Elk — The Extension Language Kit
Scheme Reference

Oliver Laumann

1. Introduction

This reference manual lists the primitive procedures, special forms, and other facilities implemented by the Scheme interpreter included in Elk. This *kernel* functionality can be augmented by applications using Elk as their extension language implementation or by reusable Elk extensions (such as the UNIX or X11 extensions included in the distribution). The predefined Elk extensions and the C/C++ programmer's interface to Elk are described in separate documents.

Only the procedures and special forms that are not defined by the official Scheme language specification "R^{4}RS" (William Clinger and Jonathan Rees (editors), *Revised^4 Report on the Algorithmic Language Scheme*, 1991) are described in detail. The language features that are part of the official language are only mentioned without a description or examples.

2. Lambda Expressions, Procedures

(lambda *formals body***)** **syntax**

See R^{4}RS.

(procedure-lambda *procedure***)** **procedure**

Returns a copy of the *lambda* expression which has been evaluated to create the given procedure. Example:

```
(define (square x) (* x x))
(procedure-lambda square)          ==>  (lambda (x) (* x x))
```

(procedure? *obj***)** **procedure**

See R^{4}RS.

(primitive? *obj***)** **procedure**

Returns #t if *obj* is a primitive procedure, #f otherwise.

(compound? *obj***)** **procedure**

Returns #t if *obj* is a compound procedure (a procedure that has been created by evaluating a lambda expression), #f otherwise.

3. Local Bindings

(let *bindings body***)** **syntax**
(let* *bindings body***)** **syntax**
(letrec *bindings body***)** **syntax**

See R^4RS.

4. Fluid Binding

(fluid-let *bindings body***)** **syntax**

bindings is of the form ((*variable*$_1$ *init1*) ...). The *init*s are temporarily assigned to the *variable*s and the *body* is executed. The variables must be bound in an enclosing scope. When the body is exited normally or by invoking a control point, the old values of the variables are restored. In the latter case, when the control returns back to the body of the fluid-let by invocation of a control point created within the body, the bindings are changed again to the values they had when the body exited.
Examples:

```
((lambda (x)
  (+ x (fluid-let ((x 3)) x))) 1)      ==>  4

(fluid-let ((print-length 2))
  (write '(a b c d)))                  ==>  '(a b ...)

(define (errset thunk)
  (call-with-current-continuation
    (lambda (catch)
      (fluid-let
          ((error-handler
            (lambda msg (catch #f))))
        (list (thunk))))))

(errset (lambda () (+ 1 2)))           ==>  (3)
(errset (lambda () (/ 1 0)))           ==>  #f
```

5. Definitions

(define *variable expression***)** **syntax**
(define *(variable formals) body***)** **syntax**
(define *(variable . formal) body***)** **syntax**

See R^4RS.
Returns a symbol, the identifier that has been bound. Definitions may appear anywhere within a local body (e. g. a lambda body or a *let*). If the *expression* is omitted, *void* (the non-printing object) is used.
Examples:

```
(define nil #f)

(define ((f x) y) (cons x y))
(define (g x) ((f x) 5))
(g 'a)                              ==>  (a . 5)
```

6. Assignment

(set! *variable expression***)** **syntax**

See R^4RS.
Returns the previous value of *variable*.
Examples:

```
(define-macro (swap x y)
  `(set! ,x (set! ,y ,x)))
```

7. Procedure Application

(operator *operand*$_1$ ...**)** **syntax**

See R^4RS. *operator* can be a macro (see below).

(apply *arg*$_1$ ... *args***)** **procedure**

See R^4RS.

8. Quotation, Quasiquotation

(quote *datum***)** **syntax**
'*datum* **syntax**
constant **syntax**

See R^4RS.

(quasiquote *expression***) syntax**
(unquote *expression***) syntax**
(unquote-splicing *expression***) syntax**
See R^4RS.

9. Sequencing

(begin $expression_1$ $expression_2$ ...**) syntax**
See R^4RS.

(begin1 $expression_1$ $expression_2$ ...**) syntax**
Identical to *begin*, except that the result of the first *expression* is returned.

