

Entry format Chapters [expressionchapter] and [builtinchapter] are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a procedure. An entry begins with one or more header lines of the form

template *category*

for identifiers in the base library, or

template *name* library *category*

where *name* is the short name of a library as defined in Appendix [stdlibraries].

If *category* is "syntax," the entry describes an expression type, and the template gives the syntax of the expression type. Components of expressions are designated by syntactic variables, which are written using angle brackets, for example expression and variable. Syntactic variables are intended to denote segments of program text; for example, expression stands for any string of characters which is a syntactically valid expression. The notation

`thing ...`

indicates zero or more occurrences of a `thing`, and

`thing thing ...`

indicates one or more occurrences of a thing.

If *category* is "auxiliary syntax," then the entry describes a syntax binding that occurs only as part of specific surrounding expressions. Any use as an independent syntactic construct or variable is an error.

If *category* is "procedure," then the entry describes a procedure, and the header line gives a template for a call to the procedure. Argument names in the template are *italicized*. Thus the header line

`(vector-ref vector k)` *procedure*

indicates that the procedure bound to the `vector-ref` variable takes two arguments, a vector *vector* and an exact non-negative integer k (see below). The header lines

(make-vector k) procedure

(make-vector k *fill*) procedure

indicate that the `make-vector` procedure must be defined to take either one or two arguments.

It is an error for a procedure to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section [disjointness], then it is an error if that argument is not of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` is a vector. The following naming conventions also imply type restrictions:

alist	association list (list of pairs)
boolean	boolean value (#t or #f)
byte	exact integer 0 $\leq$ *byte* $<$ 256
bytevector	bytevector
char	character
end	exact non-negative integer
k, $k1$, ... kj, ...	exact non-negative integer
letter	alphabetic character
list, *list*1, ... *listj*, ...	list (see section [listsection])
n, $n1$, ... nj, ...	integer
obj	any object
pair	pair
port	port
proc	procedure
q, $q1$, ... qj, ...	rational number
start	exact non-negative integer
string	string
symbol	symbol

thunk	zero-argument procedure
*vecto*r*	vector
x, *x*1, ... *xj*, ...	real number
y, *y*1, ... *yj*, ...	real number
z, *z*1, ... *zj*, ...	complex number

The names *start* and *end* are used as indexes into strings, vectors, and bytevectors. Their use implies the following:

- It is an error if *start* is greater than *end*.
- It is an error if *end* is greater than the length of the string, vector, or bytevector.
- If *start* is omitted, it is assumed to be zero.
- If *end* is omitted, it assumed to be the length of the string, vector, or bytevector.
- The index *start* is always inclusive and the index *end* is always exclusive. As an example, consider a string. If *start* and *end* are the same, an empty substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

Evaluation examples The symbol " " used in program examples is read "evaluates to." For example,

```
(* 5 8) ;; => 40
```

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in an environment containing the base library, to an object that can be represented externally by the sequence of characters "`40`." See section [externalreps] for a discussion of external representations of objects.

Naming conventions By convention, `?` is the final character of the names of procedures that always return a boolean value. Such procedures are called *predicates*. Predicates are generally understood to be side-effect free, except that they may raise an exception when passed the wrong type of argument.

Similarly, `!` is the final character of the names of procedures that store values into previously allocated locations (see section [storagemodel]). Such procedures are called *mutation procedures*. The value returned by a mutation procedure is unspecified.

By convention, "`->`" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, list-

>vector takes a list and returns a vector whose elements are the same as those of the list.

A *command* is a procedure that does not return useful values to its continuation.

A *thunk* is a procedure that does not accept arguments.

Lexical conventions

This section gives an informal account of some of the lexical conventions used in writing Scheme programs. For a formal syntax of Scheme, see section [BNF].

Identifiers

An identifier is any sequence of letters, digits, and "extended identifier characters" provided that it does not have a prefix which is a valid number. However, the `.` token (a single period) used in the list syntax is not an identifier.

All implementations of Scheme must support the following extended identifier characters:

```
!\ \$ \% \verb"&" * + - . / :\ < = > ? @ \verb"^" \verb"_" \verb"~" %
```

Alternatively, an identifier can be represented by a sequence of zero or more characters enclosed within vertical lines (|), analogous to string literals. Any character, including whitespace characters, but excluding the backslash and vertical line characters, can appear verbatim in such an identifier. In addition, characters can be specified using either an inline hex escape or the same escapes available in strings.

For example, the identifier `|H\x65;llo|` is the same identifier as `Hello`, and in an implementation that supports the appropriate Unicode character the identifier `|\x3BB;|` is the same as the identifier . What is more, `|\t\t|` and `|\x9;\x9;|` are the same. Note that `||` is a valid identifier that is different from any other identifier.

Here are some examples of identifiers:

```
...                         {+}
+soup+                      <=?
->string                    a34kTMNs
lambda                      list->vector
q                           V17a
|two words|                 |two\x20;words|
the-word-recursion-has-many-meanings
```

See section [extendedalphas] for the formal syntax of identifiers.

Identifiers have two uses within Scheme programs:

- Any identifier can be used as a variable or as a syntactic keyword (see sections [variablesection] and [macrosection]).

- When an identifier appears as a literal or within a literal (see section [quote]), it is being used to denote a *symbol* (see section [symbolsection]).

In contrast with earlier revisions of the report , the syntax distinguishes between upper and lower case in identifiers and in characters specified using their names. However, it does not distinguish between upper and lower case in numbers, nor in inline hex escapes used in the syntax of identifiers, characters, or strings. None of the identifiers defined in this report contain upper-case characters, even when they appear to do so as a result of the English-language convention of capitalizing the first word of a sentence.

The following directives give explicit control over case folding.

```
#!fold-case
#!no-fold-case
```

These directives can appear anywhere comments are permitted (see section [wscommentsection]) but must be followed by a delimiter. They are treated as comments, except that they affect the reading of subsequent data from the same port. The #!fold-case directive causes subsequent identifiers and character names to be case-folded as if by string-foldcase (see section [stringsection]). It has no effect on character literals. The #!no-fold-case directive causes a return to the default, non-folding behavior.

Whitespace and comments

Whitespace characters include the space, tab, and newline characters. (Implementations may provide additional whitespace characters such as page break.) Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number, but is otherwise insignificant. Whitespace can occur between any two tokens, but not within a token. Whitespace occurring inside a string or inside a symbol delimited by vertical lines is significant.

The lexical syntax includes several comment forms. Comments are treated exactly like whitespace.

A semicolon (`;`) indicates the start of a line comment. The comment continues to the end of the line on which the semicolon appears.

Another way to indicate a comment is to prefix a datum (cf. section [datumsyntax]) with `#;` and optional whitespace. The comment consists of the comment prefix `#;`, the space, and the datum together. This notation is useful for "commenting out" sections of code.

Block comments are indicated with properly nested `#|` and `|#` pairs.

```
#|
The FACT procedure computes the factorial
```

```
of a non-negative integer.
|#
(define fact
  (lambda (n)
    (if (= n 0)
        #;(= n 1)
        1         ;Base case: return 1
        (* n (fact (- n 1))))))
```

Other notations

For a description of the notations used for numbers, see section [numbersection].

- `. + -` These are used in numbers, and can also occur anywhere in an identifier. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (section [listsection]), and to indicate a rest-parameter in a formal parameter list (section [lambda]). Note that a sequence of two or more periods *is* an identifier.
- `( )` Parentheses are used for grouping and to notate lists (section [listsection]).
- `'` The apostrophe (single quote) character is used to indicate literal data (section [quote]).
- “‘ The grave accent (backquote) character is used to indicate partly constant data (section [quasiquote]).
- `, ,@` The character comma and the sequence comma at-sign are used in conjunction with quasiquotation (section [quasiquote]).
- `"` The quotation mark character is used to delimit strings (section [stringsection]).
- `\` Backslash is used in the syntax for character constants (section [charactersection]) and as an escape character within string constants (section [stringsection]) and identifiers (section [extendedalphas]).
- `[ ] { }` Left and right square and curly brackets (braces) are reserved for possible future extensions to the language.
- `#` The number sign is used for a variety of purposes depending on the character that immediately follows it:
- `#t #f` These are the boolean constants (section [booleansection]), along with the alternatives `#true` and `#false`.
- `#``'` This introduces a character constant (section [charactersection]).
- `#``(` This introduces a vector constant (section [vectorsection]). Vector constants are terminated by `)` .

- `#``u8(` This introduces a bytevector constant (section [bytevectorsection]). Bytevector constants are terminated by `)` .
- `#e #i #b #o #d #x` These are used in the notation for numbers (section [numbernotations]).
- `#n= #n#` These are used for labeling and referencing other literal data (section [labelsection]).

Datum labels

#n=datum lexical syntax #n# lexical syntax The lexical syntax #n=datum reads the same as datum, but also results in datum being labelled by n. It is an error if n is not a sequence of digits.

The lexical syntax #n# serves as a reference to some object labelled by #n=; the result is the same object as the #n= (see section [equivalencesection]).

Together, these syntaxes permit the notation of structures with shared or circular substructure.

```
(let ((x (list 'a 'b 'c)))
  (set-cdr! (cddr x) x)
  x) ;; => #0=(a b c . #0#)
```

The scope of a datum label is the portion of the outermost datum in which it appears that is to the right of the label. Consequently, a reference #n# can occur only after a label #n=; it is an error to attempt a forward reference. In addition, it is an error if the reference appears as the labelled object itself (as in #n= #n#), because the object labelled by #n= is not well defined in this case.

It is an error for a program or library to include circular references except in literals. In particular, it is an error for quasiquote (section [quasiquote]) to contain them.

```
    #1=(begin (display #\x) #1#)
 ;; => \scherror
```

Basic concepts

Variables, syntactic keywords, and regions

An identifier can name either a type of syntax or a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* and is said to be *bound* to a transformer for that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology,

the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and to bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called *binding constructs*.

Those that bind syntactic keywords are listed in section [macrosection]. The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs (except top-level bindings) can be explained in terms of lambda expressions. The other variable binding constructs are let, let*, letrec, letrec*, let-values, let*-values, and do expressions (see sections [lambda], [letrec], and [do]).

Scheme is a language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the global environment, if any (chapters [expressionchapter] and [initialenv]); if there is no binding for the identifier, it is said to be *unbound*.

Disjointness of types

No object satisfies more than one of the following predicates:

```
boolean?          bytevector?
char?             eof-object?
null?             number?
pair?             port?
procedure?        string?
symbol?           vector?
```

and all predicates created by define-record-type.

These predicates define the types *boolean, bytevector, character*, the empty list object, *eof-object, number, pair, port, procedure, string, symbol, vector*, and all record types.

Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test. As explained in section [booleansection], all values count as true in such a test except for #f. This report uses the word “true” to refer to any Scheme value except #f, and the word “false” to refer to #f.

External representations

An important concept in Scheme (and Lisp) is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters "`28`", and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters "`(8 13)`".

The external representation of an object is not necessarily unique. The integer 28 also has representations "`#e28.000`" and "`#x1c`", and the list in the previous paragraph also has the representations "`( 08 13 )`" and "`(8 . (13 . ()))`" (see section [listsection]).

Many objects have standard external representations, but some, such as procedures, do not have standard representations (although particular implementations may define representations for them).

