The Swish Lite Library

Bob Burger

Contents

Chapter 1

Erlang Embedding

1.1 Introduction

This chapter describes a Scheme embedding of a tuples, pattern matching, and other useful concepts from the Erlang programming language [1, 2].[1] Tuple and pattern matching macros provide succinct ways of composing and decomposing data structures.

1.2 Programming Interface

1.2.1 Tuples

A *tuple* is a container of named, immutable fields implemented as a vector whose first element is the tuple name and remaining elements are the fields. Each tuple definition is a macro that provides all tuple operations using field names only, not field indices. The macro makes it easy to copy a tuple without having to specify the fields that don't change. We decided not to use the Scheme record facility because it does not provide name-based constructors, copy operators, or convenient serialization.

`(define-tuple` *name* *field* ...`)` **syntax**

expands to: a macro definition of *name* described below

The `define-tuple` macro defines a macro for creating, copying, identifying, and accessing tuple type *name*. *name* and *field* ... must be identifiers. No two field names can be the same. The following field names are reserved: `make`, `copy`, `copy*`, and `is?`.

`(`*name* `make [`*field* *value*`] ...)` **syntax**

returns: a new instance of tuple type *name* with *field* = *value* ...

The `make` form creates a new instance of the tuple type *name*. *field* bindings may appear in any order. All fields from the tuple definition must be specified.

[1] Tuples, denoted by $\{e_1, \ldots, e_n\}$ in Erlang, are implemented as vectors: `#(`e_1 ... e_n`)`. Similarly records, defined as syntactic sugar over tuples in Erlang, are implemented as syntactic sugar over vectors.

(*name field instance*) **syntax**

returns: *instance.field*

The field accessor form retrieves the value of the specified *field* of *instance*. If r = *instance* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the field accessor form if available.

(*name field*) **syntax**

returns: a procedure that, given *instance*, returns *instance.field*

The (*name field*) form expands to `(lambda (instance) (`*name field* `instance))`.

(*name* `open` *instance* [*prefix*] (*field* ...)) **syntax**

expands to: definitions for *field* ... or *prefixfield* ... described below

The `open` form defines identifier syntax for each specified *field* so that a reference to *field* expands to (*name field r*) where r is the value of *instance*. If r is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `open` form if avalable. The `open` form is equivalent to the following, except that it checks the tuple type only once:

```
(begin
  (define instance instance)
  (define-syntax field (identifier-syntax (name field instance)))
  ...)
```

The `open` form introduces definitions only for fields listed explicitly in (*field* ...). If the optional *prefix* identifier is supplied, `open` produces a definition for *prefixfield* rather than *field* for each *field* specified.

(*name* `copy` *instance* [*field value*] ...) **syntax**

returns: a new instance of tuple type *name* with *field* = *value* ... and remaining fields copied from *instance*

The `copy` form creates a copy of *instance* except that each specified *field* is set to the associated *value*. If r = *instance* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `copy` form if avalable. *field* bindings may appear in any order.

(*name* `copy*` *instance* [*field value*] ...) **syntax**

returns: a new instance of tuple type *name* with *field* = *value* ... and remaining fields copied from *instance*

The `copy*` form is like `copy` except that, within the *value* expressions, each specified *field* is bound to an identifier macro that returns the value of *instance.field*. If r = *instance* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `copy*` form if avalable. The `copy*` form is equivalent to the following, except that it checks the tuple type only once:

```
(let ([instance instance])
  (name open instance (field ...))
  (name copy instance [field value] ...))
```

<table>
<tr><th>pattern</th><th>matches</th></tr>
<tr><td>symbol</td><td>itself</td></tr>
<tr><td>number</td><td>itself</td></tr>
<tr><td>boolean</td><td>itself</td></tr>
<tr><td>character</td><td>itself</td></tr>
<tr><td>string</td><td>itself</td></tr>
<tr><td>bytevector</td><td>itself</td></tr>
<tr><td>()</td><td>itself</td></tr>
<tr><td>(p_1 . p_2)</td><td>a pair whose car matches p_1 and cdr matches p_2</td></tr>
<tr><td>#(p_1 ... p_n)</td><td>a vector of n elements whose elements match p_1 ... p_n</td></tr>
<tr><td>#!eof</td><td>a datum satisfying eof-object?</td></tr>
<tr><td>,_</td><td>any datum</td></tr>
<tr><td>,variable</td><td>any datum and binds a fresh variable to it</td></tr>
<tr><td>,@variable</td><td>any datum equal? to the bound variable</td></tr>
<tr><td>,(variable <= pattern)</td><td>any datum that matches pattern and binds a fresh variable to it</td></tr>
<tr><td colspan="2">`(type {,field | ,@field | [field pattern]} ...)</td></tr>
<tr><td></td><td>an instance of the tuple or native record type, each field of which is bound to fresh variable field or matches the corresponding pattern; ,@field is treated as [field ,@field]; type must be known at expand time</td></tr>
<tr><td>`(ext spec ...)</td><td>as specified by define-match-extension for ext</td></tr>
</table>

Table 1.1: Pattern Grammar

(*name* `is?` *x*) **syntax**

returns: a boolean

The `is?` form determines whether or not the datum x is an instance of tuple type *name*.

(*name* `is?`) **syntax**

expands to: a predicate that returns true if and only if its argument is an instance of tuple type *name*

The (*name* `is?`) form expands to (`lambda (x)` (*name* `is? x`)).

(*name* `field-index` *field*) **syntax**

expands to: an integer n, such that (`vector-ref` *instance* n) returns *instance.field*

Avoid using this form when possible, as it leaks implementation details.

1.2.2 Pattern Matching

The pattern matching syntax of Table 1.1 provides a concise and expressive way to match data structures and bind variables to parts. The `match`, `match-define`, and `match-let*` macros use this pattern language. The implementation makes a structurally recursive pass over the pattern to check for duplicate pattern variables as it emits code that matches the input against the pattern left to right.

(match *exp* **syntax**
  (<pattern> [(guard *g*)] *b1 b2* ...)
  ...)

returns: the value of the last expression *b1 b2* ... for the matched pattern

The `match` macro evaluates *exp* once and tests its value *v* against each pattern and optional guard. Each guard expression *g* is evaluated in the scope of its associated pattern variables. When *g* returns `#f`, *v* fails to match that clause. For the first pattern and guard that matches *v*, the expressions *b1 b2* ... are evaluated in the scope of its pattern variables. If *v* fails to match all patterns, exception `#(bad-match` *v src*`)` is raised, where *src* is the source location of the `match` clause if available.

See Table 1.1 for the pattern grammar.

(match-define <pattern> *exp*) **syntax**

expands to: see below

The `match-define` macro evaluates *exp* and matches the resulting input against the pattern. Pattern-variable bindings are established via `define` and inhabit the same scope in which the `match-define` form appears. The `match-define` macro does not support guard expressions. If the pattern fails to match, exception `#(bad-match` *v src*`)` is raised, where *v* is the datum that failed to match the pattern at source location *src* if available.

See Table 1.1 for the pattern grammar.

(match-let* ([<pattern> [(guard *g*)] *exp*] **syntax**
             ...)
  b1 b2 ...)

returns: the value of the last expression *b1 b2* ...

The `match-let*` macro evaluates each *exp* in the order specified and matches its value against its pattern and guard. The pattern variables of each clause extend the scope of its guard expression *g* and all subsequent pattern clauses and body expressions *b1 b2* .... The `match-let*` macro returns the value of the last body expression. If any pattern fails to match or any *g* returns `#f`, exception `#(bad-match` *v src*`)` is raised, where *v* is the datum that failed to match the pattern or guard at source location *src* if available.

See Table 1.1 for the pattern grammar.

(define-match-extension *ext handle-object* [*handle-field*]) **syntax**

expands to: see below

The `define-match-extension` macro attaches a property to the identifier *ext*, via `define-property`, so that the expander calls *handle-object* to translate `` `( ``*ext spec* `...)` patterns when generating code for `match`, `match-define`, `match-let*`, or `receive`. The *handle-object* procedure takes two arguments: *v*, an identifier that will be bound in the generated code to the value to be matched, and *pattern*, a syntax object for an expression of the form `` `( ``*ext spec* `...)`. The *handle-object* procedure can return `#f` to report an invalid *pattern*. Otherwise, *handle-object* should translate the given *pattern* to a list of one or more instructions in the following simple language:

`(bind` *v* *e*`)`	binds *v* to the value of *e* via `let` or `define`
`(guard` *g*`)`	rejects the match if *g* evaluates to `#f`
`(sub-match` *e* *pattern*`)`	matches the value of *e* against *pattern*
`(handle-fields` *input* *field-spec* ...`)`	invokes *handle-field* to translate each *field-spec*

The generated code evaluates the instructions in the order they are returned. For example, a `guard` expression may refer to a binding established by a `bind` earlier in the list of instructions. The `sub-match` and `handle-fields` instructions are processed at expand time and may appear only as the final instruction in the list returned by *handle-object*.

The `(handle-fields` *input* *field-spec* ...`)` instruction parses each *field-spec* from left to right and calls *handle-field* with five arguments: the *input* from the instruction, the *field* identified, the *var* that should be bound to the value of *field*, a list of *options* appearing in the *field-spec*, and the original pattern *context*. The following table shows how each *field-spec* is parsed into arguments for *handle-field*:

field-spec	*field*	*var*	*options*	notes
`,`*field*	*field*	*field*	`()`	*field* must be an identifier
`,@`*field*	*field*	*unique*	`()`	*field* must be an identifier
`[`*field* *pattern* *option* ...`]`	*field*	*unique*	`(`*option* ...`)`	*unique* is matched against *pattern*

The *handle-field* procedure can return `#f` to report an invalid *field*. Otherwise, *handle-field* should return a list of `bind` or `guard` instructions that bind *var* and perform any checks needed to confirm a match. The resulting instructions are evaluated in the order they are returned.

Where temporaries are introduced in the generated output, the *handle-object* and *handle-field* procedures should use `with-temporaries` to avoid unintended variable capture.

1.2.3 Exceptions

`(catch` *e1* *e2* ...`)` **syntax**

expands to: `($trap (lambda ()` *e1* *e2* ...`) ->EXIT)`

The `catch` macro evaluates expressions *e1* *e2* ... in a dynamic context that traps exceptions. If no exception is raised, the return value is the value of the last expression. If exception *reason* is raised, `#(EXIT` *reason*`)` is returned.

`(try` *e1* *e2* ...`)` **syntax**

expands to: `($trap (lambda ()` *e1* *e2* ...`) ->fault-condition)`

The `try` macro evaluates expressions *e1* *e2* ... in a dynamic context that traps exceptions. If no exception is raised, the return value is the value of the last expression. If exception *reason* is raised, the return value is a fault condition matching the extended match pattern `` `(catch `` *reason* [*e*]`)`.

`` `(catch `` *r* [*e*]`)` **match-extension**

matches: exceptions trapped by `try` or `catch`

The extended match pattern `` `(catch `` *r* [*e*]`)` matches exceptions trapped by `try`. For compatibility with older code, this pattern also matches exceptions trapped by `catch`. The *r* pattern is matched against the exit reason in the trapped exception. The optional *e* pattern is typically a `,`*variable* pattern that binds *variable* for use as an argument to `throw` or `raise`. If the trapped exception is a fault condition generated by `throw`, `make-fault`, or `make-fault/no-cc`, then *e* is matched against the fault condition, which may contain additional debugging context. Otherwise, *e* is matched against the exit reason.

`(throw` *r* [*inner*]`)` **procedure**
returns: does not return

The `throw` procedure raises a fault condition containing reason *r*, an optional inner exception *inner*, and the current continuation, which may provide useful debugging context. The exception raised may be trapped by `try` and matched using the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(make-fault` *r* [*inner*]`)` **procedure**
returns: a fault condition

The `make-fault` procedure returns a fault condition containing reason *r*, an optional inner exception *inner*, and the current continuation, which may provide useful debugging context. The return value matches the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(make-fault/no-cc` *r* [*inner*]`)` **procedure**
returns: a fault condition

The `make-fault/no-cc` procedure returns a fault condition containing reason *r*, and an optional inner exception *inner*, but omits the current continuation. The return value matches the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(arg-check` *who* `[`*arg* *pred* ...`]` ...`)` **syntax**
expands to:

```
(let ([who who])
  (let ([arg arg])
    (unless (and (pred arg) ...)
      (profile-me-as arg-check)
      (bad-arg who arg)))
  ...
  (void))
```

The `arg-check` macro raises a `bad-arg` exception if any *arg* fails any *pred* specified for that *arg*. Within coverage reports, profile counts on the `arg-check` keyword indicate the number of `bad-arg` cases encountered.

`(bad-arg` *who* *arg*`)` **procedure**
returns: never

The `bad-arg` procedure raises exception `#(bad-arg` *who* *arg*`)`.

`(dump-stack` [*op*]`)` **procedure**
`(dump-stack` *k* *op* *max-depth*`)`
returns: unspecified

The `dump-stack` procedure calls `walk-stack` to print information about the stack to textual output port *op*, which defaults to the current output port.

k is a continuation, and *max-depth* is either the symbol `default` or a positive fixnum. See `walk-stack` for details on the *max-depth* argument.

`(dump-stack` *op*`)` calls `(call/cc (lambda (`*k*`) (dump-stack` *k* *op* `'default)))`.

`(limit-stack` *e0 e1 ...*`)` **syntax**
expands to: `($limit-stack (lambda ()` *e0 e1 ...*`)` *source*`)`

The `limit-stack` macro adds a stack frame that may be recognized by `limit-stack?`. By default, `walk-stack` avoids descending below such frames. The `limit-stack` macro evaluates expressions *e0 e1 ...* from left to right and returns the values of the last expression.

`(limit-stack?` *x*`)` **procedure**
returns: see below

The `limit-stack?` procedure returns true if *x* is a continuation whose top frame is a `limit-stack` frame. Otherwise it returns `#f`.

`(walk-stack` *k base handle-frame combine* [*who max-depth truncated*]`)` **procedure**
returns: see below

The `walk-stack` procedure walks the stack of continuation *k* by calling the *handle-frame* and *combine* procedures for each stack frame until it reaches the base of the stack or a `limit-stack` frame, or depth reaches the optional *max-depth*, or the *next* argument to *combine* is not called.

The *handle-frame* procedure is called with four arguments:

description	a string describing the stack frame, e.g., `"#<continuation in g>"`
source	a source object identifying the return point or `#f`
proc-source	a source object identifying the procedure containing the return point or `#f`
vars	a list associating live free variables by name (or index) with their values

If *max-depth* is omitted or is the symbol `default`, then `walk-stack` uses the value of `walk-stack-max-depth` as *max-depth* and stops if recognizes a `limit-stack` frame. If *max-depth* is specified explicitly, then *walk-stack* does not stop at `limit-stack` frames. If `walk-stack` reaches a depth of *max-depth*, it calls the optional *truncated* procedure with *base* and *depth*. Otherwise, `walk-stack` calls the *combine* procedure with four arguments:

frame	the value returned by *handle-frame* for the current frame
base	the accumulator
depth	the zero-based depth of the current frame
next	a procedure that takes *base* and continues with the next frame

If `walk-stack` receives an invalid argument *val*, it calls `(bad-arg` *who val*`)` with the symbol `walk-stack` as the default value for the optional *who* argument. The default *truncated* procedure simply returns the value of *base* passed in.

`walk-stack-max-depth` **parameter**
returns: a nonnegative fixnum

The `walk-stack-max-depth` parameter specifies the default maximum depth to which `walk-stack` descends when the optional *max-depth* argument is omitted or is the symbol `default`.

`(exit-reason->stacks` *x*`)` **procedure**

returns: a list of continuations

The `exit-reason->stacks` procedure takes a Swish condition *x*, as created by `throw` or trapped by `try`, and returns a list of continuations recorded in *x*. The continuations are listed innermost to outermost.

`(make-process-parameter` *initial* [*filter*]`)` **procedure**

returns: a parameter procedure

The `make-process-parameter` procedure creates a parameter procedure *p* that provides mutable storage. Calling *p* with no arguments returns the current value of the parameter, and calling *p* with one argument sets the value of the parameter. The *filter*, if present, is a procedure of one argument that is applied to the *initial* and all subsequent values. If *filter* is not a procedure, exception `#(bad-arg make-process-parameter` *filter*`)` is raised.

`(reset-process-parameters!)` **procedure**

returns: unspecified

The `reset-process-parameters!` procedure resets all process parameters to their initial values.

`(on-exit` *finally b1 b2* ...`)` **syntax**

expands to:

```
(dynamic-wind
  void
  (lambda () b1 b2 ...)
  (lambda () finally))
```

The `on-exit` macro executes the body expressions *b1 b2* ... in a dynamic context that executes the *finally* expression whenever control leaves the body.

`(profile-me)` **procedure**

returns: unspecified

The `profile-me` procedure does nothing but provide a place-holder for the system profiler to count the call site. When profiling is turned off, `(profile-me)` expands to `(void)`, and the system optimizer eliminates it.

`(profile-me-as` *form*`)` **syntax**

returns: unspecified

The `profile-me-as` macro does nothing but provide a place-holder for the system profiler to count the call site. If source information is present on *form*, the profile count for this call site is attributed to that *form*. When profiling is turned off or when source information is not present on *form*, `profile-me-as` expands to `(void)`, and the system optimizer eliminates it.

`windows?` **syntax**

expands to: a boolean

The `windows?` macro expands to `#t` if the host is running Microsoft Windows and `#f` if not.

1.2.4 I/O

`(binary->utf8` *bp*`)` **procedure**

returns: a transcoded textual port wrapping *bp*

The `binary->utf8` procedure takes a binary port *bp* and returns a textual port wrapping *bp* using `transcoded-port` and `(make-utf8-transcoder)`. The original port *bp* is marked closed so that it cannot be used except through the associated textual port.

`(make-directory-path` *path* [*mode*]`)` **procedure**

returns: *path*

The `make-directory-path` procedure creates directories as needed for the file *path* using *mode*, which defaults to `#o777`. It returns *path*.

`(make-utf8-transcoder)` **procedure**

returns: a UTF-8 transcoder

The `make-utf8-transcoder` procedure creates a UTF-8 transcoder with end-of-line style `none` and error-handling mode `replace`.

`(path-absolute` *path* [*base*]`)` **procedure**

returns: a string

The `path-absolute` procedure returns the normalized absolute path of *path* relative to the *base* directory, which defaults to the current directory.

`(path-combine` $path_1$ $path_2$ ...`)` **procedure**

returns: a string combining the paths

The `path-combine` procedure appends one or more paths, inserting the directory-separator character between each pair of paths as needed.

`(path-normalize` *path*`)` **procedure**

returns: a string with the normalized path

The `path-normalize` procedure removes unnecessary directory separators and simplifies "." and ".." components from *path*.

1.2.5 Error Strings

`current-exit-reason->english` **parameter**

value: a procedure of one argument that returns an English string

The `current-exit-reason->english` parameter specifies the conversion procedure used by `exit-reason->english`. It defaults to `swish-exit-reason->english`.

`(exit-reason->english` *x*`)`	**procedure**

returns: a string in U.S. English

The `exit-reason->english` procedure converts an exit reason into an English string using the procedure stored in parameter `current-exit-reason->english`.

`(swish-exit-reason->english` *x*`)`	**procedure**

returns: a string in U.S. English

The `swish-exit-reason->english` procedure converts an exit reason from Swish into an English string.

1.2.6 String Utilities

The string utilities below are found in the `(swish string-utils)` library.

`(ct:join` *sep s ...*`)`	**syntax**

expands to: a string or a call to `string-append`

The `ct:join` macro uses `ct:string-append` to join adjacent string literals into a literal string or a call to `string-append` where adjacent string literals are combined. The *sep*, which must be a literal string or character, is inserted between adjacent elements of *s ...*.

`(ct:string-append` *s ...*`)`	**syntax**

expands to: a string or a call to `string-append`

The `ct:string-append` macro appends adjacent string literals at compile time and expands into the resulting literal string or a call to `string-append` where adjacent string literals are combined.

`(ends-with?` *s p*`)`	**procedure**

returns: a boolean

The `ends-with?` procedure determines whether or not the string *s* ends with string *p* using case-sensitive comparisons.

`(ends-with-ci?` *s p*`)`	**procedure**

returns: a boolean

The `ends-with-ci?` procedure determines whether or not the string *s* ends with string *p* using case-insensitive comparisons.

`(format-rfc2822` *d*`)`	**procedure**

returns: a string like "Thu, 28 Jul 2016 17:20:11 -0400"

The `format-rfc2822` procedure returns a string representation of the date object *d* in the form specified in Section 3.3 of RFC 2822 [6].

(`join` *ls separator* [*last-separator*]) **procedure**
returns: a string

The `join` procedure returns the string formed by displaying each of the elements of list *ls* separated by displaying *separator*. When *last-separator* is specified, it is used as the last separator.

(`oxford-comma` [*prefix*] *elt-fmt conj* [*suffix*]) **procedure**
returns: a string

The `oxford-comma` procedure constructs a format string for use with `errorf`, `format`, `printf`, etc., to join the elements of a list with commas and/or *conj*, as appropriate. The *elt-fmt* argument is the format string for individual items of the list. The *conj* argument is a string used to separate the final two elements of the list. The *prefix* and *suffix* arguments must be supplied together or omitted. If omitted, *prefix* defaults to `"~{"` and *suffix* defaults to `"~}"`.

(`split` *str separator*) **procedure**
returns: a list of strings

The `split` procedure divides the *str* string by the *separator* character into a list of strings, none of which contain *separator*.

(`split-n` *str separator n*) **procedure**
returns: a list of no more than *n* strings

The `split-n` procedure divides the *str* string by the *separator* character into a list of at most *n* strings. The last string may contain *separator*.

(`starts-with?` *s p*) **procedure**
returns: a boolean

The `starts-with?` procedure determines whether or not the string *s* starts with string *p* using case-sensitive comparisons.

(`starts-with-ci?` *s p*) **procedure**
returns: a boolean

The `starts-with-ci?` procedure determines whether or not the string *s* starts with string *p* using case-insensitive comparisons.

(`trim-whitespace` *s*) **procedure**
returns: a string

The `trim-whitespace` procedure returns a string in which any leading or trailing whitespace in *s* has been removed. Internal whitespace is not affected.

(`wrap-text` *op width initial-indent subsequent-indent text*) **procedure**
returns: unspecified

The `wrap-text` procedure writes the given *text* to the textual output port *op* after collapsing spaces that separate words. The first line is indented by *initial-indent* spaces. Subsequent lines are indented by *subsequent-indent* spaces. If possible, `wrap-text` breaks lines that would exceed *width*. Newlines and tabs are preserved, but tabs are treated as if they were the width of a single character.

The *text* argument may be a string or a list of strings. If *text* is a list, it is treated as if it were the string obtained via `(join` *text* `#\space)`.

`(symbol-append .` *ls*`)` **procedure**
returns: a symbol

The `symbol-append` procedure returns the symbol formed by appending the symbols passed as arguments.

1.2.7 Macro Utilities

`(pretty-syntax-violation` *msg form* [*subform* [*who*]]`)` **procedure**
returns: never

The `pretty-syntax-violation` procedure raises a syntax violation. It differs from the native `syntax-violation` in that it formats *form* and *subform* using `pretty-format` abbreviations, and it does not attempt to infer a who condition when *who* is not provided, as this can produce confusing results in error messages involving match patterns. To provide more readable exception messages, it constructs the formatted message condition by calling `pretty-print` before raising the exception, and it prevents `display-condition` from formatting the `&syntax` condition within the compound condition it constructs.

`(with-temporaries (`*id* `...) e0 e1 ...)` **syntax**
expands to:

```
(with-syntax ([(id ...) (generate-temporaries '(id ...))])
  e0 e1 ...)
```

The `with-temporaries` macro binds each macro-language pattern variable *id* to a fresh generated identifier within the body `(begin` *e0 e1* `...)`.

`(define-syntactic-monad` *m id* `...)` **syntax**
expands to: see below

The `define-syntactic-monad` macro defines a macro *m* for defining and calling procedures that take implicit *id* ... arguments in addition to any explicit arguments that may be provided. Such macros make it easier to write state machines in a functional style by allowing the programmer to specify only the values that change at a call.

A call to *m* takes the form `(`*m* e_0 `([`id_i e_i`] ...)` *x* `...)` or `(`*m kwd form* `...)` where *kwd* is `case-lambda`, `define`, `lambda`, `let`, `trace-case-lambda`, `trace-define`, `trace-lambda`, or `trace-let`. The first form constructs a call to e_0 and is described below along with the `let` case. The other cases may be understood in terms of the following template expansions or their natural extension to tracing variants:

`(`*m* `lambda` *fmls body* `...)` → `(lambda (`*id* `... .` *fmls*`)` *body* `...)`

(*m* `define` (*proc* . *fmls*) *body* ...) → (`define` *proc* (*m* `lambda` *fmls* *body* ...))

(*m* `case-lambda` [*fmls* *body* ...] ...) → (`case-lambda` [(*id* ... . *fmls*) *body* ...] ...)

The call form (*m* e_0 ([id_i e_i] ...) *x* ...) constructs a call to e_0 where the arguments are the *id* ... with any id_i replaced by e_i all followed by the *x* ... expressions. Any *id* that does not have an explicit [id_i e_i] binding in the call form must have a binding in scope. For calls within the body of an (*m* `case-lambda` ...), (*m* `define` ...), (*m* `lambda` ...), or (*m* `let` ...) form such bindings are already in scope. As a convenience, the call syntax (*m* *f*) is equivalent to (*m* *f* ()) which specifies an empty list of implicit-binding updates.