An external representation can be written in a program to obtain the corresponding object (see quote, section [quote]).

External representations can also be used for input and output. The procedure read (section [read]) parses external representations, and the procedure write (section [write]) generates them. Together, they provide an elegant and powerful input/output facility.

Note that the sequence of characters "`(+ 2 6)`" is *not* an external representation of the integer 8, even though it *is* an expression evaluating to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol `+` and the integers 2 and 6. Scheme's syntax has the property that any sequence of characters that is an expression is also the external representation of some object. This can lead to confusion, since it is not always obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects in the appropriate sections of chapter [initialenv].

Storage model

Variables and objects such as pairs, strings, vectors, and bytevectors implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. A new value can be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as car, vector-ref, or string-ref, is equivalent in the sense of `eqv?` (sec-

tion [equivalencesection]) to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use.

Whenever this report speaks of storage being newly allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them. Notwithstanding this, it is understood that the empty list cannot be newly allocated, because it is a unique object. It is also understood that empty strings, empty vectors, and empty bytevectors, which contain no locations, may or may not be newly allocated.

Every object that denotes locations is either mutable or immutable. Literal constants, the strings returned by `symbol->string`, and possibly the environment returned by scheme-report-environment are immutable objects. All objects created by the other procedures listed in this report are mutable. It is an error to attempt to store a new value into a location that is denoted by an immutable object.

These locations are to be understood as conceptual, not physical. Hence, they do not necessarily correspond to memory addresses, and even if they do, the memory address might not be constant.

Rationale: In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only memory. Making it an error to alter constants permits this implementation strategy, while not requiring other systems to distinguish between mutable and immutable objects.

Proper tail recursion

Implementations of Scheme are required to be *properly tail-recursive*. Procedure calls that occur in certain syntactic contexts defined below are *tail calls*. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is *active* if the called procedure might still return. Note that this includes calls that might be returned from either by the current continuation or by continuations captured earlier by call-with-current-continuation that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in .

Rationale:

Intuitively, no space is needed for an active tail call because the continuation that is used in the tail call has the same semantics as the continuation passed to the procedure containing the call. Although an improper implementation might use a new continuation in the call, a return to this new continuation would be

followed immediately by a return to the continuation passed to the procedure. A properly tail-recursive implementation returns to that continuation directly.

Proper tail recursion was one of the central ideas in Steele and Sussman's original version of Scheme. Their first Scheme interpreter implemented both functions and actors. Control flow was expressed using actors, which differed from functions in that they passed their results on to another actor instead of returning to a caller. In the terminology of this section, each actor finished with a tail call to another actor.

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language.

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as tail expression below, occurs in a tail context. The same is true of all the bodies of case-lambda expressions. ¯ |¯ (lāmbda formals expression tail expression)

  (case-lambda (formals tail body))

- If one of the following expressions is in a tail context, then the subexpressions shown as tail expression are in a tail context. These were derived from rules in the grammar given in chapter [formalchapter] by replacing some occurrences of body with tail body, some occurrences of expression with tail expression, and some occurrences of sequence with tail sequence. Only those rules that contain tail contexts are shown here.

  ¯ |¯ (if expression tail expression tail expression) (if expression tail expression)

  (cond cond clause+) (cond cond clause (else tail sequence))

  (cāse expression +) (cāse expression (else tail sequence))

  (and expression tail expression) (or expression tail expression)

  (when test tail sequence) (unless test tail sequence)

  (let (binding spec) tail body) (let variable (binding spec) tail body) (let* (binding spec) tail body) (letrec (binding spec) tail body) (letrec* (binding spec) tail body) (let-values (mv binding spec) tail body) (let*-values (mv binding spec) tail body)

  (let-syntax (syntax spec) tail body) (letrec-syntax (syntax spec) tail body)

  (begin tail sequence)

  (dō (iteration spec) (test tail sequence))

where

cond clause (test tail sequence) case clause ((datum) tail sequence)

tail body definition tail sequence tail sequence expression tail expression

- If a cond or case expression is in a tail context, and has a clause of the form (expression1 => expression2) then the (implied) call to the procedure that results from the evaluation of expression2 is in a tail context. expression2 itself is not in a tail context.

Certain procedures defined in this report are also required to perform tail calls. The first argument passed to `apply` and to `call-with-current-continuation`, and the second argument passed to `call-with-values`, must be called via a tail call. Similarly, `eval` must evaluate its first argument as if it were in tail position within the `eval` procedure.

In the following example the only tail call is the call to f. None of the calls to g or h are tail calls. The reference to x is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

Note: Implementations may recognize that some non-tail calls, such as the call to h above, can be evaluated as though they were tail calls. In the example above, the let expression could be compiled as a tail call to h. (The possibility of h returning an unexpected number of values can be ignored, because in that case the effect of the let is explicitly unspecified and implementation-dependent.)

Expressions

Expression types are categorized as *primitive* or *derived*. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive, but can instead be defined as macros. Suitable syntax definitions of some of the derived expressions are given in section [derivedsection].

The procedures force, promise?, make-promise, and make-parameter are also described in this chapter because they are intimately associated with the delay, delay-force, and parameterize expression types.

Primitive expression types

Variable references variable syntax An expression consisting of a variable (section [variablesection]) is a variable reference. The value of the variable ref-

erence is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

```
(define x 28)
x ;; => 28
```

Literal expressions (quote *datum*) syntax 'datum syntax constant syntax (quote datum) evaluates to datum. Datum can be any external representation of a Scheme object (see section [externalreps]). This notation is used to include literal constants in Scheme code.

```
(quote a) ;; => a
(quote #(a b c)) ;; => #(a b c)
(quote (+ 1 2)) ;; => (+ 1 2)
```

(quote datum) can be abbreviated as 'datum. The two notations are equivalent in all respects.

```
'a ;; => a
'#(a b c) ;; => #(a b c)
'() ;; => ()
'(+ 1 2) ;; => (+ 1 2)
'(quote a) ;; => (quote a)
''a ;; => (quote a)
```

Numerical constants, string constants, character constants, vector constants, bytevector constants, and boolean constants evaluate to themselves; they need not be quoted.

```
'145932 ;; => 145932
145932 ;; => 145932
'"abc" ;; => "abc"
"abc" ;; => "abc"
'#\a ;; => #\a
#\a ;; => #\a
'#(a 10) ;; => #(a 10)
#(a 10) ;; => #(a 10)
'#u8(64 65) ;; => #u8(64 65)
#u8(64 65) ;; => #u8(64 65)
'#t ;; => #t
#t ;; => #t
```

As noted in section [storagemodel], it is an error to attempt to alter a constant (i.e. the value of a literal expression) using a mutation procedure like set-car! or string-set!.

Procedure calls (operator operand1 ...) syntax A procedure call is written by enclosing in parentheses an expression for the procedure to be called followed by expressions for the arguments to be passed to it. The operator and operand

expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

```
(+ 3 4) ;; => 7
((if #f + *) 3 4) ;; => 12
```

The procedures in this document are available as the values of variables exported by the standard libraries. For example, the addition and multiplication procedures in the above examples are the values of the variables + and * in the base library. New procedures are created by evaluating lambda expressions (see section [lambda]).

Procedure calls can return any number of values (see `values` in section [proceduresection]). Most of the procedures defined in this report return one value or, for procedures such as apply, pass on the values returned by a call to one of their arguments. Exceptions are noted in the individual descriptions.

Note: In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

Note: Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

Note: In many dialects of Lisp, the empty list, `()`, is a legitimate expression evaluating to itself. In Scheme, it is an error.

Procedures (lambda *formals body*) syntax *Syntax:* Formals is a formal arguments list as described below, and body is a sequence of zero or more definitions followed by one or more expressions.

Semantics: A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, and the corresponding actual argument values will be stored in those locations. (A *fresh* location is one that is distinct from every previously existing location.) Next, the expressions in the body of the lambda expression (which, if it contains definitions, represents a letrec* form — see section [letrecstar]) will be evaluated sequentially in the extended environment. The results of the last expression in the body will be returned as the results of the procedure call.

```
(lambda (x) (+ x x)) ;; => a procedure
((lambda (x) (+ x x)) 4) ;; => 8

(define reverse-subtract
```

```
  (lambda (x y) (- y x)))
(reverse-subtract 7 10) ;; => 3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6) ;; => 10
```

Formals have one of the following forms:

- `(variable_1 \ldots\,)`: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in fresh locations that are bound to the corresponding variables.
- variable: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in a fresh location that is bound to variable.
- `(variable_1 \ldots\, variable_{n} . variable_{n+1})`: If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period (it is an error if there is not at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a variable to appear more than once in formals.

```
((lambda x x) 3 4 5 6) ;; => (3 4 5 6)
((lambda (x y . z) z) 3 4 5 6) ;; => (5 6)
```

Each procedure created as the result of evaluating a lambda expression is (conceptually) tagged with a storage location, in order to make `eqv?` and `eq?` work on procedures (see section [equivalencesection]).

Conditionals (if *test consequent alternate*) syntax (if *test consequent*) syntax
Syntax: Test, consequent, and alternate are expressions.

Semantics: An if expression is evaluated as follows: first, test is evaluated. If it yields a true value (see section [booleansection]), then consequent is evaluated and its values are returned. Otherwise alternate is evaluated and its values are returned. If test yields a false value and no alternate is specified, then the result of the expression is unspecified.

```
(if (> 3 2) 'yes 'no) ;; => yes
(if (> 2 3) 'yes 'no) ;; => no
(if (> 3 2)
    (- 3 2)
    (+ 3 2)) ;; => 1
```

Assignments (set! *variable expression*) syntax *Semantics:* Expression is evaluated, and the resulting value is stored in the location to which variable is bound. It is an error if variable is not bound either in some region enclosing the set! expression or else globally. The result of the set! expression is unspecified.

```
(define x 2)
(+ x 1) ;; => 3
(set! x 4) ;; => unspecified
(+ x 1) ;; => 5
```

Inclusion (include *string1 string2 …*) syntax (include-ci *string1 string2 …*) syntax *Semantics:* Both `include` and `include-ci` take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files, read the contents of the files in the specified order as if by repeated applications of read, and effectively replace the include or include-ci expression with a begin expression containing what was read from the files. The difference between the two is that `include-ci` reads each file as if it began with the #!fold-case directive, while `include` does not.

Note: Implementations are encouraged to search for files in the directory which contains the including file, and to provide a way for users to specify other directories to search.

Derived expression types

The constructs in this section are hygienic, as discussed in section [macrosection]. For reference purposes, section [derivedsection] gives syntax definitions that will convert most of the constructs described in this section into the primitive constructs described in the previous section.

Conditionals (cond *clause1 clause2 …*) syntax else auxiliary syntax => auxiliary syntax *Syntax:* Clauses take one of two forms, either

```
(\hyper{test} \hyperi{expression} \dotsfoo)
```

where test is any expression, or

```
(\hyper{test} => \hyper{expression})
```

The last clause can be an "else clause," which has the form

```
(else \hyperi{expression} \hyperii{expression} \dotsfoo)\rm.
```

Semantics: A cond expression is evaluated by evaluating the test expressions of successive clauses in order until one of them evaluates to a true value (see section [booleansection]). When a test evaluates to a true value, the remaining expressions in its clause are evaluated in order, and the results of the last expression in the clause are returned as the results of the entire cond expression.

If the selected clause contains only the test and no expressions, then the value of the test is returned as the result. If the selected clause uses the `=>` alternate form, then the expression is evaluated. It is an error if its value is not a procedure that accepts one argument. This procedure is then called on the value of the test and the values returned by this procedure are returned by the cond expression.

If all tests evaluate to #f, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its expressions are evaluated in order, and the values of the last one are returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less)) ;; => greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal)) ;; => equal

(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f)) ;; => 2
```

(case *key clause1 clause2 …*) syntax *Syntax:* Key can be any expression. Each clause has the form

```
((\hyperi{datum} \dotsfoo) \hyperi{expression} \hyperii{expression} \dotsfoo)\rm,
```

where each datum is an external representation of some object. It is an error if any of the datums are the same anywhere in the expression. Alternatively, a clause can be of the form

```
((\hyperi{datum} \dotsfoo) => \hyper{expression})
```

The last clause can be an "else clause," which has one of the forms

```
(else \hyperi{expression} \hyperii{expression} \dotsfoo)
```

or

```
(else => \hyper{expression})\rm.
```

Semantics: A case expression is evaluated as follows. Key is evaluated and its result is compared against each datum. If the result of evaluating key is the same (in the sense of eqv?; see section [eqv?]) to a datum, then the expressions in the corresponding clause are evaluated in order and the results of the last expression in the clause are returned as the results of the case expression.

If the result of evaluating key is different from every datum, then if there is an else clause, its expressions are evaluated and the results of the last are the results of the case expression; otherwise the result of the case expression is unspecified.

If the selected clause or else clause uses the `=>` alternate form, then the expression is evaluated. It is an error if its value is not a procedure accepting one

argument. This procedure is then called on the value of the key and the values returned by this procedure are returned by the case expression.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ;; => composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b)) ;; => unspecified
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else => (lambda (x) x))) ;; => c
```

(and *test1 ...*) syntax *Semantics:* The test expressions are evaluated from left to right, and if any expression evaluates to #f (see section [booleansection]), then #f is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the values of the last expression are returned. If there are no expressions, then #t is returned.

```
(and (= 2 2) (> 2 1)) ;; => #t
(and (= 2 2) (< 2 1)) ;; => #f
(and 1 2 'c '(f g)) ;; => (f g)
(and) ;; => #t
```

(or *test1 ...*) syntax *Semantics:* The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value (see section [booleansection]) is returned. Any remaining expressions are not evaluated. If all expressions evaluate to #f or if there are no expressions, then #f is returned.

```
(or (= 2 2) (> 2 1)) ;; => #t
(or (= 2 2) (< 2 1)) ;; => #t
(or #f #f #f) ;; => #f
(or (memq 'b '(a b c))
    (/ 3 0)) ;; => (b c)
```

(when *test expression1 expression2 ...*) syntax *Syntax:* The test is an expression.

Semantics: The test is evaluated, and if it evaluates to a true value, the expressions are evaluated in order. The result of the when expression is unspecified.

```
(when (= 1 1.0)
  (display "1")
  (display "2")) ;; => unspecified
;; and prints 12
```

(unless *test expression1 expression2 ...*) syntax *Syntax:* The test is an expression.

Semantics: The test is evaluated, and if it evaluates to #f, the expressions are evaluated in order. The result of the unless expression is unspecified.

```
(unless (= 1 1.0)
  (display "1")
  (display "2")) ;; => unspecified
;; and prints nothing
```

(cond-expand *ce-clause1 ce-clause2 …*) syntax *Syntax:* The `cond-expand` expression type provides a way to statically expand different expressions depending on the implementation. A ce-clause takes the following form:

`(feature requirement expression \ldots\,)`

The last clause can be an "else clause," which has the form

`(else expression \ldots\,)`

A feature requirement takes one of the following forms:

- `feature identifier`
- `(library library name)`
- `(and feature requirement \ldots\,)`
- `(or feature requirement \ldots\,)`
- `(not feature requirement)`

Semantics: Each implementation maintains a list of feature identifiers which are present, as well as a list of libraries which can be imported. The value of a feature requirement is determined by replacing each feature identifier and `(library library name)` on the implementation's lists with #t, and all other feature identifiers and library names with #f, then evaluating the resulting expression as a Scheme boolean expression under the normal interpretation of and, or, and not.

A `cond-expand` is then expanded by evaluating the feature requirements of successive ce-clauses in order until one of them returns #t. When a true clause is found, the corresponding expressions are expanded to a begin, and the remaining clauses are ignored. If none of the feature requirements evaluate to #t, then if there is an else clause, its expressions are included. Otherwise, the behavior of the `cond-expand` is unspecified. Unlike cond, cond-expand does not depend on the value of any variables.

The exact features provided are implementation-defined, but for portability a core set of features is given in appendix [stdfeatures].

Binding constructs The binding constructs let, let*, letrec, letrec*, let-values, and let*-values give Scheme a block structure, like Algol 60. The syntax of the first four constructs is identical, but they differ in the regions they establish for their variable bindings. In a let expression, the initial values are computed before any of the variables become bound; in a let* expression, the bindings and evaluations are performed sequentially; while in letrec and letrec*

expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. The let-values and let*-values constructs are analogous to let and let* respectively, but are designed to handle multiple-valued expressions, binding different identifiers to the returned values.

(let *bindings body*) syntax *Syntax:* Bindings has the form

```
((\hyperi{variable} \hyperi{init}) \dotsfoo)\rm,
```

where each init is an expression, and body is a sequence of zero or more definitions followed by a sequence of one or more expressions as described in section [lambda]. It is an error for a variable to appear more than once in the list of variables being bound.

Semantics: The inits are evaluated in the current environment (in some unspecified order), the variables are bound to fresh locations holding the results, the body is evaluated in the extended environment, and the values of the last expression of body are returned. Each binding of a variable has body as its region.

```
(let ((x 2) (y 3))
  (* x y)) ;; => 6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x))) ;; => 35
```

See also "named let," section [namedlet].

(let* *bindings body*) syntax

Syntax: Bindings has the form

```
((\hyperi{variable} \hyperi{init}) \dotsfoo)\rm,
```

and body is a sequence of zero or more definitions followed by one or more expressions as described in section [lambda].

Semantics: The let* binding construct is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (variable init) is that part of the let* expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on. The variables need not be distinct.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x))) ;; => 70
```

(letrec *bindings body*) syntax *Syntax:* Bindings has the form

((\hyperi{variable} \hyperi{init}) \dotsfoo)\rm,

and body is a sequence of zero or more definitions followed by one or more expressions as described in section [lambda]. It is an error for a variable to appear more than once in the list of variables being bound.

Semantics: The variables are bound to fresh locations holding unspecified values, the inits are evaluated in the resulting environment (in some unspecified order), each variable is assigned to the result of the corresponding init, the body is evaluated in the resulting environment, and the values of the last expression in body are returned. Each binding of a variable has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
;; => #t
```

One restriction on letrec is very important: if it is not possible to evaluate each init without assigning or referring to the value of any variable, it is an error. The restriction is necessary because letrec is defined in terms of a procedure call where a lambda expression binds the variables to the values of the inits. In the most common uses of letrec, all the inits are lambda expressions and the restriction is satisfied automatically.

(letrec* *bindings body*) syntax *Syntax:* Bindings has the form

((\hyperi{variable} \hyperi{init}) \dotsfoo)\rm,

and body is a sequence of zero or more definitions followed by one or more expressions as described in section [lambda]. It is an error for a variable to appear more than once in the list of variables being bound.

Semantics: The variables are bound to fresh locations, each variable is assigned in left-to-right order to the result of evaluating the corresponding init (interleaving evaluations and assignments), the body is evaluated in the resulting environment, and the values of the last expression in body are returned. Despite the left-to-right evaluation and assignment order, each binding of a variable has the entire letrec* expression as its region, making it possible to define mutually recursive procedures.

If it is not possible to evaluate each init without assigning or referring to the

value of the corresponding variable or the variable of any of the bindings that follow it in bindings, it is an error. Another restriction is that it is an error to invoke the continuation of an init more than once.

```
;; Returns the arithmetic, geometric, and
;; harmonic means of a nested list of numbers
(define (means ton)
  (letrec*
     ((mean
        (lambda (f g)
          (f (/ (sum g ton) n))))
      (sum
        (lambda (g ton)
          (if (null? ton)
            (+)
            (if (number? ton)
                (g ton)
                (+ (sum g (car ton))
                   (sum g (cdr ton)))))))
      (n (sum (lambda (x) 1) ton)))
    (values (mean values values)
            (mean exp log)
            (mean / /))))
```

Evaluating (means '(3 (1 4))) returns three values: 8/3, 2.28942848510666 (approximately), and 36/19.

(let-values *mv binding spec body*) syntax *Syntax:* Mv binding spec has the form

```
((\hyperi{formals} \hyperi{init}) \dotsfoo)\rm,
```

where each init is an expression, and body is zero or more definitions followed by a sequence of one or more expressions as described in section [lambda]. It is an error for a variable to appear more than once in the set of formals.

Semantics: The inits are evaluated in the current environment (in some unspecified order) as if by invoking call-with-values, and the variables occurring in the formals are bound to fresh locations holding the values returned by the inits, where the formals are matched to the return values in the same way that the formals in a lambda expression are matched to the arguments in a procedure call. Then, the body is evaluated in the extended environment, and the values of the last expression of body are returned. Each binding of a variable has body as its region.

It is an error if the formals do not match the number of values returned by the corresponding init.

```
(let-values (((root rem) (exact-integer-sqrt 32)))
  (* root rem)) ;; => 35
```

(let*-values *mv binding spec body*) syntax

Syntax: Mv binding spec has the form

`((\hyper{formals} \hyper{init}) \dotsfoo)\rm,`

and body is a sequence of zero or more definitions followed by one or more expressions as described in section [lambda]. In each formals, it is an error if any variable appears more than once.

Semantics: The let*-values construct is similar to let-values, but the inits are evaluated and bindings created sequentially from left to right, with the region of the bindings of each formals including the inits to its right as well as body. Thus the second init is evaluated in an environment in which the first set of bindings is visible and initialized, and so on.

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y))) ;; => (x y x y)
```

Sequencing Both of Scheme's sequencing constructs are named begin, but the two have slightly different forms and uses:

(begin *expression or definition …*) syntax This form of begin can appear as part of a body, or at the outermost level of a program, or at the REPL, or directly nested in a begin that is itself of this form. It causes the contained expressions and definitions to be evaluated exactly as if the enclosing begin construct were not present.

Rationale: This form is commonly used in the output of macros (see section [macrosection]) which need to generate multiple definitions and splice them into the context in which they are expanded.

(begin *expression1 expression2 …*) syntax This form of begin can be used as an ordinary expression. The expressions are evaluated sequentially from left to right, and the values of the last expression are returned. This expression type is used to sequence side effects such as assignments or input and output.