The (*m* `let` ...) form constructs a named `let` using syntax inspired by the call form described above. That is, (*m* `let` *name* ([id_i e_i] ...) ([x_j e_j] ...) *body* ...) constructs a named `let` *name* that binds *id* ... *x* ... with the initial value of each *id* coming from the value of the corresponding e_i or else the binding already in scope and the initial value of each x_j supplied by the corresponding e_j. Within *body* ..., a call of the form (*m* *name* (id_i ...) e_j ...) can supply required values for each of the x_j along with new values for *id* ... if needed.

Chapter 2

HTML Interface

2.1 Introduction

The programming interface includes procedures for the HyperText Markup Language (HTML) version 5 [5].

2.2 Programming Interface

`(html:encode` *s*`)` **procedure**
`(html:encode` *op s*`)`

returns: see below

The `html:encode` procedure converts special character entities in string *s*.

input	output
"	"
&	&
<	<
>	>

The single argument form of `html:encode` returns an encoded string.

The two argument form of `html:encode` sends the encoded string to the textual output port *op*.

`(html->string` *x*`)` **procedure**
`(html->string` *op x*`)`

returns: see below

The `html->string` procedure transforms an object into HTML. The transformation, H, is described below:

x	*H*(*x*)
()	nothing
#!void	nothing
string	*E*(*string*)
number	*number*
(begin *pattern* ...)	*H*(*pattern*)...
(cdata *string* ...)	[!CDATA[*string*...]]
(html5 [(@ *attr* ...)] *pattern* ...)	<!DOCTYPE html><html *A*(*attr*) ...>*H*(*pattern*)...</html>
(raw *string* ...)	*string*...
(script [(@ *attr* ...)] *string* ...)	<script *A*(*attr*) ...>*string*...</script>
(style [(@ *attr* ...)] *string* ...)	<style *A*(*attr*) ...>*string*...</style>
(*tag* [(@ *attr* ...)] *pattern* ...)	<*tag* *A*(*attr*) ...>*H*(*pattern*)...</*tag*>
(*void-tag* [(@ *attr* ...)])	<*void-tag* *A*(*attr*) ...>

E denotes the html:encode function.

For the html5 tag, if there is no *attr* with lang as its key, then *H* acts as if the *attr* (lang "en") were specified.

A *void-tag* is one of area, base, br, col, embed, hr, img, input, keygen, link, menuitem, meta, param, source, track, or wbr. A *tag* is any other symbol.

The attribute transformation, *A*, is described below, where *key* is a symbol:

attr	*A*(*attr*)
#!void	nothing
(*key*)	*key*
(*key* *string*)	*key*="*E*(*string*)"
(*key* *number*)	*key*="*number*"

The single argument form of html->string returns an encoded HTML string.

The two argument form of html->string sends the encoded HTML string to the textual output port *op*.

Input that does not match the specification causes a #(bad-arg html->string *x*) exception to be raised.

(html->bytevector *x*) **procedure**

returns: a bytevector

The html->bytevector procedure calls html->string on *x* using a bytevector output port transcoded using (make-utf8-transcoder) and returns the resulting bytevector.

Chapter 3

JSON Interface

3.1 Introduction

The programming interface includes procedures for JavaScript Object Notation (JSON) [3]. In order to support all floating-point values, it extends the specification with Infinity, -Infinity, and NaN.

This implementation translates JavaScript types into the following Scheme types:

JavaScript	Scheme
`true`	`#t`
`false`	`#f`
`null`	symbol `null`
`Infinity`	`+inf.0`
`-Infinity`	`-inf.0`
`NaN`	`+nan.0`
string	*string*
number	*number*
array	*list*
object	hashtable mapping symbols to values

This implementation does not range check values to ensure that a JavaScript implementation can interpret the data.

3.2 Programming Interface

`(json:extend-object` *ht* `[`*key* *value*`] ...)` **syntax**

The `json:extend-object` construct adds the *key* / *value* pairs to the hashtable *ht* using `hashtable-set!`. Each *key* is a literal identifier or an unquoted expression `,`*e* that evaluates to a symbol. The resulting expression returns *ht*.

`(json:make-object [key value] ...)` **syntax**

The `json:make-object` construct expands into a call to `json:extend-object` with a new hashtable.

`(json:object? x)` **procedure**
returns: a boolean

The `json:object?` procedure determines whether or not the datum *x* is an object created by `json:make-object`.

`(json:cells ht)` **procedure**
returns: a vector

The `json:cells` procedure returns a vector containing the cells of the underlying hashtable.

`(json:size ht)` **procedure**
returns: an integer

The `json:size` procedure returns the number of cells in the underlying hashtable.

`(json:delete! ht path)` **procedure**
returns: unspecified

The `json:delete!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:delete!` is equivalent to `hashtable-delete!`. Otherwise, `json:delete!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:delete!` reaches the final key in *path*, it calls `hashtable-delete!` to remove the association for that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:delete!` has no effect.

`(json:ref ht path default)` **procedure**
returns: the value found by traversing *path* in *ht*, *default* if none

The `json:ref` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:ref` is equivalent to `hashtable-ref`. Otherwise, `json:ref` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:ref` reaches the final key in *path*, it calls `hashtable-ref` to retrieve the value of that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, or if the final hashtable does not contain the final key, `json:ref` returns *default*.

`(json:set! ht path value)` **procedure**
returns: unspecified

The `json:set!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:set!` is equivalent to `hashtable-set!`. Otherwise, `json:set!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:set!` reaches the final key in *path*, it calls `hashtable-set!` to set that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:set!` installs an empty hashtable at that key before proceding. If *path* is malformed at some point, `json:set!` may still mutate hashtables along the valid portion of the path before reporting an error.

(`json:update!` *ht path procedure default*) **procedure**
returns: unspecified

The `json:update!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:update!` is equivalent to `hashtable-update!`. Otherwise, `json:update!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:update!` reaches the final key in *path*, it calls `hashtable-update!` to update that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:update!` installs an empty hashtable at that key before proceding. If *path* is malformed at some point, `json:update!` may still mutate hashtables along the valid portion of the path before reporting an error.

(`json:read` *ip* [*custom-inflate*]) **procedure**
returns: a Scheme object or the eof object

The `json:read` procedure reads characters from the textual input port *ip* and returns an appropriate Scheme object. When `json:read` encounters a JSON object, it builds the corresponding hashtable and calls *custom-inflate* to perform application-specific conversion. By default, *custom-inflate* is the identity function.

The following exceptions may be raised:

- `invalid-surrogate-pair`
- `unexpected-eof`
- `#(unexpected-input` *data input-position*`)`

(`json:write` *op x* [*indent*] [*custom-write*]) **procedure**
returns: unspecified

The `json:write` procedure writes the object *x* to the textual output port *op* in JSON format. JSON objects are sorted by key using `string<?` on the string values of the symbols to provide stable output. Scheme fixnums, bignums, and finite flonums may be used as numbers.

When *indent* is a non-negative fixnum, the output is more readable by a human. List items and key/value pairs are indented on individual lines by the specified number of spaces. When *indent* is 0, a newline is added to the end of the output. The default indent of `#f` produces compact output.

The optional *custom-write* procedure may intervene to handle lists and hashtables differently or to handle objects that have no direct JSON counterpart. If *custom-write* does not handle a given object, it should return false to let `json:write` proceed normally. The *custom-write* procedure is called with four arguments: the textual output port *op*, the Scheme object *x*, the current *indent* level, and a writer procedure *wr* that should be used to write the values of arbitrary Scheme objects. The *wr* procedure is equivalent to `(lambda (`*op x indent*`) (json:write` *op x indent custom-write*`))`.

If an object cannot be formatted, `#(invalid-datum` *x*`)` is raised.

`(json:write-object` *op indent wr* `[`*key value*`] ...)` **syntax**
returns: `#t`

Given a textual output port *op*, an *indent* level, and a writer procedure *wr*, the `json:write-object` construct writes a JSON object with the given *key* / *value* pairs to *op*, sorted by key using `string<?` on the string values of the symbols. Each *key* must be a distinct symbol. The *wr* procedure takes *op*, an object *x*, and an *indent* level just like the *wr* procedure that is passed to `json:write`'s *custom-write* procedure.

The following are equivalent, provided the keys are symbols.

```
(begin (json:write op (json:make-object [key value] ...) indent) #t)
(json:write-object op indent json:write [key value] ...)
```

The latter trades code size and compile time for run-time efficiency. At compile time, `json:write-object` sorts the keys and preformats the strings that will separate values.

`(json:object->bytevector` *x* [*indent*] [*custom-write*]`)` **procedure**
returns: a bytevector

The `json:object->bytevector` procedure calls `json:write` on *x* with the optional *indent* and *custom-write*, if any, using a bytevector output port transcoded using `(make-utf8-transcoder)` and returns the resulting bytevector.

`(json:bytevector->object` *x* [*custom-inflate*]`)` **procedure**
returns: a Scheme object

The `json:bytevector->object` procedure creates a bytevector input port on *x*, calls `json:read` with the optional *custom-inflate*, if any, and returns the resulting Scheme object after making sure the rest of the bytevector is only whitespace.

`(json:object->string` *x* [*indent*] [*custom-write*]`)` **procedure**
returns: a JSON formatted string

The `json:object->string` procedure creates a string output port, calls `json:write` on *x* with the optional *indent* and *custom-write*, if any, and returns the resulting string.

`(json:string->object` *x* [*custom-inflate*]`)` **procedure**
returns: a Scheme object

The `json:string->object` procedure creates a string input port on *x*, calls `json:read` with the optional *custom-inflate*, if any, and returns the resulting Scheme object after making sure the rest of the string is only whitespace.

`(json:write-structural-char` *x indent op*`)` **procedure**
returns: the new indent level

The `json:write-structural-char` procedure writes the character *x* at an appropriate *indent* level to the textual output port *op*. The character should be one of the following JSON structural characters: `[ ] { } : ,`

This procedure is intended for use within custom writers passed in to `json:write` and, for performance, it does not check its input arguments.

`(stack->json` *k* [*max-depth*]`)` **procedure**

returns: a JSON object

The `stack->json` procedure renders the stack of continuation *k* as a JSON object by calling `walk-stack`. The return value may contain the following keys:

`type`	`"stack"`
`depth`	the depth of the stack
`truncated`	if present, the *max-depth* at which the stack dump was truncated
`frames`	if present, a list of JSON objects representing stack frames

A stack frame may contain the following keys:

`type`	`"stack-frame"`
`depth`	the depth of this frame
`source`	if present, a source object for the return point
`procedure-source`	if present, a source object for the procedure containing the return point
`free`	if present, a list of JSON objects representing free variables

A source object *x* with source file descriptor *sfd* is represented by a JSON object containing the following keys:

`bfp`	`(source-object-bfp` *x*`)`
`efp`	`(source-object-efp` *x*`)`
`path`	`(source-file-descriptor-path` *sfd*`)`
`checksum`	`(source-file-descriptor-checksum` *sfd*`)`

A free variable with value *val* is represented by a JSON object containing the following keys:

`name`	a string containing the variable name or its index
`value`	the result of `(format "~s"` *val*`)`

`(json-stack->string` [*op*] *x*`)` **procedure**

returns: see below

The two argument form of `json-stack->string` prints the stack represented by JSON object *x* to the textual output port *op*. The single argument form of `json-stack->string` prints the stack represented by JSON object *x* to a string output port and returns the resulting string. In either case, the printed form resembles that generated by `dump-stack` except that source locations are given as file offsets rather than line and character numbers.

Chapter 4

Regular Expressions

4.1 Introduction

The regular expressions library `(swish pregexp)` is a derivative of pregexp: Portable Regular Expressions for Scheme and Common Lisp [7]. It provides regular expressions modeled on Perl's [4, 8] and includes such powerful directives as numeric and non-greedy quantifiers, capturing and non-capturing clustering, POSIX character classes, selective case- and space-insensitivity, back-references, alternation, backtrack pruning, positive and negative look-ahead and look-behind, in addition to the more basic directives familiar to all regexp users.

A *regexp* is a string that describes a pattern. A regexp matcher tries to *match* this pattern against (a portion of) another string, which we will call the *text string*. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `"abc"` matches a string that contains the characters `a`, `b`, `c` in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `"a.c"`, the characters `a` and `c` do stand for themselves but the *metacharacter* '`.`' can match *any* character (other than newline). Therefore, the pattern `"a.c"` matches an `a`, followed by *any* character, followed by a `c`.

If we needed to match the character '`.`' itself, we *escape* it, i.e., precede it with a backslash (\). The character sequence `\.` is thus a *metasequence*, since it doesn't match itself but rather just '`.`'. So, to match `a` followed by a literal '`.`' followed by `c`, we use the regexp pattern `"a\\.c"`.[1] Another example of a metasequence is `\t`, which is a readable way to represent the tab character.

We will call the string representation of a regexp the *U-regexp*, where *U* can be taken to mean *Unix-style* or *universal*, because this notation for regexps is universally familiar. Our implementation uses an intermediate tree-like representation called the *S-regexp*, where *S* can stand for *Scheme*, *symbolic*, or *s-expression*. S-regexps are more verbose and less readable than U-regexps, but they are much easier for Scheme's recursive procedures to navigate.

[1] The double backslash is an artifact of Scheme strings, not the regexp pattern itself. When we want a literal backslash inside a Scheme string, we must escape it so that it shows up in the string at all. Scheme strings use backslash as the escape character, so we end up with two backslashes.

4.2 Programming Interface

`(pregexp` *regexp*`)` **procedure**

returns: an S-regexp

The `pregexp` procedure takes a U-regexp string *regexp* and returns an S-regexp.

`(re` *regexp*`)` **syntax**

expands to: `(pregexp` *regexp*`)`

If *regexp* is a literal string, the `re` macro expands to the result of evaluating `(pregexp` *regexp*`)` at expand time. Otherwise it expands into a run-time call to `pregexp`.

`(pregexp-match-positions` *pat* *str* [*start* [*end*]]`)` **procedure**

returns: `((`*s* `.` *e*`)` ...`)` or `#f`

The `pregexp-match-positions` procedure takes a regexp pattern *pat* and a text string *str* and returns a *match* if the regexp matches (some part of) the text string between the inclusive *start* index (defaults to 0) and the exclusive *end* index (defaults to the length of *str*).

The regexp may be either a U- or an S-regexp. `pregexp-match-positions` will internally compile a U-regexp to an S-regexp before proceeding with the matching. If you find yourself calling `pregexp-match-positions` repeatedly with the same U-regexp, it may be advisable to explicitly convert the latter into an S-regexp once beforehand, using `pregexp`, to save needless recompilation.

`pregexp-match-positions` returns a list of *index pairs* if the regexp matches the string and `#f` if it does not match. Index pair `(`*s* `.` *e*`)` gives the inclusive starting index *s* and exclusive ending index *e* of the matching substring with respect to *str*. The first index pair indicates the entire match, and subsequent pairs indicate submatches. Some of the submatches may be `#f`.

`(pregexp-match` *pat* *str* [*start* [*end*]]`)` **procedure**

returns: list of matching substrings or `#f`

The `pregexp-match` procedure is called like `pregexp-match-positions`, but instead of returning index pairs, it returns the matching substrings. The first substring is the entire match, and subsequent substrings are submatches, some of which may be `#f`.

`(pregexp-split` *pat* *str*`)` **procedure**

returns: list of substrings from *str*

The `pregexp-split` procedure takes two arguments, a regexp pattern *pat* and a text string *str*, and returns a list of substrings of the text string, where the pattern identifies the delimiter separating the substrings. The returned substrings do not include the delimiter.

If the pattern can match an empty string, then the list of all the single-character substrings is returned.

To identify one or more spaces as the delimiter, take care to use the regexp `" +"`, not `" *"`.

`(pregexp-replace` *pat* *str* *ins*`)` **procedure**

returns: a string

The `pregexp-replace` procedure replaces the matched portion of the text string by another string. The first argument is the pattern *pat*, the second the text string *str*, and the third is the string to be inserted *ins*, which may contain back-references (see §4.3.4).

If the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to *str*.

`(pregexp-replace*` *pat str ins*`)` **procedure**
returns: a string

The `pregexp-replace*` procedure replaces *all* matches of regexp *pat* in the text string *str* by the insert string *ins*, which may contain back-references (see §4.3.4).

As with `pregexp-replace`, if the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to *str*.

`(pregexp-quote` *str*`)` **procedure**
returns: a U-regexp

The `pregexp-quote` procedure takes an arbitrary string *str* and returns a U-regexp string that precisely represents it. In particular, characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

`pregexp-quote` is useful when building a composite regexp from a mix of regexp strings and verbatim strings.

4.3 The Regexp Pattern Language

4.3.1 Basic Assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string respectively. They ensure that their adjoining regexps match at the beginning or end of the text string. Examples:

`(pregexp-match-positions "^contact" "first contact")` ⇒ `#f`

The regexp fails to match because `contact` does not occur at the beginning of the text string.

`(pregexp-match-positions "laugh$" "laugh laugh laugh laugh")` ⇒ `((18 . 23))`.

The regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a *word boundary* exists.

`(pregexp-match-positions "yack\\b" "yackety yack")` ⇒ `((8 . 12))`

The `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` has the opposite effect to `\b`. It asserts that a word boundary does not exist.

`(pregexp-match-positions "an\\B" "an analysis")` ⇒ `((3 . 5))`

The `an` that doesn't end in a word boundary is matched.

4.3.2 Characters and Character Classes

Typically a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. Thus, metasequences `\n`, `\r`, `\t`, and `\.` match the newline, return, tab, and period characters respectively.

The *metacharacter* period (`.`) matches *any* character other than newline.

`(pregexp-match "p.t" "pet")` ⇒ `("pet")`

It also matches `pat`, `pit`, `pot`, `put`, and `p8t` but not `peat` or `pffffft`.

A *character class* matches any one character from a set of characters. A typical format for this is the *bracketed character class* `[...]`, which matches any one character from the non-empty sequence of characters enclosed within the brackets.[2] Thus `"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put` and nothing else.

Inside the brackets, a hyphen (`-`) between two characters specifies the ASCII range between the characters. For example, `"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, *and* `tag`, *and* `tan`, `tao`, `tap`.

An initial caret (`^`) after the left bracket inverts the set specified by the rest of the contents, i.e., it specifies the set of characters *other than* those identified in the brackets. For example, `"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc.) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. `-` is a metacharacter only when it's inside brackets, and neither the first nor the last character.

Bracketed character classes cannot contain other bracketed character classes (although they contain certain other types of character classes—see below). Thus a left bracket (`[`) inside a bracketed character class doesn't have to be a metacharacter; it can stand for itself. For example, `"[a[b]"` matches `a`, `[`, and `b`.

Furthermore, since empty bracketed character classes are disallowed, a right bracket (`]`) immediately occurring after the opening left bracket also doesn't need to be a metacharacter. For example, `"[]ab]"` matches `]`, `a`, and `b`.

Some Frequently Used Character Classes

Some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions. `\d` matches a digit using `char-numeric?`; `\s` matches a whitespace character using `char-whitespace?`; and `\w` matches a character that could be part of a word.[3]

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes. Thus `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-word character.

Remember to include a double backslash when putting these metasequences in a Scheme string:

[2] Requiring a bracketed character class to be non-empty is not a limitation, since an empty character class can be more easily represented by an empty string.

[3] Following regexp custom, we identify word characters as alphabetic, numeric, or underscore (`_`).

`(pregexp-match "\\d\\d" "0 dear, 1 have 2 read catch 22 before 9")` ⇒ `("22")`

These character classes can be used inside a bracketed expression. For example, `"[a-z\\d]"` matches a lower-case letter or a digit.

POSIX Character Classes

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression. The POSIX classes supported are:

`[:alnum:]`	letters and digits
`[:alpha:]`	letters
`[:algor:]`	the letters `c`, `h`, `a` and `d`
`[:ascii:]`	7-bit ASCII characters
`[:blank:]`	widthful whitespace, i.e., space and tab
`[:cntrl:]`	control characters, viz, those with code < 32
`[:digit:]`	digits, same as `\d`
`[:graph:]`	characters that use ink
`[:lower:]`	lower-case letters
`[:print:]`	ink-users plus widthful whitespace
`[:space:]`	whitespace, same as `\s`
`[:upper:]`	upper-case letters
`[:word:]`	letters, digits, and underscore, same as `\w`
`[:xdigit:]`	hex digits

For example, the regexp `"[[:alpha:]_]"` matches a letter or underscore.

`(pregexp-match "[[:alpha:]_]" "-x-")` ⇒ `("x")`

`(pregexp-match "[[:alpha:]_]" "-_-")` ⇒ `("_")`

`(pregexp-match "[[:alpha:]_]" "-:-")` ⇒ `#f`

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will *not* be read as the letter class. Rather it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, and `h`.

`(pregexp-match "[:alpha:]" "-a-")` ⇒ `("a")`

`(pregexp-match "[:alpha:]" "-_-")` ⇒ `#f`

By placing a caret (`^`) immediately after `[:`, you get the inversion of that POSIX character class. Thus, `[:^alpha:]` is the class containing all characters except the letters.

4.3.3 Quantifiers

The *quantifiers* `*`, `+`, and `?` match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

`(pregexp-match-positions "c[ad]*r" "cadaddadddr")` ⇒ `((0 . 11))`

```
(pregexp-match-positions "c[ad]*r" "cr") ⇒ ((0 . 2))
(pregexp-match-positions "c[ad]+r" "cadaddadddr") ⇒ ((0 . 11))
(pregexp-match-positions "c[ad]+r" "cr") ⇒ #f
(pregexp-match-positions "c[ad]?r" "cadaddadddr") ⇒ #f
(pregexp-match-positions "c[ad]?r" "cr") ⇒ ((0 . 2))
(pregexp-match-positions "c[ad]?r" "car") ⇒ ((0 . 3))
```

Numeric Quantifiers

You can use braces to specify much finer-tuned quantification than is possible with *, +, and ?.

The quantifier {m} matches exactly m instances of the preceding subpattern. m must be a nonnegative integer.

The quantifier {m,n} matches at least m and at most n instances. m and n are nonnegative integers with $m \leq n$. You may omit either or both numbers, in which case m defaults to 0 and n to infinity.

It is evident that + and ? are abbreviations for {1,} and {0,1} respectively. * abbreviates {,}, which is the same as {0,}.

```
(pregexp-match "[aeiou]{3}" "vacuous") ⇒ ("uou")
(pregexp-match "[aeiou]{3}" "evolve") ⇒ #f
(pregexp-match "[aeiou]{2,3}" "evolve") ⇒ #f
(pregexp-match "[aeiou]{2,3}" "zeugma") ⇒ ("eu")
```

Non-greedy Quantifiers

The quantifiers described above are *greedy*, i.e., they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
(pregexp-match "<.*>" "<tag1> <tag2> <tag3>") ⇒ ("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a ? to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
(pregexp-match "<.*?>" "<tag1> <tag2> <tag3>") ⇒ ("<tag1>")
```

The non-greedy quantifiers are respectively: *?, +?, ??, {m}?, and {m,n}?. Note the two uses of the metacharacter ?.

4.3.4 Clusters

Clustering, i.e., enclosure within parentheses (...), identifies the enclosed subpattern as a single entity. It causes the matcher to *capture* the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match.

```
(pregexp-match "([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
⇒ ("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity.

```
(pregexp-match "(poo )*" "poo poo platter") ⇒ ("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
(pregexp-match "([a-z ]+;)*" "lather; rinse; repeat;")
⇒ ("lather; rinse; repeat;" " repeat;")
```

Here the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`.

```
(define date-re
  ;; match 'month year' or 'month day, year'.
  ;; subpattern matches day, if present
  (pregexp "([a-z]+) +([0-9]+,)? *([0-9]+)"))
```

```
(pregexp-match date-re "jan 1, 1970") ⇒ ("jan 1, 1970" "jan" "1," "1970")

(pregexp-match date-re "jan 1970") ⇒ ("jan 1970" "jan" #f "1970")
```

Back-references

Submatches can be used in the insert string argument of the procedures `pregexp-replace` and `pregexp-replace*`. The insert string can use \\n as a *back-reference* to refer back to the n^{th} submatch, i.e., the substring that matched the n^{th} subpattern. `\0` refers to the entire match, and it can also be specified as `\&`.

```
(pregexp-replace "_(.+?)_" "the _nina_, the _pinta_, and the _santa maria_" "*\\1*")
⇒ "the *nina*, the _pinta_, and the _santa maria_"
```

```
(pregexp-replace* "_(.+?)_" "the _nina_, the _pinta_, and the _santa maria_" "*\\1*")
⇒ "the *nina*, the *pinta*, and the *santa maria*"
```

```
(pregexp-replace "(\\S+) (\\S+) (\\S+)" "eat to live" "\\3 \\2 \\1")
⇒ "live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a back-reference \\n from an immediately following number.

Back-references can also be used within the regexp pattern to refer back to an already matched subpattern in the pattern. \\n stands for an exact repeat of the n^{th} submatch.[4]

[4] `\0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.