```
(define x 0)

(and (= x 0)
     (begin (set! x 5)
            (+ x 1))) ;; => 6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1))) ;; => unspecified
;; and prints 4 plus 1 equals 5
```

Note that there is a third form of begin used as a library declaration: see section [librarydeclarations].

Iteration (do ((variable1 init1 step1) syntax `\ldots\,) (test expression \ldots\,) command \ldots\,)`

Syntax: All of init, step, test, and command are expressions.

Semantics: A do expression is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the expressions.

A do expression is evaluated as follows: The init expressions are evaluated (in some unspecified order), the variables are bound to fresh locations, the results of the init expressions are stored in the bindings of the variables, and then the iteration phase begins.

Each iteration begins by evaluating test; if the result is false (see section [booleansection]), then the command expressions are evaluated in order for effect, the step expressions are evaluated in some unspecified order, the variables are bound to fresh locations, the results of the steps are stored in the bindings of the variables, and the next iteration begins.

If test evaluates to a true value, then the expressions are evaluated from left to right and the values of the last expression are returned. If no expressions are present, then the value of the do expression is unspecified.

The region of the binding of a variable consists of the entire do expression except for the inits. It is an error for a variable to appear more than once in the list of do variables.

A step can be omitted, in which case the effect is the same as if (variable init variable) had been written instead of (variable init).

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i)) ;; => #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum))) ;; => 25
```

(let *variable bindings body*) syntax *Semantics:* "Named let" is a variant on the syntax of `let` which provides a more general looping construct than do and can also be used to express recursion. It has the same syntax and semantics as ordinary let except that variable is bound within body to a procedure whose

formal arguments are the bound variables and whose body is body. Thus the execution of body can be repeated by invoking the procedure named by variable.

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
;; => ((6 1 3) (-5 -2))
```

Delayed evaluation (delay *expression*) lazy library syntax *Semantics:* The delay construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*. `(delay expression)` returns an object called a *promise* which at some point in the future can be asked (by the force procedure) to evaluate expression, and deliver the resulting value.

The effect of expression returning multiple values is unspecified.

(delay-force *expression*) lazy library syntax *Semantics:* The expression (delay-force *expression*) is conceptually similar to (delay (force *expression*)), with the difference that forcing the result of delay-force will in effect result in a tail call to (force *expression*), while forcing the result of (delay (force *expression*)) might not. Thus iterative lazy algorithms that might result in a long series of chains of delay and force can be rewritten using delay-force to prevent consuming unbounded space during evaluation.

(force *promise*) lazy library procedure The force procedure forces the value of a *promise* created by `delay`, `delay-force`, or `make-promise`. If no value has been computed for the promise, then a value is computed and returned. The value of the promise must be cached (or "memoized") so that if it is forced a second time, the previously computed value is returned. Consequently, a delayed expression is evaluated using the parameter values and exception handler of the call to force which first requested its value. If *promise* is not a promise, it may be returned unchanged.

```
(force (delay (+ 1 2))) ;; => 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
;; => (3 3)

(define integers
```

```
  (letrec ((next
            (lambda (n)
              (delay (cons n (next (+ n 1)))))))
    (next 0)))
(define head
  (lambda (stream) (car (force stream))))
(define tail
  (lambda (stream) (cdr (force stream))))

(head (tail (tail integers)))
;; => 2
```

The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in delay, and each argument passed to a deconstructor is wrapped in force. The use of (delay-force ...) instead of (delay (force ...)) around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust available storage, because force will in effect force such sequences iteratively.

```
(define (stream-filter p? s)
  (delay-force
   (if (null? (force s))
       (delay '())
       (let ((h (car (force s)))
             (t (cdr (force s))))
         (if (p? h)
             (delay (cons h (stream-filter p? t)))
             (stream-filter p? t))))))

(head (tail (tail (stream-filter odd? integers))))
;; => 5
```

The following examples are not intended to illustrate good programming style, as delay, force, and delay-force are mainly intended for programs written in the functional style. However, they do illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p ;; => a promise
(force p) ;; => 6
p ;; => a promise, still
(begin (set! x 10)
```

```
(force p)) ;; => 6
```

Various extensions to this semantics of delay, force and delay-force are supported in some implementations:

- Calling force on an object that is not a promise may simply return the object.
- It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either #t or to #f, depending on the implementation:

  ```
  (eqv? (delay 1) 1) ;; => unspecified
  (pair? (delay (cons 1 2))) ;; => unspecified
  ```

- Implementations may implement "implicit forcing," where the value of a promise is forced by procedures that operate only on arguments of a certain type, like cdr and *. However, procedures that operate uniformly on their arguments, like list, must not force them.

  ```
  (+ (delay (* 3 7)) 13) ;; => unspecified
  (car
   (list (delay (* 3 7)) 13)) ;; => a promise
  ```

(promise? **obj**) lazy library procedure The promise? procedure returns #t if its argument is a promise, and #f otherwise. Note that promises are not necessarily disjoint from other Scheme types such as procedures.

(make-promise **obj**) lazy library procedure The make-promise procedure returns a promise which, when forced, will return *obj*. It is similar to delay, but does not delay its argument: it is a procedure rather than syntax. If *obj* is already a promise, it is returned.

Dynamic bindings The *dynamic extent* of a procedure call is the time between when it is initiated and when it returns. In Scheme, call-with-current-continuation (section [continuations]) allows reentering a dynamic extent after its procedure call has returned. Thus, the dynamic extent of a call might not be a single, continuous time period.

This sections introduces *parameter objects*, which can be bound to new values for the duration of a dynamic extent. The set of all parameter bindings at a given time is called the *dynamic environment*.

(make-parameter *init*) procedure (make-parameter *init converter*) procedure Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially, this value is the value of (*converter init*), or of *init* if the conversion procedure *converter* is not specified. The associated value can be temporarily changed using parameterize, which is described below.

The effect of passing arguments to a parameter object is implementation-dependent.

(parameterize ((param1 value1) ...) syntax)

Syntax: Both param1 and value1 are expressions.

It is an error if the value of any param expression is not a parameter object.

Semantics: A parameterize expression is used to change the values returned by specified parameter objects during the evaluation of the body.

The param and value expressions are evaluated in an unspecified order. The body is evaluated in a dynamic environment in which calls to the parameters return the results of passing the corresponding values to the conversion procedure specified when the parameters were created. Then the previous values of the parameters are restored without passing them to the conversion procedure. The results of the last expression in the body are returned as the results of the entire parameterize expression.

Note: If the conversion procedure is not idempotent, the results of (parameterize ((x (x))) ...), which appears to bind the parameter x to its current value, might not be what the user expects.

If an implementation supports multiple threads of execution, then parameterize must not change the associated values of any parameters in any thread other than the current thread and threads created inside body.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

```
(define radix
  (make-parameter
   10
   (lambda (x)
     (if (and (exact-integer? x) (<= 2 x 16))
         x
         (error "invalid radix")))))

(define (f n) (number->string n (radix)))

(f 12) ;; => "12"
(parameterize ((radix 2))
  (f 12)) ;; => "1100"
(f 12) ;; => "12"

(radix 16) ;; => unspecified

(parameterize ((radix 0))
  (f 12)) ;; => error
```

Exception handling (guard (variable syntax `cond clause_2 \ldots\,) )`
Syntax: Each cond clause is as in the specification of cond.

Semantics: The body is evaluated with an exception handler that binds the raised object (see `raise` in section [exceptionsection]) to variable and, within the scope of that binding, evaluates the clauses as if they were the clauses of a cond expression. That implicit cond expression is evaluated with the continuation and dynamic environment of the guard expression. If every cond clause's test evaluates to #f and there is no else clause, then raise-continuable is invoked on the raised object within the dynamic environment of the original call to raise or raise-continuable, except that the current exception handler is that of the guard expression.

See section [exceptionsection] for a more complete discussion of exceptions.

```
(guard (condition
        ((assq 'a condition) => cdr)
        ((assq 'b condition)))
       (raise (list (cons 'a 42))))
;; => 42

(guard (condition
        ((assq 'a condition) => cdr)
        ((assq 'b condition)))
       (raise (list (cons 'b 23))))
;; => (b . 23)
```

Quasiquotation (quasiquote *qq template*) syntax `qq template` `syntax unquote` `auxiliary syntax` `` `' `` `auxiliary syntax unquote-splicing` `auxiliary syntax` `` `' `` `auxiliary syntax "Quasiquote"` expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance. If no commas appear within the qq template, the result of evaluating ``qq template is equivalent to the result of evaluating'qq template. If a comma appears within the qq template, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed without intervening whitespace by a commercial at-sign (',@'), then it is an error if the following expression does not evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign normally appears only within a list or vector qq template.

Note: In order to unquote an identifier beginning with @, it is necessary to use either an explicit unquote or to put whitespace after the comma, to avoid colliding with the comma at-sign sequence.

```
`(list ,(+ 1 2) 4) ;; => (list 3 4)
(let ((name 'a)) `(list ,name ',name))
;; => (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
;; => (a 3 4 5 6 b)
`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
;; => ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
;; => #(10 5 2 4 3 8)
(let ((foo '(foo bar)) (@baz 'baz))
  `(list ,@foo , @baz))
;; => (list foo bar baz)
```

Quasiquote expressions can be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost quasiquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
;; => (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
;; => (a `(b ,x ,'y d) e)
```

A quasiquote expression may return either newly allocated, mutable objects or literal structure for any structure that is constructed at run time during the evaluation of the expression. Portions that do not need to be rebuilt are always literal. Thus,

```
(let ((a 3)) `((1 2) ,a ,4 ,'five 6))
```

may be treated as equivalent to either of the following expressions:

```
`((1 2) 3 4 five 6)

(let ((a 3))
  (cons '(1 2)
        (cons a (cons 4 (cons 'five '(6))))))
```

However, it is not equivalent to this expression:

```
(let ((a 3)) (list (list 1 2) a 4 'five 6))
```

The two notations ‘qq template and(quasiquote qq template)are identical in all respects. ,expression is identical to (unquote expression), and ,@expression is identical to (unquote-splicing expression). Thewrite‘ procedure may output either format.

```
(quasiquote (list (unquote (+ 1 2)) 4))
;; => (list 3 4)
```

```
'(quasiquote (list (unquote (+ 1 2)) 4))
;; => `(list ,(+ 1 2) 4)
;; i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

It is an error if any of the identifiers quasiquote, unquote, or unquote-splicing appear in positions within a qq template otherwise than as described above.

Case-lambda (case-lambda *clause …*) case-lambda library syntax *Syntax:* Each clause is of the form (formals body), where formals and body have the same syntax as in a lambda expression.

Semantics: A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first clause for which the arguments agree with formals is selected, where agreement is specified as for the formals of a lambda expression. The variables of formals are bound to fresh locations, the values of the arguments are stored in those locations, the body is evaluated in the extended environment, and the results of body are returned as the results of the procedure call.

It is an error for the arguments not to agree with the formals of any clause.

```
(define range
  (case-lambda
   ((e) (range 0 e))
   ((b e) (do ((r '() (cons e r))
               (e (- e 1) (- e 1)))
              ((< e b) r)))))

(range 3) ;; => (0 1 2)
(range 3 5) ;; => (3 4)
```

Macros

Scheme programs can define and use new derived expression types, called *macros*. Program-defined expression types have the syntax

```
(\hyper{keyword} {\hyper{datum}} ...)
```

where keyword is an identifier that uniquely determines the expression type. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of the datums, and their syntax, depends on the expression type.

Each instance of a macro is called a *use* of the macro. The set of rules that specifies how a use of a macro is transcribed into a more primitive expression is called the *transformer* of the macro.