```
(pregexp-match "([a-z]+) and \\1" "billions and billions")
⇒ ("billions and billions" "billions")
```

Note that the back-reference is not simply a repeat of the previous subpattern. Rather it is a repeat of *the particular substring already matched by the subpattern*.

In the above example, the back-reference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to—`([a-z]+)`—would have had no problem doing so:

```
(pregexp-match "([a-z]+) and \\1" "billions and millions") ⇒ #f
```

The following corrects doubled words:

```
(pregexp-replace* "(\\S+) \\1" "now is the the time for all good men to to come to
the aid of of the party" "\\1")
⇒ "now is the time for all good men to come to the aid of the party"
```

The following marks all immediately repeating patterns in a number string:

```
(pregexp-replace* "(\\d+)\\1" "123340983242432420980980234" "\\1,\\1")
⇒ "123,34098324,243242098,0980234"
```

Non-capturing Clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. In such cases, use `(?:` instead of `(` as the cluster opener. In the following example, the non-capturing cluster eliminates the directory portion of a given pathname, and the capturing cluster identifies the basename.

```
(pregexp-match "^(?:[a-z]*/)*([a-z]+)$" "/usr/local/bin/scheme")
⇒ ("/usr/local/bin/scheme" "scheme")
```

Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*.[5] You can put *modifiers* there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match *case-insensitively*:

```
(pregexp-match "(?i:hearth)" "HeartH") ⇒ ("HeartH")
```

The modifier `x` causes the subpattern to match *space-insensitively*, i.e., spaces and comments within the subpattern are ignored. Comments are introduced as usual with a semicolon (`;`) and extend till the end of the line. If you need to include a literal space or semicolon in a space-insensitized subpattern, escape it with a backslash.

```
(pregexp-match "(?x: a lot)" "alot") ⇒ ("alot")

(pregexp-match "(?x: a \\ lot)" "a lot")
⇒ ("a lot")


(pregexp-match "(?x:
```

[5] A useful, if terminally cute, coinage from the abbots of Perl [8].

```
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "a man; a plan; a canal")
```

⇒ ("a man; a plan; a canal")

You can put more than one modifier in the cloister.

```
(pregexp-match "(?ix:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "A Man; a Plan; a Canal")
```

⇒ ("A Man; a Plan; a Canal")

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` and `-x` in a *subcluster* to overturn the insensitivities caused by an enclosing cluster.

`(pregexp-match "(?i:the (?-i:TeX)book)" "The TeXbook")` ⇒ `("The TeXbook")`

This regexp will allow any casing for `the` and `book` but insists that `TeX` not be differently cased.

4.3.5 Alternation

You can specify a list of *alternate* subpatterns by separating them by `|`. The `|` separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parentheses).

`(pregexp-match "f(ee|i|o|um)" "a small, final fee")` ⇒ `("fi" "i")`

```
(pregexp-replace* "([yi])s(e[sdr]?|ing|ation)"
 "it is energising to analyse an organisation pulsing with noisy organisms"
 "\\1z\\2")
```

⇒ "it is energizing to analyze an organization pulsing with noisy organisms"

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

`(pregexp-match "f(?:ee|i|o|um)" "fun for all")` ⇒ `("fo")`

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

`(pregexp-match "call|call/cc" "call/cc")` ⇒ `("call")`

To allow the longer alternate to have a shot at matching, place it before the shorter one:

`(pregexp-match "call/cc|call" "call/cc")` ⇒ `("call/cc")`

In any case, an overall match for the entire regexp is always preferred to an overall non-match. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
(pregexp-match "(?:call|call/cc) constrained" "call/cc constrained")
⇒ ("call/cc constrained")
```

4.3.6 Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
(pregexp-match "a*a" "aaaa")
```

The regexp consists of two subregexps, `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `"aaaa"`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of *three* `a`'s. If even this fails, as in the call

```
(pregexp-match "a*aa" "aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation, too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

Disabling Backtracking

Sometimes it is efficient to disable backtracking. For example, we may wish to *commit* to a choice, or we know that trying alternatives is fruitless. A non-backtracking regexp is enclosed in `(?>...)`.

```
(pregexp-match "(?>a+)." "aaaa") ⇒ #f
```

In this call, the subregexp `?>a+` greedily matches all four `a`'s, and is denied the opportunity to backtrack. So the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

4.3.7 Looking Ahead and Behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These look-around assertions are specified by putting the subpattern checked for in a cluster whose leading characters are `?=` for positive look-ahead, `?!` for negative look-ahead,

`?<=` for positive look-behind, and `?<!` for negative look-behind. Note that the subpattern in the assertion does not generate a match in the final result. It merely allows or disallows the rest of the match.

Look-ahead

Positive look-ahead (`?=`) peeks ahead to ensure that its subpattern *could* match.

```
(pregexp-match-positions "grey(?=hound)" "i left my grey socks at the greyhound")
⇒ ((28 . 32))
```

The regexp `"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative look-ahead (`?!`) peeks ahead to ensure that its subpattern could not possibly match.

```
(pregexp-match-positions "grey(?!hound)" "the gray greyhound ate the grey socks")
⇒ ((27 . 31))
```

The regexp `"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

Look-behind

Positive look-behind (`?<=`) checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
(pregexp-match-positions "(?<=grey)hound" "the hound is not a greyhound")
⇒ ((23 . 28))
```

The regexp `(?<=grey)hound` matches `hound`, but only if it is preceded by `grey`.

Negative look-behind (`?<!`) checks that its subpattern could not possibly match immediately to the left.

```
(pregexp-match-positions "(?<!grey)hound" "the greyhound is not a hound")
⇒ ((23 . 28))
```

The regexp `(?<!grey)hound` matches `hound`, but only if it is *not* preceded by `grey`.

Look-aheads and look-behinds can be convenient when they are not confusing.

Chapter 5

Heap Library

5.1 Introduction

A *heap* is a tree-based data structure in which no node is less than its parent according to a given node-comparison function. Heaps are often used to implement priority queues efficiently.

This implementation uses a binary tree stored in a vector a such that $a[0]$ is a minimal element, and $a[i]$ has children $a[2i+1]$ and $a[2i+2]$. When the vector is full, its size is doubled, and when it is less than one quarter full, its size is halved. This technique provides $O(\log n)$ average performance for insert and delete operations.

5.2 Programming Interface

`(make-heap` *value<? [min-size]*`)` **procedure**

returns: a heap

The `make-heap` procedure creates an empty heap that uses the *value<?* procedure to compare two values. (*value<? a b*) should return true if and only if a is less than b. The optional *min-size* argument specifies the minimum vector size and must be a fixnum greater than 2. It defaults to 3.

`(heap?` *x*`)` **procedure**

returns: `#t` if x is a heap, `#f` otherwise

`(heap-delete-min!` *h*`)` **procedure**

returns: a minimal element of heap h

The `heap-delete-min!` procedure deletes a minimal element of heap h and returns it. If h is empty, exception `heap-empty` is raised.

`(heap-insert!` *h x*`)` **procedure**

returns: unspecified

The `heap-insert!` procedure inserts element x into heap h.

`(heap-min` *h*`)` **procedure**

returns: a minimal element of heap *h*

The `heap-min` procedure returns a minimal element of heap *h*. If *h* is empty, exception `heap-empty` is raised.

`(heap-min-size` *h*`)` **procedure**

returns: the minimum vector size of heap *h*

`(heap-size` *h*`)` **procedure**

returns: the number of elements in heap *h*

`(heap-value<?` *h*`)` **procedure**

returns: the *value<?* procedure of heap *h*

Chapter 6

OOP Library

The `(swish oop)` library provides a simple object system with classes that support single inheritance. It extends Scheme's record system with instance and virtual instance methods. The virtual function table is stored in the record-type descriptor to avoid any extra storage in each class instance.

```
(define-class name                                                    syntax
  [(parent parent)]
  [(fields ({immutable | mutable} field) ...)]
  [(protocol protocol)]
  (method (method formal ...) body ...) ...
  (virtual (virtual formal ...) body ...) ...
  )
```

expands to:

```
(module ((name implicit-export ...))
  (meta define ctcls make-ctcls)
  (define-syntax name (make-class-dispatcher ctcls))
  (define-property name class-key ctcls)
  (define ($name.virtual.arity.impl inst formal ...)
    (open-instance name inst body ...)) ...
  (define ($name.override.arity.impl inst formal ...)
    (open-instance name inst body ...)) ...
  (define name.rtd (make-class-rtd name))
  (define name.rcd
    (make-record-constructor-descriptor name.rtd parent-rcd protocol))
  (define name.make (record-constructor name.rcd))
  (define (name.field inst)
    (record-check 'field inst name.rtd)
    (object-ref inst offset)) ...
  (define (name.mutable-field.set! inst val)
    (record-check 'mutable-field inst name.rtd)
    (object-set! inst offset val)) ...
  (define ($name.method.arity inst formal ...)
    (open-instance name inst body ...)) ...
```

```
(define (name.method.arity inst formal ...)
  (record-check 'method inst name.rtd)
  ($name.method.arity inst formal ...)) ...
(define ($name.virtual.arity inst formal ...)
  ((object-ref (record-rtd inst) offset) inst formal ...)) ...
(define (name.virtual.arity inst formal ...)
  (record-check 'virtual inst name.rtd)
  ($name.new-virtual.arity inst formal ...)) ...
)
```

The `define-class` macro defines a class, *name*, a macro described in Figure 6.2. The optional `parent` clause specifies the parent class. The class inherits all fields and methods from its parent. The default parent is a hidden root class with no fields or methods.

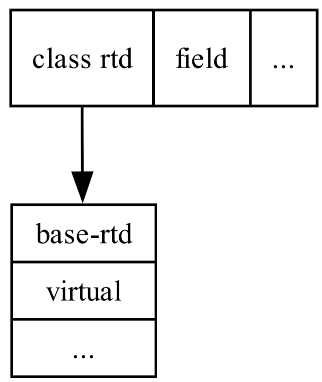
Figure 6.1 diagrams the layout of class records.

The optional `fields` clause specifies the set of instance fields, which defaults to the empty set. Within a class, no two fields may have the same name, and the set of field names must be disjoint from the set of method names. A field shadows any inherited field or method of the same name.

The `protocol` clause specifies the instance constructor in the form used by Scheme's `define-record-type` for record types with parents. A protocol is required when the parent class has a protocol.

Each `method` clause specifies a non-virtual instance method, and each `virtual` clause specifies a virtual instance method. The body expressions are evaluated in *open-instance* scope, described in Figure 6.3. Within a class, no two instance methods may have the same name and arity, and the set of instance method names must be disjoint from the set of field names. An instance method shadows any inherited field of the same name. An instance method shadows—or overrides in the case of virtual methods—any inherited instance method of the same name and arity.

No field or method may be named any of the following: `base`, `isa?`, `make`, and `this`.



The record-type descriptor for the class record is extended with a field for each virtual method.

Figure 6.1: Class records

The `define-class` macro defines a *name* macro that behaves according to the following chart:

syntax	**returns**
(*name* `isa?` *e*)	(`record?` *e* *name*`.rtd`)
(*name* `make` *arg* ...)	(*name*`.make` *arg* ...)
(*name* *field* *inst*)	(*class*`.`*field* *inst*)
(*name* *mutable-field* *inst* *val*)	(*class*`.`*mutable-field*`.set!` *inst* *val*)
(*name* *method* *inst* *arg* ...)	(*class*`.`*method*`.`*arity* *inst* *arg* ...)

- *field* includes every field and unshadowed inherited field.
- *mutable-field* includes every mutable field and unshadowed inherited mutable field.
- *method* includes every instance method and unshadowed inherited instance method.
- *class* is the class that defined the field or method.

Figure 6.2: Description of class *name* macro

The *open-instance* macro introduces a scope with the following definitions:

syntax	**returns**
`this`	`inst`
`(this` *field*`)`	`(`*object-ref* `inst` *offset*`)`
`(this` *mutable-field* *val*`)`	`(`*object-set!* `inst` *offset* *val*`)`
`(this` *method* *arg* `...)`	`($`*class*`.`*method*`.`*arity* `inst` *arg* `...)`
`(base` *base-field*`)`	`(`*object-ref* `inst` *offset*`)`
`(base` *base-mutable-field* *val*`)`	`(`*object-set!* `inst` *offset* *val*`)`
`(base` *base-nonvirt* *arg* `...)`	`($`*class*`.`*base-nonvirt*`.`*arity* `inst` *arg* `...)`
`(base` *base-virtual* *arg* `...)`	`($`*class*`.`*base-virtual*`.`*arity*`.impl inst` *arg* `...)`
field	`(`*object-ref* `inst` *offset*`)`
`(set!` *mutable-field* *val*`)`	`(`*object-set!* `inst` *offset* *val*`)`
`(`*method* *arg* `...)`	`($`*class*`.`*method*`.`*arity* `inst` *arg* `...)`

- *field* includes every field and unshadowed inherited field.
- *mutable-field* includes every mutable field and unshadowed inherited mutable field.
- *method* includes every instance method and unshadowed inherited instance method.
- *base-field* includes every parent-class field and unshadowed parent-class-inherited field.
- *base-mutable-field* includes every parent-class mutable field and unshadowed parent-class-inherited mutable field.
- *base-nonvirt* includes every parent-class non-virtual instance method and unshadowed parent-class-inherited non-virtual instance method.
- *base-virtual* includes every parent-class virtual instance method and unshadowed parent-class-inherited virtual instance method.
- *class* is the class that defined the method.

Figure 6.3: Description of *open-instance* scope

Chapter 7

Stream Library

7.1 Introduction

Streams facilitate composition of higher-order functions like map, filter, and fold, while minimizing allocation of intermediate data structures. The `s/>` procedure allows expressions analogous to those found in $\mathrm{C}^\sharp$, Ruby, and other popular languages, in which a stream of values is passed through a pipeline of operators.

A *stream* is a procedure of zero arguments that advances the position within the stream and returns the next value or the end-of-stream object *eos*.

A *transformer* is a procedure that takes a single argument and returns a value. The argument and return value are often streams. If a transformer requires its argument to be a stream, it is responsible for validating or coercing it. A well-behaved transformer can be applied safely to multiple streams or repeatedly to the same stream. All transformers defined in this library are well-behaved.

A *transformer constructor* is a procedure that returns a transformer. `s/map` is one such constructor. `(s/map f)` returns a transformer that maps stream values with procedure `f`.

Two transformers can be composed into a new transformer. The `s/>` procedure composes zero or more transformers and then applies them to an argument.

```
> (s/> '(1 2 3) (s/map 1+))
(2 3 4)
```

7.2 Basics

`(s/>` *x t* ...`)` **procedure**

returns: see below

The `s/>` procedure applies transformers *t* ... to *x* in order. If *x* is a list, vector, or hashtable, it is converted to a stream. The final transformation result is passed to `unstream` before returning.

`s/>` is pronounced “stream pipe.”

(stream *x* ...) **procedure**

returns: a stream

The `stream` procedure returns a stream containing values *x* ....

(list->stream *list*) **procedure**

returns: a stream

The `list->stream` procedure returns a stream containing the values in *list*, in order.

(vector->stream *vector*) **procedure**

returns: a stream

The `vector->stream` procedure returns a stream containing the values in *vector*, in order.

(hashtable->stream *hashtable*) **procedure**

returns: a stream

The `hashtable->stream` procedure returns a stream containing the cells of *hashtable* in no particular order. If the hashtable is immutable, a mutable copy is made first.

(stream->list *stream*) **procedure**

returns: a list

The `stream->list` procedure returns a list containing the values in a finite *stream*, in order.

(stream->vector *stream*) **procedure**

returns: a vector

The `stream->vector` procedure returns a vector containing the values in a finite *stream*, in order.

(stream-unfold *procedure state*) **procedure**

returns: a stream

The `stream-unfold` procedure returns a stream formed by the initial *state* and repeated calls to *procedure*. *procedure* is called with the current state and should return zero, one, or two values. Zero values indicate the end of the stream. One value indicates the last value in the stream. Two values indicate the next value and the next state.

```
> (define (numbers-from n)
    (stream-unfold (lambda (n) (values n (+ n 1))) n))
> (s/> (numbers-from 1) (s/take 3))
(1 2 3)

> (define (range a b step)
    (stream-unfold
     (lambda (n)
       (cond
        [(< n b) (values n (+ n step))]
```

```
        [(= n b) (values n)]
        [else (values)]))
     a))
> (stream->list (range 10 30 5))
(10 15 20 25 30)
```

`(stream-repeat` *n procedure*`)` **procedure**

returns: a stream

The `stream-repeat` procedure returns a stream of *n* values. The i^{th} value is `(`*procedure i*`)`. *procedure* is invoked in order with *i* starting at 0.

```
> (stream->list (stream-repeat 5 values))
(0 1 2 3 4)
```

`(stream-lift` *p*`)` **procedure**

returns: a procedure

The `stream-lift` procedure returns a procedure that converts its argument with `unstream` and then passes it to *p*. It can be used to include a non-stream-aware procedure in a stream pipeline.

`(unstream` *x*`)` **procedure**

returns: see below

If *x* is a stream, it returns `(stream->list` *x*`)`. If *x* is a stream box (see `s/stream`), it is unboxed. Otherwise, it returns *x*.

7.3 Transformers

`(s/all?` *procedure*`)` **procedure**

returns: a transformer

The `s/all?` procedure returns a transformer that returns true if and only if *procedure* returns true for all values in the finite stream.

`s/any` **variable**

The `s/any` variable is a transformer that returns false if the stream is empty, and the stream itself if non-empty.

`(s/any?` [*procedure*]`)` **procedure**

returns: a transformer

The `s/any?` procedure returns a transformer that returns true if and only if the stream has at least one value. If *procedure* is provided, the stream is first filtered with *procedure*.

(s/by-key *procedure* [*hash equiv?*]**)** — **procedure**

returns: a transformer

The `s/by-key` procedure returns a transformer that indexes the values of a finite stream by key function *procedure* into a hashtable created with `(make-hashtable` *hash equiv?*`)`. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1). If two values v_1 and v_2 have the same key k, error `#(duplicate-key` k v_1 v_2`)` is raised.

Key Type	Constructor
fixnum, record, or boolean	`(make-eq-hashtable)`
non-fixnum number or char	`(make-eqv-hashtable)`
string	`(make-hashtable string-hash string=?)`
symbol	`(make-hashtable symbol-hash eq?)`
else	`(make-hashtable equal-hash equal?)`

Table 7.1: Hashtable Type Inference

s/cells — **variable**

The `s/cells` variable is a transformer that returns a stream of cells of the given hashtable in no particular order. If the input value is not a hashtable, it is returned. If the hashtable is immutable, a mutable copy is made first.

(s/chunk *procedure***)** — **procedure**

returns: a transformer

The `s/chunk` procedure returns a transformer that groups stream values into lists based on the return value of *procedure*. Consecutive values v ... for which `(procedure v)` returns the same value are grouped into a list. Return values are compared with `equal?`.

```
> (s/> '(1 3 4) (s/chunk odd?))
((1 3) (4))
> (s/> '("one" "two" "three" "four" "five" "six") (s/chunk string-length))
(("one" "two") ("three") ("four" "five") ("six"))
```

(s/chunk2 *procedure***)** — **procedure**

returns: a transformer

The `s/chunk2` procedure returns a transformer that groups stream values into lists based on the return value of *procedure*. Consecutive value pairs v_1 and v_2 for which `(procedure` v_1 v_2`)` returns the same value are grouped into a list. Return values are compared with `equal?`.

```
> (s/> '(2 3 4 3 2 1 2 3 4) (s/chunk2 <=))
((2 3 4) (3 2 1) (2 3 4))
```

(s/chunk-every *n***)** — **procedure**

returns: a transformer

The `s/chunk-every` procedure returns a transformer that groups stream values into lists of length n. The last list length L is $0 < L \leq n$. n must be an exact, positive integer.

```
> (s/> (iota 5) (s/chunk-every 2))
((0 1) (2 3) (4))
```

`s/concat` **variable**

The `s/concat` variable is a transformer that concatenates a heterogeneous stream of streams, lists, or vectors.

```
> (s/> (list '(1 2) '#(3 4) (stream 5 6)) s/concat)
(1 2 3 4 5 6)
```

`(s/count` *procedure*`)` **procedure**
returns: a transformer

The `s/count` procedure returns a transformer that applies *procedure* to each value in a finite stream and sums the results, treating false as 0 and other non-numbers as 1. For example, if *procedure* returns a boolean, `s/count` counts the number of values for which *procedure* returns true.

```
> (s/> '(1 2 3) (s/count odd?))
2
> (s/> '("1" "two" "3") (s/count string->number))
4
```

`(s/do` *procedure*`)` **procedure**
returns: a transformer

The `s/do` procedure returns a transformer that invokes *procedure* on every value in the stream, in order, and returns an unspecified value if the stream is finite.

`(s/drop` *n*`)` **procedure**
returns: a transformer

The `s/drop` procedure returns a transformer that returns the n^{th} tail of a stream. n must be an exact integer. If n is negative, it returns the input stream unchanged. If n is greater than or equal to the length of the stream, it returns the empty stream.

`(s/drop-last` *n*`)` **procedure**
returns: a transformer

The `s/drop-last` procedure returns a transformer that returns a stream containing all but the last n values in a finite input stream. n must be an exact integer. If n is negative, it returns the input stream unchanged. If n is greater than or equal to the length of the stream, it returns the empty stream.

`(s/drop-while` *procedure*`)` **procedure**

returns: a transformer

The `s/drop-while` procedure returns a transformer that returns the tail of the stream starting with the first value for which *procedure* returns false. If *procedure* returns a true value for all values, it returns the empty stream.

```
> (s/> '(1 3 4 5) (s/drop-while odd?))
(4 5)
```

`(s/extrema` [*lt gt*]`)` **procedure**

returns: a transformer

The `s/extrema` procedure returns a transformer that performs `s/min` and `s/max` simultaneously, returning the list `(`*min max*`)`, or false if the stream is empty.

`(s/extrema-by` *procedure* [*lt gt*]`)` **procedure**

returns: a transformer

The `s/extrema-by` procedure returns a transformer that performs `s/min-by` and `s/max-by` simultaneously, returning the list `(`*min max*`)`, or false if the stream is empty.

`(s/filter` [*procedure*]`)` **procedure**

returns: a transformer

The `s/filter` procedure returns a transformer that filters stream values by *procedure*. If *procedure* is not provided, the identity procedure is used.

`(s/find` *procedure*`)` **procedure**

returns: a transformer

The `s/find` procedure returns the transformer composition of `(s/filter` *procedure*`)` and `s/first`.

`s/first` **variable**

The `s/first` variable is a transformer that returns the first value in the stream, or false if the stream is empty.

`(s/fold-left` *procedure acc*`)` **procedure**

returns: a transformer

The `s/fold-left` procedure returns a transformer that folds *procedure* over a finite stream with initial accumulator *acc*. The semantics are the same as `fold-left`.

`(s/fold-right` *procedure acc*`)` **procedure**

returns: a transformer

The `s/fold-right` procedure returns a transformer that folds *procedure* over a finite stream with initial accumulator *acc*. The semantics are the same as `fold-right`.

(s/group-by *procedure* [*hash equiv?*]**)** **procedure**
returns: a transformer

The `s/group-by` procedure returns a transformer that groups the values of a finite stream by key function *procedure* into a hashtable created with `(make-hashtable` *hash equiv?*`)`. The value for key k is the list of stream values for which *procedure* returns k, in order of their appearance in the stream. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1).

(s/ht [*fk fv* [*hash equiv?*]]**)** **procedure**
returns: a transformer

The `s/ht` procedure returns a transformer that maps each value x in a finite stream to a cell with key `(`*fk x*`)` and value `(`*fv x*`)` in a hashtable created with `(make-hashtable` *hash equiv?*`)`. If *fk* and *fv* are not provided, `car` and `cdr` are used, respectively. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1). If two values v_1 and v_2 have the same key k, error `#(duplicate-key` k v_1 v_2`)` is raised.

```
> (s/> '(1 2 3)
    (s/ht 1- 1+)
    s/cells
    (s/sort-by car <))
((0 . 2) (1 . 3) (2 . 4))
```

(s/ht* [*fk fv* [*hash equiv?*]]**)** **procedure**
returns: a transformer

The `s/ht*` procedure returns a transformer that maps each value in a finite stream into a hashtable created with `(make-hashtable` *hash equiv?*`)`. Stream value x is mapped to key `(`*fk x*`)`. The value for a key is the list of stream values with that key, in order of their appearance in the stream, mapped with *fv*. If *fk* and *fv* are not provided, `car` and `cdr` are used, respectively. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1).

```
> (s/> '("a" "b" "cc")
    (s/ht* string-length string->symbol)
    s/cells
    (s/sort-by car <))
((1 a b) (2 cc))
```

(s/ht** [*fk fv* [*hash equiv?*]]**)** **procedure**
returns: a transformer

The `s/ht**` procedure returns a transformer that maps each value in a finite stream into a hashtable created with `(make-hashtable` *hash equiv?*`)`. Stream value x is mapped to key `(`*fk x*`)`. The value for a key is the list of stream values with that key, in order of their appearance in the stream, map-concatenated with *fv*. If *fk* and *fv* are not provided, `car` and `cdr` are used, respectively. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1).

```
> (s/> '(("a" 1 2) ("b" 3 4) ("a" 5 6))
    (s/ht** car cdr)
    s/cells
    (s/sort-by car string<?))
(("a" 1 2 5 6) ("b" 3 4))
```

`(s/join` *sep* [*sep-2* *sep-last*]`)` **procedure**

returns: a transformer

The `s/join` procedure returns a transformer that joins the values of a finite stream into a string, separated by string *sep*. If strings *sep-2* and *sep-last* are provided, *sep-2* is used to separate a stream of two values, and *sep-last* is used as the last separator for a stream of three or more values. Values are written to the string as with `display`. It returns the empty string if the stream is empty.