The macro definition facility consists of two parts:

- A set of expressions used to establish that certain identifiers are macro keywords, associate them with macro transformers, and control the scope within which a macro is defined, and
- a pattern language for specifying macro transformers.

The syntactic keyword of a macro can shadow variable bindings, and local variable bindings can shadow syntactic bindings. Two mechanisms are provided to prevent unintended conflicts:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers. Note that a global variable definition may or may not introduce a binding; see section [defines].
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that surround the use of the macro.

In consequence, all macros defined using the pattern language are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping.

Implementations may provide macro facilities of other types.

Binding constructs for syntactic keywords The let-syntax and letrec-syntax binding constructs are analogous to let and letrec, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords can also be bound globally or locally with define-syntax; see section [define-syntax].

(let-syntax *bindings body*) syntax *Syntax:* Bindings has the form

```
((\hyper{keyword} \hyper{transformer spec}) \dotsfoo)
```

Each keyword is an identifier, each transformer spec is an instance of syntax-rules, and body is a sequence of zero or more definitions followed by one or more expressions. It is an error for a keyword to appear more than once in the list of keywords being bound.

Semantics: The body is expanded in the syntactic environment obtained by extending the syntactic environment of the let-syntax expression with macros whose keywords are the keywords, bound to the specified transformers. Each binding of a keyword has body as its region.

```
(let-syntax ((given-that (syntax-rules ()
                           ((given-that test stmt1 stmt2 ...)
                            (if test
                                (begin stmt1
                                       stmt2 ...))))))
  (let ((if #t))
    (given-that if (set! if 'now))
```

```
    if)) ;; => now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m)))) ;; => outer
```

(letrec-syntax *bindings body*) syntax *Syntax:* Same as for let-syntax.

Semantics: The body is expanded in the syntactic environment obtained by extending the syntactic environment of the letrec-syntax expression with macros whose keywords are the keywords, bound to the specified transformers. Each binding of a keyword has the transformer specs as well as the body within its region, so the transformers can transcribe expressions into uses of the macros introduced by the letrec-syntax expression.

```
(letrec-syntax
    ((my-or (syntax-rules ()
              ((my-or) #f)
              ((my-or e) e)
              ((my-or e1 e2 ...)
               (let ((temp e1))
                 (if temp
                     temp
                     (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y))) ;; => 7
```

Pattern language A transformer spec has one of the following forms:

(syntax-rules (pattern literal …) syntax `\ldots\,`) (syntax-rules ellipsis (pattern literal …) syntax `\ldots\,`) __ auxiliary syntax … auxiliary syntax *Syntax:* It is an error if any of the pattern literals, or the ellipsis in the second form, is not an identifier. It is also an error if syntax rule is not of the form

`(\hyper{pattern} \hyper{template})`

The pattern in a syntax rule is a list pattern whose first element is an identifier.

A pattern is either an identifier, a constant, or one of the following

`(\hyper{pattern} \ldots)`

```
(\hyper{pattern} \hyper{pattern} \ldots . \hyper{pattern})
(\hyper{pattern} \ldots \hyper{pattern} \hyper{ellipsis} \hyper{pattern} \ldots)
(\hyper{pattern} \ldots \hyper{pattern} \hyper{ellipsis} \hyper{pattern} \ldots
  . \hyper{pattern})
#(\hyper{pattern} \ldots)
#(\hyper{pattern} \ldots \hyper{pattern} \hyper{ellipsis} \hyper{pattern} \ldots)
```

and a template is either an identifier, a constant, or one of the following

```
(\hyper{element} \ldots)
(\hyper{element} \hyper{element} \ldots . \hyper{template})
(\hyper{ellipsis} \hyper{template})
#(\hyper{element} \ldots)
```

where an element is a template optionally followed by an ellipsis. An ellipsis is the identifier specified in the second form of syntax-rules, or the default identifier … (three consecutive periods) otherwise.

Semantics: An instance of syntax-rules produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by syntax-rules is matched against the patterns contained in the syntax rules, beginning with the leftmost syntax rule. When a match is found, the macro use is transcribed hygienically according to the template.

An identifier appearing within a pattern can be an underscore (_), a literal identifier listed in the list of pattern literals, or the ellipsis. All other identifiers appearing within a pattern are *pattern variables*.

The keyword at the beginning of the pattern in a syntax rule is not involved in the matching and is considered neither a pattern variable nor a literal identifier.

Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is an error for the same pattern variable to appear more than once in a pattern.

Underscores also match arbitrary input elements but are not pattern variables and so cannot be used to refer to those elements. If an underscore appears in the pattern literals list, then that takes precedence and underscores in the pattern match as literals. Multiple underscores can appear in a pattern.

Identifiers that appear in (pattern literal …) are interpreted as literal identifiers to be matched against corresponding elements of the input. An element in the input matches a literal identifier if and only if it is an identifier and either both its occurrence in the macro expression and its occurrence in the macro definition have the same lexical binding, or the two identifiers are the same and both have no lexical binding.

A subpattern followed by ellipsis can match zero or more elements of the input, unless ellipsis appears in the pattern literals, in which case it is matched as a literal.

More formally, an input expression E matches a pattern P if and only if:

- P is an underscore (_).
- P is a non-literal identifier; or
- P is a literal identifier and E is an identifier with the same binding; or
- P is a list $(P1\ \ldots\ Pn)$ and E is a list of n elements that match $P1$ through Pn, respectively; or
- P is an improper list $(P1\ P2\ \ldots\ Pn\ .\ Pn+1)$ and E is a list or improper list of n or more elements that match $P1$ through Pn, respectively, and whose nth tail matches $Pn+1$; or
- P is of the form $(P1\ \ldots\ Pk\ Pe$ ellipsis $Pm+1\ \ldots\ Pn)$ where E is a proper list of n elements, the first k of which match $P1$ through Pk, respectively, whose next $m-k$ elements each match Pe, whose remaining $n-m$ elements match $Pm+1$ through Pn; or
- P is of the form $(P1\ \ldots\ Pk\ Pe$ ellipsis $Pm+1\ \ldots\ Pn\ .\ Px)$ where E is a list or improper list of n elements, the first k of which match $P1$ through Pk, whose next $m-k$ elements each match Pe, whose remaining $n-m$ elements match $Pm+1$ through Pn, and whose nth and final cdr matches Px; or
- P is a vector of the form $\#(P1\ \ldots\ Pn)$ and E is a vector of n elements that match $P1$ through Pn; or
- P is of the form $\#(P1\ \ldots\ Pk\ Pe$ ellipsis $Pm+1\ \ldots Pn)$ where E is a vector of n elements the first k of which match $P1$ through Pk, whose next $m-k$ elements each match Pe, and whose remaining $n-m$ elements match $Pm+1$ through Pn; or
- P is a constant and E is equal to P in the sense of the equal? procedure.

It is an error to use a macro keyword, within the scope of its binding, in an expression that does not match any of the patterns.

When a macro use is transcribed according to the template of the matching syntax rule, pattern variables that occur in the template are replaced by the elements they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier ellipsis are allowed only in subtemplates that are followed by as many instances of ellipsis. They are replaced in the output by all of the elements they match in the input, distributed as indicated. It is an error if the output cannot be built up as specified.

Identifiers that appear in the template but are not pattern variables or the identifier ellipsis are inserted into the output as literal identifiers. If a literal identifier is inserted as a free identifier then it refers to the binding of that identifier within whose scope the instance of syntax-rules appears. If a literal

identifier is inserted as a bound identifier then it is in effect renamed to prevent inadvertent captures of free identifiers.

A template of the form (ellipsis template) is identical to template, except that ellipses within the template have no special meaning. That is, any ellipses contained within template are treated as ordinary identifiers. In particular, the template (ellipsis ellipsis) produces a single ellipsis. This allows syntactic abstractions to expand into code containing ellipses.

```
(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
     (define-syntax name
       (syntax-rules ()
         ((name expr (... ...))
          (begin expr (... ...))))))))

(be-like-begin sequence)
(sequence 1 2 3 4) ;; => 4
```

As an example, if `let` and `cond` are defined as in section [derivedsection] then they are hygienic (as required) and the following is not an error.

```
(let ((=> #f))
  (cond (#t => 'ok))) ;; => ok
```

The macro transformer for cond recognizes => as a local variable, and hence an expression, and not as the base identifier =>, which the macro transformer treats as a syntactic keyword. Thus the example expands into

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

instead of

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

which would result in an invalid procedure call.

Signaling errors in macro transformers (syntax-error message args ...) syntax syntax-error behaves similarly to error ([exceptionsection]) except that implementations with an expansion pass separate from evaluation should signal an error as soon as syntax-error is expanded. This can be used as a syntax-rules template for a pattern that is an invalid use of the macro, which can provide more descriptive error messages. message is a string literal, and args arbitrary expressions providing additional information. Applications cannot count on being able to catch syntax errors with exception handlers or guards.

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
      body1 body2 ...)
     (syntax-error
      "expected an identifier but got"
      (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

Program structure

Programs

A Scheme program consists of one or more import declarations followed by a sequence of expressions and definitions. Import declarations specify the libraries on which a program or library depends; a subset of the identifiers exported by the libraries are made available to the program. Expressions are described in chapter [expressionchapter]. Definitions are either variable definitions, syntax definitions, or record-type definitions, all of which are explained in this chapter. They are valid in some, but not all, contexts where expressions are allowed, specifically at the outermost level of a program and at the beginning of a body.

At the outermost level of a program, `(begin expression or definition_1 \ldots\,)` is equivalent to the sequence of expressions and definitions in the `begin`. Similarly, in a body, `(begin definition_1 \ldots\,)` is equivalent to the sequence definition1 … . Macros can expand into such begin forms. For the formal definition, see [sequencing].

Import declarations and definitions cause bindings to be created in the global environment or modify the value of existing global bindings. The initial environment of a program is empty, so at least one import declaration is needed to introduce initial bindings.

Expressions occurring at the outermost level of a program do not create any bindings. They are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

Programs and libraries are typically stored in files, although in some implementations they can be entered interactively into a running Scheme system. Other paradigms are possible. Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.

Import declarations

An import declaration takes the following form:

`(import <import-set> ...)`

An import declaration provides a way to import identifiers exported by a library. Each import set names a set of bindings from a library and possibly specifies local names for the imported bindings. It takes one of the following forms:

- `library name`
- `(only import set identifier \ldots\,)`
- `(except import set identifier \ldots\,)`
- `(prefix import set identifier)`
- `(rename import set  (identifier_1 identifier_2) \ldots\,)`

In the first form, all of the identifiers in the named library's export clauses are imported with the same names (or the exported names if exported with `rename`). The additional import set forms modify this set as follows:

- `only` produces a subset of the given import set including only the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- `except` produces a subset of the given import set, excluding the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
- `rename` modifies the given import set, replacing each instance of identifier1 with identifier2. It is an error if any of the listed identifier1s are not found in the original set.
- `prefix` automatically renames all identifiers in the given import set, prefixing each with the specified identifier.

In a program or library declaration, it is an error to import the same identifier more than once with different bindings, or to redefine or mutate an imported binding with a definition or with set!, or to refer to an identifier before it is imported. However, a REPL should permit these actions.

Variable definitions

A variable definition binds one or more identifiers and specifies an initial value for each of them. The simplest kind of variable definition takes one of the following forms:

- `(define variable expression)`
- `(define (variable formals) body)`

  Formals are either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression). This form is equivalent to

```
(define <variable>
  (lambda (<formals>) <body>))
```

- `(define (variable . formal) body)`

  Formal is a single variable. This form is equivalent to

  ```
  (define <variable>
    (lambda <formal> <body>))
  ```

Top level definitions At the outermost level of a program, a definition

```
(define <variable> \hyper{expression})
```

has essentially the same effect as the assignment expression

```
(set!}\ <variable> \hyper{expression})
```

if variable is bound to a non-syntax value. However, if variable is not bound, or is a syntactic keyword, then the definition will bind variable to a new location before performing the assignment, whereas it would be an error to perform a set! on an unbound variable.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3) ;; => 6
(define first car)
(first '(1 2)) ;; => 1
```

Internal definitions Definitions can occur at the beginning of a body (that is, the body of a `lambda`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `let-syntax`, `letrec-syntax`, `parameterize`, `guard`, or `case-lambda`). Note that such a body might not be apparent until after expansion of other syntax. Such definitions are known as *internal definitions* as opposed to the global definitions described above. The variables defined by internal definitions are local to the body. That is, variable is bound rather than assigned, and the region of the binding is the entire body. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3))) ;; => 45
```

An expanded body containing internal definitions can always be converted into a completely equivalent letrec* expression. For example, the let expression in the above example is equivalent to

```
(let ((x 5))
  (letrec* ((foo (lambda (y) (bar x y)))
            (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

Just as for the equivalent letrec* expression, it is an error if it is not possible to evaluate each expression of every internal definition in a body without assigning or referring to the value of the corresponding variable or the variable of any of the definitions that follow it in body.

It is an error to define the same identifier more than once in the same body.

Wherever an internal definition can occur, `(begin definition_1 \ldots\,)` is equivalent to the sequence of definitions that form the body of the `begin`.

Multiple-value definitions Another kind of definition is provided by define-values, which creates multiple definitions from a single expression returning multiple values. It is allowed wherever define is allowed.

(define-values *formals expression*) syntax

It is an error if a variable appears more than once in the set of formals.

Semantics: Expression is evaluated, and the formals are bound to the return values in the same way that the formals in a lambda expression are matched to the arguments in a procedure call.

```
(define-values (x y) (exact-integer-sqrt 17))
(list x y) ;; => (4 1)

(let ()
  (define-values (x y) (values 1 2))
  (+ x y)) ;; => 3
```

Syntax definitions

Syntax definitions have this form:

```
(define-syntax keyword transformer spec)
```

Keyword is an identifier, and the transformer spec is an instance of `syntax-rules`. Like variable definitions, syntax definitions can appear at the outermost level or nested within a `body`.

If the define-syntax occurs at the outermost level, then the global syntactic environment is extended by binding the keyword to the specified transformer, but previous expansions of any global binding for keyword remain unchanged. Otherwise, it is an *internal syntax definition*, and is local to the body in which it is defined. Any use of a syntax keyword before its corresponding definition is an error. In particular, a use that precedes an inner definition will not apply an outer definition.

```
(let ((x 1) (y 2))
  (define-syntax swap!
    (syntax-rules ()
      ((swap! a b)
```

```
      (let ((tmp a))
        (set! a b)
        (set! b tmp)))))
  (swap! x y)
  (list x y)) ;; => (2 1)
```

Macros can expand into definitions in any context that permits them. However, it is an error for a definition to define an identifier whose binding has to be known in order to determine the meaning of the definition itself, or of any preceding definition that belongs to the same group of internal definitions. Similarly, it is an error for an internal definition to define an identifier whose binding has to be known in order to determine the boundary between the internal definitions and the expressions of the body it belongs to. For example, the following are errors:

```
(define define 3)

(begin (define begin list))

(let-syntax
    ((foo (syntax-rules ()
            ((foo (proc args ...) body ...)
             (define proc
               (lambda (args ...)
                 body ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

Record-type definitions

Record-type definitions are used to introduce new data types, called *record types*. Like other definitions, they can appear either at the outermost level or in a body. The values of a record type are called *records* and are aggregations of zero or more *fields*, each of which holds a single location. A predicate, a constructor, and field accessors and mutators are defined for each record type.

(define-record-type name syntax)

Syntax: name and pred are identifiers. The constructor is of the form

`(\hyper{constructor name} \hyper{field name} ...)`

and each field is either of the form

`(\hyper{field name} \hyper{accessor name})`

or of the form

(\hyper{field name} \hyper{accessor name} \hyper{modifier name})

It is an error for the same identifier to occur more than once as a field name. It is also an error for the same identifier to occur more than once as an accessor or mutator name.

The define-record-type construct is generative: each use creates a new record type that is distinct from all existing types, including Scheme's predefined types and other record types — even record types of the same name or structure.

An instance of define-record-type is equivalent to the following definitions:

- name is bound to a representation of the record type itself. This may be a run-time object or a purely syntactic representation. The representation is not utilized in this report, but it serves as a means to identify the record type for use by further language extensions.
- constructor name is bound to a procedure that takes as many arguments as there are field names in the (constructor name ...) subexpression and returns a new record of type name. Fields whose names are listed with constructor name have the corresponding argument as their initial value. The initial values of all other fields are unspecified. It is an error for a field name to appear in constructor but not as a field name.
- pred is bound to a predicate that returns #t when given a value returned by the procedure bound to constructor name and #f for everything else.
- Each accessor name is bound to a procedure that takes a record of type name and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each modifier name is bound to a procedure that takes a record of type name and a value which becomes the new value of the corresponding field; an unspecified value is returned. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

For instance, the following record-type definition

```
(define-record-type <pare>
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines kons to be a constructor, kar and kdr to be accessors, set-kar! to be a modifier, and pare? to be a predicate for instances of <pare>.

```
(pare? (kons 1 2)) ;; => #t
(pare? (cons 1 2)) ;; => #f
(kar (kons 1 2)) ;; => 1
(kdr (kons 1 2)) ;; => 2
```

```
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k)) ;; => 3
```

Libraries

Libraries provide a way to organize Scheme programs into reusable parts with explicitly defined interfaces to the rest of the program. This section defines the notation and semantics for libraries.

Library Syntax A library definition takes the following form:

```
(define-library \hyper{library name}
  \hyper{library declaration} ...)
```

library name is a list whose members are identifiers and exact non-negative integers. It is used to identify the library uniquely when importing from other programs or libraries. Libraries whose first identifier is scheme are reserved for use by this report and future versions of this report. Libraries whose first identifier is srfi are reserved for libraries implementing Scheme Requests for Implementation. It is inadvisable, but not an error, for identifiers in library names to contain any of the characters | ' ? * < " : > + [ ] / or control characters after escapes are expanded.

A library declaration is any of:

- `(export export spec \ldots\,)`
- `(import import set \ldots\,)`
- `(begin command or definition \ldots\,)`
- `(include filename_1 filename_2 \ldots\,)`
- `(include-ci filename_1 filename_2 \ldots\,)`
- `(include-library-declarations filename_1 filename_2 \ldots\,)`
- `(cond-expand ce-clause_1 ce-clause_2 \ldots\,)`

An `export` declaration specifies a list of identifiers which can be made visible to other libraries or programs. An export spec takes one of the following forms:

- identifier
- `(rename identifier_1 identifier_2)`

In an export spec, an identifier names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A `rename` spec exports the binding defined within or imported into the library and named by identifier1 in each `(identifier_1 identifier_2)` pairing, using identifier2 as the external name.

An `import` declaration provides a way to import the identifiers exported by another library. It has the same syntax and semantics as an import declaration used in a program or at the REPL (see section [import]).

The `begin`, `include`, and `include-ci` declarations are used to specify the body of the library. They have the same syntax and semantics as the corresponding expression types. This form of begin is analogous to, but not the same as, the two types of begin defined in section [sequencing].

The `include-library-declarations` declaration is similar to `include` except that the contents of the file are spliced directly into the current library definition. This can be used, for example, to share the same `export` declaration among multiple libraries as a simple form of library interface.

The `cond-expand` declaration has the same syntax and semantics as the `cond-expand` expression type, except that it expands to spliced-in library declarations rather than expressions enclosed in begin.

One possible implementation of libraries is as follows: After all `cond-expand` library declarations are expanded, a new environment is constructed for the library consisting of all imported bindings. The expressions from all `begin`, `include` and `include-ci` library declarations are expanded in that environment in the order in which they occur in the library. Alternatively, `cond-expand` and `import` declarations may be processed in left to right order interspersed with the processing of other declarations, with the environment growing as imported bindings are added to it by each `import` declaration.

When a library is loaded, its expressions are executed in textual order. If a library's definitions are referenced in the expanded form of a program or library body, then that library must be loaded before the expanded program or library body is evaluated. This rule applies transitively. If a library is imported by more than one program or library, it may possibly be loaded additional times.

Similarly, during the expansion of a library (foo), if any syntax keywords imported from another library (bar) are needed to expand the library, then the library (bar) must be expanded and its syntax definitions evaluated before the expansion of (foo).

Regardless of the number of times that a library is loaded, each program or library that imports bindings from a library must do so from a single loading of that library, regardless of the number of import declarations in which it appears. That is, (import (only (foo) a)) followed by (import (only (foo) b)) has the same effect as (import (only (foo) a b)).

Library example The following example shows how a program can be divided into libraries plus a relatively small main program . If the main program is entered into a REPL, it is not necessary to import the base library.

```
(define-library (example grid)
  (export make rows cols ref each
          (rename put! set!))
  (import (scheme base))
  (begin
    ;; Create an NxM grid.
    (define (make n m)
      (let ((grid (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((= i n) grid)
          (let ((v (make-vector m \sharpfalse{})))
            (vector-set! grid i v)))))
    (define (rows grid)
      (vector-length grid))
    (define (cols grid)
      (vector-length (vector-ref grid 0)))
    ;; Return \sharpfalse{} if out of range.
    (define (ref grid n m)
      (and (< -1 n (rows grid))
           (< -1 m (cols grid))
           (vector-ref (vector-ref grid n) m)))
    (define (put! grid n m v)
      (vector-set! (vector-ref grid n) m v))
    (define (each grid proc)
      (do ((j 0 (+ j 1)))
          ((= j (rows grid)))
        (do ((k 0 (+ k 1)))
            ((= k (cols grid)))
          (proc j k (ref grid j k)))))))

(define-library (example life)
  (export life)
  (import (except (scheme base) set!)
          (scheme write)
          (example grid))
  (begin
    (define (life-count grid i j)
      (define (count i j)
        (if (ref grid i j) 1 0))
      (+ (count (- i 1) (- j 1))
         (count (- i 1) j)
         (count (- i 1) (+ j 1))
         (count i (- j 1))
         (count i (+ j 1))
         (count (+ i 1) (- j 1))
         (count (+ i 1) j)
```

```
          (count (+ i 1) (+ j 1))))
    (define (life-alive? grid i j)
      (case (life-count grid i j)
        ((3) \sharptrue{})
        ((2) (ref grid i j))
        (else \sharpfalse{})))
    (define (life-print grid)
      (display "\x1B;[1H\x1B;[J")  ; clear vt100
      (each grid
            (lambda (i j v)
              (display (if v "*" " "))
              (when (= j (- (cols grid) 1))
                (newline)))))
    (define (life grid iterations)
      (do ((i 0 (+ i 1))
           (grid0 grid grid1)
           (grid1 (make (rows grid) (cols grid))
                  grid0))
          ((= i iterations))
        (each grid0
              (lambda (j k v)
                (let ((a (life-alive? grid0 j k)))
                  (set! grid1 j k a))))
        (life-print grid1))))