```
> (s/> '(a b c) (s/join ", "))
"a, b, c"
> (s/> '(a b c) (s/join ", " " and " ", and "))
"a, b, and c"
> (s/> '(a b) (s/join ", " " and " ", and "))
"a and b"
```

`(s/json-object` [*fk* *fv*]`)` **procedure**

returns: a transformer

The `s/json-object` procedure returns a transformer that maps each value x in a finite stream to a cell with key `(`*fk* x`)` and value `(`*fv* x`)` in a hashtable created with `(json:make-object)`. Procedure *fk* should return a symbol or string. If a string is returned, it is converted to a symbol. If *fk* and *fv* are not provided, `car` and `cdr` are used, respectively. If two values v_1 and v_2 have the same key k, error `#(duplicate-key` k v_1 v_2`)` is raised. If *fk* returns a non-symbol, non-string value k, error `#(invalid-key` k`)` is raised.

`s/last` **variable**

The `s/last` variable is a transformer that returns the last value in a finite stream, or false if the stream is empty.

`s/length` **variable**

The `s/length` variable is a transformer that returns the number of values in a finite stream.

`(s/map` *procedure*`)` **procedure**

returns: a transformer

The `s/map` procedure returns a transformer that maps stream values with *procedure*.

`(s/map-car` f`)` **procedure**

returns: a transformer

The `s/map-car` procedure returns a transformer that maps each pair `(`x `.` y`)` to `(`$f(x)$ `.` y`)`.

```
> (s/> '((a . 1) (b . 2)) (s/map-car symbol->string))
(("a" . 1) ("b" . 2))
```

`(s/map-cdr` *f*`)` **procedure**
returns: a transformer

The `s/map-cdr` procedure returns a transformer that maps each pair `(`x `.` y`)` to `(`x `.` $f(y)$`)`.

```
> (s/> '((a . 1) (b . 2)) (s/map-cdr 1+))
((a . 2) (b . 3))
```

`(s/map-concat` *procedure*`)` **procedure**
returns: a transformer

The `s/map-concat` procedure returns the transformer composition of `(s/map` *procedure*`)` and `s/concat`.

`(s/map-cons` *f*`)` **procedure**
returns: a transformer

The `s/map-cons` procedure returns a transformer that maps each value x to the pair `(`x `.` $f(x)$`)`.

```
> (s/> '(1 2) (s/map-cons -))
((1 . -1) (2 . -2))
```

`(s/map-extrema` *procedure* [*lt gt*]`)` **procedure**
returns: a transformer

The `s/map-extrema` procedure returns a transformer that performs `s/map-min` and `s/map-max` simultaneously, returning the list `(`*min max*`)`, or false if the stream is empty.

`(s/map-filter` *procedure*`)` **procedure**
returns: a transformer

The `s/map-filter` procedure returns the transformer composition of `(s/map` *procedure*`)` and `(s/filter)`.

```
> (s/> '("1" "two" "3") (s/map-filter string->number))
(1 3)
```

`(s/map-max` *procedure* [*gt*]`)` **procedure**
returns: a transformer

The `s/map-max` procedure returns the transformer composition of `(s/map` *procedure*`)` and `(s/max` [*gt*]`)`.

`(s/map-min` *procedure* [*lt*]`)` **procedure**

returns: a transformer

The `s/map-min` procedure returns the transformer composition of `(s/map` *procedure*`)` and `(s/min` [*lt*]`)`.

`(s/map-uniq` *procedure* [*hash equiv?*]`)` **procedure**

returns: a transformer

The `s/map-uniq` procedure returns the transformer composition of `(s/map` *procedure*`)` and `(s/uniq` [*hash equiv?*]`)`.

`(s/max` [*gt*]`)` **procedure**

returns: a transformer

The `s/max` procedure returns a transformer that returns the maximum value in a finite stream according to greater-than operator *gt*. If *gt* is omitted, numeric greater-than (`>`) is used. If the stream contains multiple equivalent maxima, the first is returned. If the stream is empty, it returns false.

`(s/max-by` *procedure* [*gt*]`)` **procedure**

returns: a transformer

The `s/max-by` procedure returns a transformer that returns the value x in a finite stream that maximizes `(`*procedure* x`)` according to greater-than operator *gt*. If *gt* is omitted, numeric greater-than (`>`) is used. If the stream contains multiple equivalent maxima, the first is returned. If the stream is empty, it returns false.

`(s/mean` [*procedure*]`)` **procedure**

returns: a transformer

The `s/mean` procedure returns a transformer that maps values of a finite stream with *procedure* and returns the mean, or NaN if the stream is empty. If *procedure* is omitted, the identity procedure is used.

`(s/min` [*lt*]`)` **procedure**

returns: a transformer

The `s/min` procedure returns a transformer that returns the minimum value in a finite stream according to less-than operator *lt*. If *lt* is omitted, numeric less-than (`<`) is used. If the stream contains multiple equivalent minima, the first is returned. If the stream is empty, it returns false.

`(s/min-by` *procedure* [*lt*]`)` **procedure**

returns: a transformer

The `s/min-by` procedure returns a transformer that returns the value x in a finite stream that minimizes `(`*procedure* x`)` according to less-than operator *lt*. If *lt* is omitted, numeric less-than (`<`) is used. If the stream contains multiple equivalent minima, the first is returned. If the stream is empty, it returns false.

(s/none? [*procedure*]) **procedure**

returns: a transformer

The `s/none?` procedure returns a transformer that returns true if and only if the stream is empty. If *procedure* is provided, the stream is first filtered with *procedure*.

s/reverse **variable**

The `s/reverse` variable is a transformer that reverses a finite stream and returns its values in a list.

(s/sort *lt*) **procedure**

returns: a transformer

The `s/sort` procedure returns a transformer that sorts a finite stream by less-than operator *lt*. It returns a list of the sorted values.

```
> (s/> '(1 3 2) (s/sort <))
(1 2 3)
> (s/> '(1 3 2) (s/sort >))
(3 2 1)
```

(s/sort-by *procedure* *lt*) **procedure**

returns: a transformer

The `s/sort-by` procedure returns a transformer that sorts a finite stream by less-than operator *lt* according to (*procedure* *x*) for each stream value *x*. It returns a list of the sorted values.

```
> (s/> '(-3 -2 1 4) (s/sort-by abs <))
(1 -2 -3 4)
```

s/stream **variable**

The `s/stream` variable is a transformer that boxes its input. Using `s/stream` as the last transformer in a pipeline effectively prevents `s/>` from attempting to convert the result from stream to list, which can improve the performance of nested pipelines.

```
> (s/> chapters
    (s/map-concat
     (lambda (c)
       (s/> (chapter-sections c)
         (s/map (lambda (s) (list (chapter-name c) s)))
         s/stream))))
(("Erlang Embedding" "Introduction")
 ("Erlang Embedding" "Tuples")
 ("Stream Library" "Introduction")
 ("Stream Library" "Transformers"))
```

`s/sum` **variable**

The `s/sum` variable is a transformer that returns the sum of values in a finite stream.

`(s/take` *n*`)` **procedure**

returns: a transformer

The `s/take` procedure returns a transformer that limits a stream to the first *n* values. *n* must be an exact integer. If *n* is negative, it returns the empty stream.

`(s/take-last` *n*`)` **procedure**

returns: a transformer

The `s/take-last` procedure returns a transformer that returns a stream containing the last *n* values in a finite input stream. *n* must be an exact integer. If *n* is negative, it returns the empty stream. If *n* is greater than or equal to the length of the stream, it returns the input stream unchanged.

`(s/take-while` *procedure*`)` **procedure**

returns: a transformer

The `s/take-while` procedure returns a transformer that returns the stream of values up to but not including the first one for which *procedure* returns false. If *procedure* returns a true value for all values, it returns the input stream unchanged.

`(s/uniq` [*hash equiv?*]`)` **procedure**

returns: a transformer

The `s/uniq` procedure returns a transformer that filters out all but the first instance of each duplicate value, preserving order. Duplicates are determined by hash function *hash* and equivalence function *equiv?*, as would be passed to `make-hashtable`. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1).

`(s/uniq-by` *procedure* [*hash equiv?*]`)` **procedure**

returns: a transformer

The `s/uniq-by` procedure returns a transformer that filters out all but the first instance of each value *x* that duplicates `(`*procedure x*`)`, preserving order. Duplicates are determined by hash function *hash* and equivalence function *equiv?*, as would be passed to `make-hashtable`. If *hash* and *equiv?* are not provided, a hashtable appropriate for the first key is created (see Table 7.1).

```
> (s/> '(-1 2 1) (s/uniq-by abs))
(-1 2)
```

7.4 Transformer Helpers

The following procedures and forms may be helpful in creating new transformers.

`eos` **variable**

The `eos` variable is the end-of-stream object.

`(eos?` *x*`)` **procedure**

returns: a boolean

The `eos?` procedure returns true if and only if *x* is the end-of-stream object.

`(define-stream-transformer (`*name s a* `...)` e_1 e_2 `...)` **syntax**

expands to: a definition

The `define-stream-transformer` form defines a transformer or transformer constructor with the given *name* and arguments *a* .... For each transformer input *s*, *s* is rebound to `(require-stream` *s*`)` and *a* ... are rebound to *a* .... This makes it safe for expressions e_1 e_2 ... to assume *s* is a stream and to `set!` *a* ... while preserving transformer semantics. Expressions e_1 e_2 ... should evaluate to the transformed stream.

If there are one or more arguments *a* ..., `define-stream-transformer` defines a transformer constructor.

```
> (define-stream-transformer (s/take s n)
    (lambda ()
      (if (<= n 0)
          eos
          (let ([x (s)])
            (if (eos? x)
                eos
                (begin
                  (set! n (- n 1))
                  x))))))
> (s/> (numbers-from 1) (s/take 3))
(1 2 3)
```

If there are zero arguments *a* ..., `define-stream-transformer` defines a transformer variable.

```
> (define-stream-transformer (s/first s)
    (let ([x (s)])
      (and (not (eos? x)) x)))
> (s/> (numbers-from 1) s/first)
1
```

Avoid creating a transformer that returns a zero-argument procedure that is not a stream, because this library cannot distinguish it from a stream.

`empty-stream` **variable**

The `empty-stream` variable is the empty stream.

`(stream-cons` *x* *stream*`)` **procedure**

returns: a stream

The `stream-cons` procedure returns a stream containing *x* followed by the values of *stream*.

`(stream-cons*` *x* ... *stream*`)` **procedure**

returns: a stream

The `stream-cons*` procedure returns a stream containing *x* ... followed by the values of *stream*.

`(require-stream` *x*`)` **procedure**

returns: a stream

The `require-stream` procedure returns *x* if it is a stream. If *x* is a list, `require-stream` returns `(list->stream` *x*`)`. If *x* is a vector, it returns `(vector->stream` *x*`)`. If *x* is a hashtable, it returns `(hashtable->stream` *x*`)`. Otherwise, error `#(invalid-stream` *x*`)` is raised.

`(transformer-compose` *a* *b*`)` **procedure**

returns: a transformer

The `transformer-compose` procedure returns a new transformer that applies transformer *a* followed by transformer *b*.

`(transformer-compose*` *ts*`)` **procedure**

returns: a transformer

The `transformer-compose*` procedure returns a new transformer that applies the transformers in list *ts* in order.

Chapter 8

Command Line Interface

8.1 Introduction

The command-line interface (`cli`) provides parsing of command-line arguments as well as consistent usage of common options and display of help.

8.2 Theory of Operation

Many programs parse command-line arguments and perform actions based on them. The `cli` library helps to make programs that process arguments and display help simple and consistent. Command-line arguments are parsed left to right in a single pass. Command-line interface specifications, or `cli-specs`, are used for parsing and error checking a command line, displaying one-line usage, and displaying a full help summary.

Arguments may be preceded by a single dash (`-`), a double dash (`--`), or no dash at all. A single dash precedes short, single character arguments. The API does not allow numbers as they could be mistaken as a negative numerical value supplied to another argument. A double dash precedes longer, more descriptive arguments, `--repl` for example. Positional arguments are not preceded by any dashes. As arguments with dashes are consumed, the remaining arguments are matched against the positional specifications in order.

Argument specifications include a type such as: `bool`, `count`, `string`, and `list`. A set of `bool` and `count` arguments can be specified together (`-abc` is equivalent to `-a -b -c`). Arguments of type `list` collect values in left to right order.

The API does not directly support sub-commands and alternate usage help text. These can be implemented using the primitives provided. The implementations of `swish-build` and `swish-test` provide examples of advanced command-line handling.

In the following REPL transcript, we define `example-cli` using `cli-specs`. We then set the `command-line-arguments` parameter as they would be for an application. Calling `parse-command-line-arguments` returns a procedure, `opt`, which we can use to access the parsed command-line values. Finally, we use `display-help` to display the automatically generated help.