;; Main program.
(import (scheme base)
        (only (example life) life)
        (rename (prefix (example grid) grid-)
                (grid-make make-grid)))

;; Initialize a grid with a glider.
(define grid (make-grid 24 24))
(grid-set! grid 1 1 \sharptrue{})
(grid-set! grid 2 2 \sharptrue{})
(grid-set! grid 3 0 \sharptrue{})
(grid-set! grid 3 1 \sharptrue{})
(grid-set! grid 3 2 \sharptrue{})

;; Run for 80 iterations.
(life grid 80)
```

The REPL

Implementations may provide an interactive session called a *REPL* (Read-Eval-Print Loop), where import declarations, expressions and definitions can be en-

tered and evaluated one at a time. For convenience and ease of use, the global Scheme environment in a REPL must not be empty, but must start out with at least the bindings provided by the base library. This library includes the core syntax of Scheme and generally useful procedures that manipulate data. For example, the variable abs is bound to a procedure of one argument that computes the absolute value of a number, and the variable + is bound to a procedure that computes sums. The full list of (scheme base) bindings can be found in Appendix [stdlibraries].

Implementations may provide an initial REPL environment which behaves as if all possible variables are bound to locations, most of which contain unspecified values. Top level REPL definitions in such an implementation are truly equivalent to assignments, unless the identifier is defined as a syntax keyword.

An implementation may provide a mode of operation in which the REPL reads its input from a file. Such a file is not, in general, the same as a program, because it can contain import declarations in places other than the beginning.

Standard procedures

This chapter describes Scheme's built-in procedures.

The procedures force, promise?, and make-promise are intimately associated with the expression types delay and delay-force, and are described with them in section [force]. In the same way, the procedure make-parameter is intimately associated with the expression type parameterize, and is described with it in section [make-parameter].

A program can use a global variable definition to bind any variable. It may subsequently alter any such binding by an assignment (see section [assignment]). These operations do not modify the behavior of any procedure defined in this report or imported from a library (see section [libraries]). Altering any global binding that has not been introduced by a definition has an unspecified effect on the behavior of the procedures defined in this chapter.

When a procedure is said to return a *newly allocated* object, it means that the locations in the object are fresh.

Equivalence predicates

A *predicate* is a procedure that always returns a boolean value (#t or #f). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation; it is symmetric, reflexive, and transitive. Of the equivalence predicates described in this section, eq? is the finest or most discriminating, equal? is the coarsest, and eqv? is slightly less discriminating than eq?.

(eqv? **obj1 obj2**) procedure The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if *obj1* and *obj2* are normally regarded

as the same object. This relation is left slightly open to interpretation, but the following partial specification of eqv? holds for all implementations of Scheme.

The eqv? procedure returns #t if:

- *obj1* and *obj2* are both #t or both #f.
- *obj1* and *obj2* are both symbols and are the same symbol according to the symbol=? procedure (section [symbolsection]).
- *obj1* and *obj2* are both exact numbers and are numerically equal (in the sense of =).
- *obj1* and *obj2* are both inexact numbers such that they are numerically equal (in the sense of =) and they yield the same results (in the sense of eqv?) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures, provided it does not result in a NaN value.
- *obj1* and *obj2* are both characters and are the same character according to the char=? procedure (section [charactersection]).
- *obj1* and *obj2* are both the empty list.
- *obj1* and *obj2* are pairs, vectors, bytevectors, records, or strings that denote the same location in the store (section [storagemodel]).
- *obj1* and *obj2* are procedures whose location tags are equal (section [lambda]).

The eqv? procedure returns #f if:

- *obj1* and *obj2* are of different types (section [disjointness]).
- one of *obj1* and *obj2* is #t but the other is #f.
- *obj1* and *obj2* are symbols but are not the same symbol according to the symbol=? procedure (section [symbolsection]).
- one of *obj1* and *obj2* is an exact number but the other is an inexact number.
- *obj1* and *obj2* are both exact numbers and are numerically unequal (in the sense of =).
- *obj1* and *obj2* are both inexact numbers such that either they are numerically unequal (in the sense of =), or they do not yield the same results (in the sense of eqv?) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures, provided it does not result in a NaN value. As an exception, the behavior of eqv? is unspecified when both *obj1* and *obj2* are NaN.
- *obj1* and *obj2* are characters for which the char=? procedure returns #f.
- one of *obj1* and *obj2* is the empty list but the other is not.

- *obj1* and *obj2* are pairs, vectors, bytevectors, records, or strings that denote distinct locations.
- *obj1* and *obj2* are procedures that would behave differently (return different values or have different side effects) for some arguments.

```
(eqv? 'a 'a) ;; => #t
(eqv? 'a 'b) ;; => #f
(eqv? 2 2) ;; => #t
(eqv? 2 2.0) ;; => #f
(eqv? '() '()) ;; => #t
(eqv? 100000000 100000000) ;; => #t
(eqv? 0.0 +nan.0) ;; => #f
(eqv? (cons 1 2) (cons 1 2)) ;; => #f
(eqv? (lambda () 1)
      (lambda () 2)) ;; => #f
(let ((p (lambda (x) x)))
  (eqv? p p)) ;; => #t
(eqv? #f 'nil) ;; => #f
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of eqv?. All that can be said about such cases is that the value returned by eqv? must be a boolean.

```
(eqv? "" "") ;; => unspecified
(eqv? '#() '#()) ;; => unspecified
(eqv? (lambda (x) x)
      (lambda (x) x)) ;; => unspecified
(eqv? (lambda (x) x)
      (lambda (y) y)) ;; => unspecified
(eqv? 1.0e0 1.0f0) ;; => unspecified
(eqv? +nan.0 +nan.0) ;; => unspecified
```

Note that (eqv? 0.0 -0.0) will return #f if negative zero is distinguished, and #t if negative zero is not distinguished.

The next set of examples shows the use of eqv? with procedures that have local state. The gen-counter procedure must return a distinct procedure every time, since each procedure has its own internal counter. The gen-loser procedure, however, returns operationally equivalent procedures each time, since the local state does not affect the value or side effects of the procedures. However, eqv? may or may not detect this equivalence.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g)) ;; => #t
```

```
(eqv? (gen-counter) (gen-counter))
;; => #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g)) ;; => #t
(eqv? (gen-loser) (gen-loser))
;; => unspecified

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
;; => unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
;; => #f
```

Since it is an error to modify constant objects (those returned by literal expressions), implementations may share structure between constants where appropriate. Thus the value of eqv? on constants is sometimes implementation-dependent.

```
(eqv? '(a) '(a)) ;; => unspecified
(eqv? "a" "a") ;; => unspecified
(eqv? '(b) (cdr '(a b))) ;; => unspecified
(let ((x '(a)))
  (eqv? x x)) ;; => #t
```

The above definition of eqv? allows implementations latitude in their treatment of procedures and literals: implementations may either detect or fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

Note: If inexact numbers are represented as IEEE binary floating-point numbers, then an implementation of eqv? that simply compares equal-sized inexact numbers for bitwise equality is correct by the above definition.

(eq? **obj1 obj2**) procedure The eq? procedure is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?. It must always return #f when eqv? also would, but may return #f in some cases where eqv? would return #t.

On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, eq? and eqv? are guaranteed to have the

same behavior. On procedures, eq? must return true if the arguments' location tags are equal. On numbers and characters, eq?'s behavior is implementation-dependent, but it will always return either true or false. On empty strings, empty vectors, and empty bytevectors, eq? may also behave differently from eqv?.

```
(eq? 'a 'a) ;; => #t
(eq? '(a) '(a)) ;; => unspecified
(eq? (list 'a) (list 'a)) ;; => #f
(eq? "a" "a") ;; => unspecified
(eq? "" "") ;; => unspecified
(eq? '() '()) ;; => #t
(eq? 2 2) ;; => unspecified
(eq? #\A #\A) ;; => unspecified
(eq? car car) ;; => #t
(let ((n (+ 2 3)))
  (eq? n n)) ;; => unspecified
(let ((x '(a)))
  (eq? x x)) ;; => #t
(let ((x '#()))
  (eq? x x)) ;; => #t
(let ((p (lambda (x) x)))
  (eq? p p)) ;; => #t
```

Rationale: It will usually be possible to implement eq? much more efficiently than eqv?, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it is not always possible to compute eqv? of two numbers in constant time, whereas eq? implemented as pointer comparison will always finish in constant time.

(equal? **obj1 obj2**) procedure The equal? procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning #t when the unfoldings of its arguments into (possibly infinite) trees are equal (in the sense of equal?) as ordered trees, and #f otherwise. It returns the same as eqv? when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are eqv?, they must be equal? as well. In all other cases, equal? may return either #t or #f.

Even if its arguments are circular data structures, equal? must always terminate.

```
(equal? 'a 'a) ;; => #t
(equal? '(a) '(a)) ;; => #t
(equal? '(a (b) c)
        '(a (b) c)) ;; => #t
(equal? "abc" "abc") ;; => #t
(equal? 2 2) ;; => #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a)) ;; => #t
```

```
(equal? '#1=(a b . #1#)
        '#2=(a b a b . #2#)) ;; => #t
(equal? (lambda (x) x)
        (lambda (y) y)) ;; => unspecified
```

Note: A rule of thumb is that objects are generally equal? if they print the same.

Numbers

It is important to distinguish between mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types *number*, *complex*, *real*, *rational*, and *integer* to refer to both mathematical numbers and Scheme numbers.

Numerical types Mathematically, numbers are arranged into a tower of subtypes in which each level is a subset of the level above it:

n̄umber complex number real number rational number integer

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex number. The same is true of the Scheme numbers that model 3. For Scheme numbers, these types are defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`.

There is no simple relationship between a number's type and its representation inside a computer. Although most implementations of Scheme will offer at least two different representations of 3, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use multiple internal representations of numbers, this ought not to be apparent to a casual programmer writing simple programs.

Exactness It is useful to distinguish between numbers that are represented exactly and those that might not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

A Scheme number is *exact* if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is *inexact* if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property

of a number. In particular, an *exact complex number* has an exact real part and an exact imaginary part; all other complex numbers are *inexact complex numbers*.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equal. This is generally not true of computations involving inexact numbers since approximate methods such as floating-point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

Rational operations such as + should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value. However, (/ 3 4) must not return the mathematically incorrect value 0. See section [restrictions].

Except for `exact`, the operations described in this section must generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of its arguments. For example, multiplication of any number by an exact zero may produce an exact zero result, even if the other argument is inexact.

Specifically, the expression (* 0 +inf.0) may return 0, or +nan.0, or report that inexact numbers are not supported, or report that non-rational real numbers are not supported, or fail silently or noisily in other implementation-specific ways.

Implementation restrictions Implementations of Scheme are not required to implement the whole tower of subtypes given in section [numericaltypes], but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, implementations in which all numbers are real, or in which non-real numbers are always inexact, or in which exact numbers are always integer, are still quite useful.

Implementations may also support only a limited range of numbers of any type, subject to the requirements of this section. The supported range for exact numbers of any type may be different from the supported range for inexact numbers of that type. For example, an implementation that uses IEEE binary double-precision floating-point numbers to represent all its inexact real numbers may also support a practically unbounded range of exact integers and rationals while limiting the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the IEEE binary double format. Furthermore, the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation of Scheme must support exact integers throughout the range of numbers permitted as indexes of lists, vectors, bytevectors, and strings or that result from computing the length of one of these. The `length`, `vector-length`, `bytevector-length`, and `string-length` procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore, any integer constant within the index range, if expressed by an exact integer syntax, must be read as an exact integer, regardless of any implementation restrictions that apply outside this range. Finally, the procedures listed below will always return exact integer results provided all their arguments are exact integers and the mathematically expected results are representable as exact integers within the implementation:

-	*
+	abs
ceiling	denominator
exact-integer-sqrt	expt
floor	floor/
floor-quotient	floor-remainder
gcd	lcm
max	min
modulo	numerator
quotient	rationalize
remainder	round
square	truncate
truncate/	truncate-quotient
truncate-remainder	

It is recommended, but not required, that implementations support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the / procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number; such a coercion can cause an error later. Nevertheless, implementations that do not provide exact rational numbers should return inexact rational numbers rather than reporting an implementation restriction.

An implementation may use floating-point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that implementations that use floating-point representations follow the IEEE 754 standard, and that implementations using other representations should match or exceed the precision achievable using these floating-point standards . In particular, the description of transcendental functions in IEEE 754-2008 should be followed by such implementations, particularly with respect to infinities and NaNs.

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them. For example, an im-

plementation in which all numbers are real need not support the rectangular and polar notations for complex numbers. If an implementation encounters an exact numerical constant that it cannot represent as an exact number, then it may either report a violation of an implementation restriction or it may silently represent the constant by an inexact number.

Implementation extensions Implementations may provide more than one representation of floating-point numbers with differing precisions. In an implementation which does so, an inexact result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. Although it is desirable for potentially inexact operations such as sqrt to produce exact answers when applied to exact arguments, if an exact number is operated upon so as to produce an inexact result, then the most precise representation available must be used. For example, the value of (sqrt 4) should be 2, but in an implementation that provides both single and double precision floating point numbers it may be the latter but must not be the former.

It is the programmer's responsibility to avoid using inexact number objects with magnitude or significand too large to be represented in the implementation.

In addition, implementations may distinguish special numbers called positive infinity, negative infinity, NaN, and negative zero.

Positive infinity is regarded as an inexact real (but not rational) number that represents an indeterminate value greater than the numbers represented by all rational numbers. Negative infinity is regarded as an inexact real (but not rational) number that represents an indeterminate value less than the numbers represented by all rational numbers.

Adding or multiplying an infinite value by any finite real value results in an appropriately signed infinity; however, the sum of positive and negative infinities is a NaN. Positive infinity is the reciprocal of zero, and negative infinity is the reciprocal of negative zero. The behavior of the transcendental functions is sensitive to infinity in accordance with IEEE 754.

A NaN is regarded as an inexact real (but not rational) number so indeterminate that it might represent any real value, including positive or negative infinity, and might even be greater than positive infinity or less than negative infinity. An implementation that does not support non-real numbers may use NaN to represent non-real values like (sqrt -1.0) and (asin 2.0).

A NaN always compares false to any number, including a NaN. An arithmetic operation where one operand is NaN returns NaN, unless the implementation can prove that the result would be the same if the NaN were replaced by any rational number. Dividing zero by zero results in NaN unless both zeros are exact.

Negative zero is an inexact real value written -0.0 and is distinct (in the sense of eqv?) from 0.0. A Scheme implementation is not required to distinguish

negative zero. If it does, however, the behavior of the transcendental functions is sensitive to the distinction in accordance with IEEE 754. Specifically, in a Scheme implementing both complex numbers and negative zero, the branch cut of the complex logarithm function is such that (imag-part (log -1.0-0.0i)) is $-$ rather than .

Furthermore, the negation of negative zero is ordinary zero and vice versa. This implies that the sum of two or more negative zeros is negative, and the result of subtracting (positive) zero from a negative zero is likewise negative. However, numerical comparisons treat negative zero as equal to zero.

Note that both the real and the imaginary parts of a complex number can be infinities, NaNs, or negative zero.

Syntax of numerical constants The syntax of the written representations for numbers is described formally in section [numbersyntax]. Note that case is not significant in numerical constants.

A number can be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are #b (binary), #o (octal), #d (decimal), and #x (hexadecimal). With no radix prefix, a number is assumed to be expressed in decimal.

A numerical constant can be specified to be either exact or inexact by a prefix. The prefixes are #e for exact, and #i for inexact. An exactness prefix can appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant is inexact if it contains a decimal point or an exponent. Otherwise, it is exact.

In systems with inexact numbers of varying precisions it can be useful to specify the precision of a constant. For this purpose, implementations may accept numerical constants written with an exponent marker that indicates the desired precision of the inexact representation. If so, the letter s, f, d, or l, meaning *short*, *single*, *double*, or *long* precision, respectively, can be used in place of e. The default precision has at least as much precision as *double*, but implementations may allow this default to be set by the user.

```
3.14159265358979F0
;; Round to single --- 3.141593
0.6L0
;; Extend to long --- .600000000000000
```

The numbers positive infinity, negative infinity, and NaN are written +inf.0, -inf.0 and +nan.0 respectively. NaN may also be written -nan.0. The use of signs in the written representation does not necessarily reflect the underlying sign of the NaN value, if any. Implementations are not required to support these numbers, but if they do, they must do so in general conformance with IEEE 754. However, implementations are not required to support signaling NaNs, nor to provide a way to distinguish between different NaNs.

There are two notations provided for non-real complex numbers: the *rectangular notation* *a*+*b*i, where *a* is the real part and *b* is the imaginary part; and the *polar notation* *r*@ , where *r* is the magnitude and is the phase (angle) in radians. These are related by the equation *a* + *b**i = r*cos + (r*sin)i. *All of* a, b, r, *and* * are real numbers.

Numerical operations The reader is referred to section [typeconventions] for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines. The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact number. Some examples also assume that certain numerical constants written using an inexact notation can be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use IEEE binary doubles to represent inexact numbers.

(number? *obj*) procedure (complex? *obj*) procedure (real? *obj*) procedure (rational? *obj*) procedure (integer? *obj*) procedure These numerical type predicates can be applied to any kind of argument, including non-numbers. They return #t if the object is of the named type, and otherwise they return #f. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If *z* is a complex number, then (real? *z*) is true if and only if (zero? (imag-part *z*)) is true. If *x* is an inexact real number, then (integer? *x*) is true if and only if (= *x* (round *x*)).

The numbers +inf.0, -inf.0, and +nan.0 are real but not rational.

```
(complex? 3+4i) ;; => #t
(complex? 3) ;; => #t
(real? 3) ;; => #t
(real? -2.5+0i) ;; => #t
(real? -2.5+0.0i) ;; => #f
(real? #e1e10) ;; => #t
(real? +inf.0) ;; => #t
(real? +nan.0) ;; => #t
(rational? -inf.0) ;; => #f
(rational? 3.5) ;; => #t
(rational? 6/10) ;; => #t
(rational? 6/3) ;; => #t
(integer? 3+0i) ;; => #t
(integer? 3.0) ;; => #t
(integer? 8/4) ;; => #t
```

Note: The behavior of these type predicates on inexact numbers is unreliable, since any inaccuracy might affect the result.

Note: In many implementations the `complex?` procedure will be the same as `number?`, but unusual implementations may represent some irrational numbers exactly or may extend the number system to support some kind of non-complex numbers.

(exact? **z**) procedure (inexact? **z**) procedure These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of these predicates is true.

```
(exact? 3.0) ;; => #f
(exact? #e3.0) ;; => #t
(inexact? 3.) ;; => #t
```

(exact-integer? **z**) procedure Returns #t if *z* is both exact and an integer; otherwise returns #f.

```
(exact-integer? 32) ;; => #t{}
(exact-integer? 32.0) ;; => #f
(exact-integer? 32/5) ;; => #f
```

(finite? **z**) inexact library procedure The finite? procedure returns #t on all real numbers except +inf.0, -inf.0, and +nan.0, and on complex numbers if their real and imaginary parts are both finite. Otherwise it returns #f.

```
(finite? 3) ;; => #t
(finite? +inf.0) ;; => #f
(finite? 3.0+inf.0i) ;; => #f
```

(infinite? **z**) inexact library procedure The infinite? procedure returns #t on the real numbers +inf.0 and -inf.0, and on complex numbers if their real or imaginary parts or both are infinite. Otherwise it returns #f.

```
(infinite? 3) ;; => #f
(infinite? +inf.0) ;; => #t
(infinite? +nan.0) ;; => #f
(infinite? 3.0+inf.0i) ;; => #t
```

(nan? **z**) inexact library procedure The nan? procedure returns #t on +nan.0, and on complex numbers if their real or imaginary parts or both are +nan.0. Otherwise it returns #f.

```
(nan? +nan.0) ;; => #t
(nan? 32) ;; => #f
(nan? +nan.0+5.0i) ;; => #t
(nan? 1+2i) ;; => #f
```

(= **z*1* z*2* z*3* ...**) **procedure** (< x*1* x*2* x*3* ...) procedure (> **x*1* x*2* x*3* ...**) **procedure** (<= x*1* x*2* x*3* ...) procedure (>= **x*1* x*2* x*3* ...) procedure These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing, and #f otherwise. If any of the arguments are

+nan.0, all the predicates return #f. They do not distinguish between inexact zero and inexact negative zero.

These predicates are required to be transitive.

Note: The implementation approach of converting all arguments to inexact numbers if any argument is inexact is not transitive. For example, let big be (expt 2 1000), and assume that big is exact and that inexact numbers are represented by 64-bit IEEE binary floating point numbers. Then (= (- big 1) (inexact big)) and (= (inexact big) (+ big 1)) would both be true with this approach, because of the limitations of IEEE representations of large integers, whereas (= (- big 1) (+ big 1)) is false. Converting inexact values to exact numbers that are the same (in the sense of =) to them will avoid this problem, though special care must be taken with infinities.

Note: While it is not an error to compare inexact numbers using these predicates, the results are unreliable because a small inaccuracy can affect the result; this is especially true of `=` and `zero?`. When in doubt, consult a numerical analyst.

(zero? **z**) procedure (positive? **x**) procedure (negative? **x**) procedure (odd? **n**) procedure (even? **n**) procedure These numerical predicates test a number for a particular property, returning #t or #f. See note above.

(max ***x1 x2 ...***) **procedure (min** *x1 x2 ...*) procedure These procedures return the maximum or minimum of their arguments.

```
(max 3 4) ;; => 4     ; exact
(max 3.9 4) ;; => 4.0  ; inexact
```

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If min or max is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.

(+ ***z1 ...***) **procedure (*** *z1 ...*) procedure These procedures return the sum or product of their arguments.

```
(+ 3 4) ;; => 7
(+ 3) ;; => 3
(+) ;; => 0
(* 4) ;; => 4
(*) ;; => 1
```

(- **z**) procedure (- ***z1 z2 ...***) **procedure (/** z) **procedure (/** *z1 z2 ...*) procedure With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

It is an error if any argument of / other than the first is an exact zero. If the