```
> (define example-cli
    (cli-specs
     default-help
     [verbose -v count "indicates verbosity level"]
     [output -o (string "<output>") "print output to an <output> file"]
     [repl --repl bool "start a repl"]
     [files (list "<file>" ...) "a list of input files"]))
> (command-line-arguments '("-vvv" "-o" "file.out" "file.in"))
> (define opt (parse-command-line-arguments example-cli))
> (opt 'verbose)
3
> (opt 'output)
"file.out"
> (opt 'files)
("file.in")
> (display-help "sample" example-cli)
Usage: sample [-hv] [-o <output>] [--repl] <file> ...

  -h, --help        display this help and exit
  -v                indicates verbosity level
  -o <output>       print output to an <output> file
  --repl            start a repl
  <file> ...        a list of input files
```

8.3 Programming Interface

(cli-specs **syntax**
  [default-help]
  (*name* [*short*] [*long*] *type* *help*
    [(conflicts *conflicts*)]
    [(requires *requires*)]
    [(usage [*visibility*] [*how*])])
  ...)

expands to:

a list of <arg-spec> tuples

The cli-specs macro simplifies the creation of the <arg-spec> tuples. The <arg-spec> name field uniquely identifies a specification, and is used to retrieve parsed argument values and check constraints.

name: a symbol to identify the argument
short: a symbol of the form -x, where x is a single character, see below
long: a symbol of the form --x, where x is a string
type: see Table 8.1
help: a string or list of strings that describes the argument
conflicts: a list of <arg-spec> names
requires: a list of <arg-spec> names

To specify `-i` or `-I` for *short*, use |`-i`| and |`-I`| respectively to prevent Chez Scheme from reading them as the complex number $0 - 1i$.

Type	Result
`bool`	`#t`
`count`	a positive integer
`(string` *x*`)`	a string
`(list` *x*`)`	a list of one item
`(list` *x* ...`)`	a list of one or more items up to the next argument
`(list .` *x*`)`	a list of the rest of the arguments

For each type where *x* is specified, *x* is a string that is used in the help display.

Table 8.1: Command-line argument types

The `list` types can support multiple *x* arguments, for instance `(list "i1" "i2")` would specify a list of two arguments.

visibility → `show`
| `hide`
| `fit`

When printing the help usage line, a *visiblility* of `show` means the argument must be displayed. `hide` forces the argument to be hidden. `fit` displays the argument if it fits on the line.

how → `short`
| `long`
| `opt`
| `req`

The *how* expands into input of the `format-spec` procedure according to Table 8.2.

Keyword	Expands into:
`short`	`(opt (and short args))`
`long`	`(opt (and long args))`
`opt`	`(opt (and (or short long) args))`
`req`	`(req (and (or short long) args))`

Table 8.2: `cli-specs` *how* field

For options with *short* or *long* specified, `fit` and `opt` are the defaults. For other options, `show` and `req` are the defaults.

conflicts is a list of specification names that prevent this argument from processing correctly. When multiple command-line arguments are specified that are in conflict, an exception is raised.

requires is a list of other specification names that are necessary for this argument to be processed correctly. Unless all the required command-line arguments are specified, an exception is raised.

The `conflicts`, `requires`, and `usage` clauses may be specified in any order.

(display-help *exe-name specs* [*args*] [*op*]**)** **procedure**

returns: unspecified

The `display-help` procedure is equivalent to calling `display-usage` with a prefix of `"Usage:"` followed by `display-options`.

(display-options *specs* [*args*] [*op*]**)** **procedure**

returns: unspecified

For each specification in *specs*, the `display-options` procedure renders two columns of output to *op*, which defaults to the current output port. The first column renders the short and long form of the argument with its additional inputs. The second column renders the `<arg-spec> help` field and will automatically wrap if the `help-wrap-width` is exceeded.

If an *args* hash table is specified, the specified value appended to the second column. This is useful for displaying default or current values.

(display-usage *prefix exe-name specs* [*width*] [*op*]**)** **procedure**

returns: unspecified

The `display-usage` procedure displays the first line of help output to *op*, which defaults to the current output port. It starts with *prefix* and *exe-name* then attempts to fit *specs* onto the line using `format-spec`. When the line will exceed *width* characters, some arguments may collapse to `[options]`.

A *width* of `#f` defaults the line width to `help-wrap-width`.

(format-spec *spec* [*how*]**)** **procedure**

returns: a string

The `format-spec` procedure is responsible rendering *spec* as a string as specified by *how*. `format-spec` can display dashes in front of arguments, ellipses on list types, and brackets around optional arguments. A *how* of `#f` defaults to the `<arg-spec> usage` field.

<table>
<tr><th>how</th><th>Return value:</th></tr>
<tr><td><code>short</code></td><td><code>"-x"</code> if spec short is <code>x</code>, else <code>#f</code></td></tr>
<tr><td><code>long</code></td><td><code>"--x"</code> if spec long is <code>x</code>, else <code>#f</code></td></tr>
<tr><td><code>args</code></td><td>The spec type is evaluated as follows:
<table>
<tr><th>type</th><th>Return value:</th></tr>
<tr><td><code>bool</code></td><td><code>#f</code></td></tr>
<tr><td><code>count</code></td><td><code>#f</code></td></tr>
<tr><td><code>(string</code> x<code>)</code></td><td><code>"x"</code></td></tr>
<tr><td><code>(list</code> x<code>)</code></td><td><code>"x"</code></td></tr>
<tr><td><code>(list</code> x ...<code>)</code></td><td><code>"x ..."</code></td></tr>
<tr><td><code>(list .</code> x<code>)</code></td><td><code>"x ..."</code></td></tr>
</table></td></tr>
<tr><td><code>(or</code> how ...<code>)</code></td><td>Recur and use the first non-<code>#f</code></td></tr>
<tr><td><code>(and</code> how ...<code>)</code></td><td>Recur and concatenate all non-<code>#f</code> values</td></tr>
<tr><td><code>(opt</code> how<code>)</code></td><td>Recur and surround the result with square brackets <code>[]</code> if non-<code>#f</code></td></tr>
<tr><td><code>(req</code> how<code>)</code></td><td>Recur and use the result</td></tr>
</table>

`help-wrap-width` **parameter**

value: a positive fixnum

The `help-wrap-width` parameter specifies the default width for `display-usage` and `display-options`.

`(parse-command-line-arguments` *specs* [*ls*] [*fail*]`)` **procedure**

returns: a procedure

The `parse-command-line-arguments` procedure processes the elements of *ls* from left to right in a single pass. As it scans each x in *ls*, the parser must find a suitable s within *specs*. If a suitable s cannot be found, the parser reports an error by calling *fail*. Based on the type of s, the parser may consume additional elements following x. The type of s determines what data the parser records for that argument. When s is satisfied, the parser continues scanning the remaining elements of *ls*.

The parser returns a procedure p that accepts zero or one argument. When called with no arguments, p returns a hash table that maps the name of each s found while processing *ls* to the data recorded for that argument. When called with the name of an element s in *specs*, p returns the data, if any, recorded for that name in the hash table or else `#f`. If a particular s was not found while processing *ls*, the internal hash table has no entry for the name of s and p returns `#f` when given that name. If called with a name that is not in *specs*, p raises an exception.

The following table summarizes the parser's behavior.

`<arg-spec> type`	extra arguments consumed / recorded	return value of `(`p *name*`)`
`bool`	none	`#t`
`count`	none	an exact positive integer
`(string` x`)`	one	a string
`(list` x_0 `...` x_n `...)`	n or more, up to the next option	a list of strings
`(list` x_0 `...` x_n `.` *rest*`)`	at least n and all remaining	a list of strings

By default *ls* is the value of `(command-line-arguments)` and *fail* is a procedure that applies `errorf` to its arguments. Providing a *fail* procedure allows a developer to accumulate parsing errors without necessarily generating exceptions.

`<arg-spec>` **tuple**

name: a symbol to use as the key of the output hash table
type: see Table 8.1
short: `#f` | a character
long: `#f` | a string
help: a string or list of strings describing argument
conflicts: a list of `<arg-spec>` names
requires: a list of `<arg-spec>` names
usage: a list containing one *visibility* symbol and a `format-spec` *how* expression

Chapter 9

Testing

The `(swish mat)` library provides methods to define, iterate through, and run test cases, and to log the results.

Test cases are called *mats* and consist of a name, a set of tags, and a test procedure of no arguments. The set of mats is stored in reverse order in a single, global list. The list of tags allows the user to group tests or mark them. For example, tags can be used to note that a test was created for a particular change request.

`(mat` *name* `(`*tag* ...`)` e_1 e_2 ...`)` **syntax**

expands to: `(add-mat '`*name* `'(`*tag* ...`) (lambda ()` e_1 e_2 ...`))`

The `mat` macro creates a mat with the given *name*, *tag*s, and test procedure e_1 e_2 ... using the `add-mat` procedure.

`(add-mat` *name tags test*`)` **procedure**

returns: unspecified

The `add-mat` procedure adds a mat to the front of the global list. *name* is a symbol, *tags* is a list, and *test* is a procedure of no arguments.

If *name* is already used, an exception is raised.

`(clear-mats)` **procedure**

returns: unspecified

The `clear-mats` procedure clears the global list of mats.

`(run-mat` *name reporter*`)` **procedure**

returns: see below

The `run-mat` procedure runs the mat of the given *name* by executing its test procedure with an altered exception handler. If the test procedure completes without raising an exception, the mat result is `pass`. If the test procedure raises exception *e*, the mat result is `(fail .` *e*`)`.

After the mat completes, the `run-mat` procedure tail calls `(`*reporter name tags result statistics*`)`.

If no mat with the given *name* exists, an exception is raised.

`(run-mats [`*name*`] ...)`	**syntax**

returns: `#t` if all mats pass, `#f` otherwise

The `run-mats` macro runs each mat specified by symbols *name* .... When no names are supplied, all mats are executed. After each mat is executed, its result, name, and exception if it failed are displayed. When the mats are finished, a summary of the number run, passed, and failed is displayed.

`(run-mats-from-file` *filename*`)`	**procedure**

returns: `#t` if all mats pass, `#f` otherwise

The `run-mats-from-file` procedure clears the global list of mats, loads *filename* into an isolated environment, and calls `(run-mats)`.

`(run-mats-to-file` *filename*`)`	**procedure**

returns: unspecified

The `run-mats-to-file` procedure executes all mats and writes the results into the file specified by the string *filename*. If the file exists, its contents are overwritten. The file format is a sequence of JSON objects readable with `load-results` and `summarize`.

`(for-each-mat` *procedure*`)`	**procedure**

returns: unspecified

The `for-each-mat` procedure calls (*procedure name tags*) for each mat, in no particular order.

`(load-results` *filename*`)`	**procedure**

returns: a JSON object

The `load-results` procedure reads the contents of the file specified by string *filename* and returns a JSON object with the following keys:

`meta-data`	a JSON object
`report-file`	*filename*
`results`	a list of JSON objects

The `meta-data` object contains at least the following keys:

`completed`	`#t` if test suite completed, `#f` otherwise
`date`	`(format-rfc2822 (current-date))` at the start of the test suite
`machine-type`	`(machine-type)` of the host system
`test-file`	name of the file containing the tests

Each result is a JSON object with the following keys:

`message`	error message from failing test, or empty string
`sstats`	a JSON object representing the `sstats-difference` for the test
`stacks`	if test failed with exception *e*, then `(map stack->json (exit-reason->stacks` *e*`))`
`tags`	a list of strings corresponding to the symbolic tags in the `mat` form
`test`	a string corresponding to the symbolic `mat` name
`test-file`	the name of the test file
`type`	the type of result: `"pass"`, `"fail"`, `"skip"`

`(summarize` *files*`)` **procedure**

returns: five values: the number of passing mats, the number of failing mats, the number of skipped mats, the number of completed suites, and the length of *files*.

The `summarize` procedure reads the contents of each file in *files*, a list of string filenames, and returns the number of passing mats, the number of failing mats, the number of skipped mats, the number of completed test suites, and the number of files specified. An error is raised if any entry is malformed.

Bibliography

[1] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] Joe Armstrong. *Programming Erlang—Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[3] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014. `http://www.ietf.org/rfc/rfc7159.txt`.

[4] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly, second edition, 2002.

[5] Ian Hickson. HTML 5, October 2014. `http://www.w3.org/TR/2014/REC-html5-20141028/`.

[6] E. Resnick. Internet Message Format, April 2001. `http://www.ietf.org/rfc/rfc2822.txt`.

[7] Dorai Sitaram. pregexp: Portable Regular Expressions for Scheme and Common Lisp, 2005. `https://ds26gte.github.io/pregexp/`.

[8] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* O'Reilly, third edition, 2000.

List of Figures

List of Tables

Index