10. Conditionals

(if *test consequent alternate***) syntax**
(if *test consequent***) syntax**
See R^4RS.
In the first form, *alternate* can be a sequence of expressions (implicit *begin*).

(case *key* $clause_1$ $clause_2$ ...**) syntax**
See R^4RS.
Each *clause* not beginning with *else* can be of the form

(($datum_1$ ...) $expression_1$ $expression_2$ ...)

or

(*datum* $expression_1$ $expression_2$ ...)

In the latter case, the *key* is matched against the *datum*.

(cond $clause_1$ $clause_2$ ...**) syntax**
See R^4RS.

(and $test_1$ ...**) syntax**
(or $test_1$ ...**) syntax**
See R^4RS.

11. Booleans

(not *obj***)** **procedure**

See R^4RS.

(boolean? *obj***)** **procedure**

See R^4RS.

12. Iteration

(let *variable bindings body***)** **syntax**

"Named *let*". See R^4RS.

(map *procedure* $list_1$ $list_2$ *...***)** **procedure**
(for-each *procedure* $list_1$ $list_2$ *...***)** **procedure**

See R^4RS. *for-each* returns the empty list.

(do *initializations test body***)** **syntax**

See R^4RS.

13. Continuations

(call-with-current-continuation *procedure***)** **procedure**

See R^4RS.

(control-point? *obj***)** **procedure**

Returns #t if *obj* is a control point (a continuation), #f otherwise.

(dynamic-wind *thunk thunk thunk***)** **procedure**

dynamic-wind is a generalization of the *unwind-protect* facility provided by many Lisp systems. All three arguments are procedures of no arguments. In the normal case, all three thunks are applied in order. The first thunk is also applied when the body (the second thunk) is entered by the application of a control point created within the body (by means of *call-with-current-continuation*). Similarly, the third thunk is also applied whenever the body is exited by invocation of a control point created outside the body.
Examples:

```
(define-macro (unwind-protect body . unwind-forms)
  `(dynamic-wind
    (lambda () #f)
    (lambda () ,body)
    (lambda () ,@unwind-forms)))
```

```
(let ((f (open-input-file "foo")))
  (dynamic-wind
    (lambda () #f)
    (lambda () do something with f)
    (lambda () (close-input-port f))))
```

14. Delayed Evaluation

(delay *expression***)** **syntax**
(force *promise***)** **procedure**
See R^4RS.

(promise? *obj***)** **procedure**

Returns #t if *obj* is a promise, an object returned by the application of *delay*. Otherwise #f is returned.

15. Equivalence Predicates

(eq? obj_1 obj_2**)** **procedure**
(eqv? obj_1 obj_2**)** **procedure**
(equal? obj_1 obj_2**)** **procedure**
See R^4RS.

16. Pairs and Lists

(cons obj_1 obj_2**)** **procedure**
See R^4RS.

(car *pair***)** **procedure**
(cdr *pair***)** **procedure**
See R^4RS.

(cxr *pair pattern***)** **procedure**

pattern is either a symbol or a string consisting of a combination of the characters ‘a’ and ‘d’. It encodes a sequence of *car* and *cdr* operations; each ‘a’ denotes the application of *car*, and each ‘d’ denotes the application of *cdr*. For example, *(cxr p "ada")* is equivalent to *(cadar p)*.

(caar *pair***)** **procedure**

...

(cddddr *pair***)** **procedure**

See R^4RS.

(set-car! *pair obj***)** **procedure**
(set-cdr! *pair obj***)** **procedure**

See R^4RS.
Both procedures return *obj*.

(make-list *k obj***)** **procedure**

Returns a list of length *k* initialized with *obj*.
Examples:

```
(make-list 0 'a)                ==>  ()
(make-list 2 (make-list 2 1))   ==>  ((1 1) (1 1))
```

(list *obj ...***)** **procedure**

See R^4RS.

(length *list***)** **procedure**

See R^4RS.

(list-ref *list k***)** **procedure**

See R^4RS.

(list-tail *list k***)** **procedure**

See R^4RS.

(last-pair *list***)** **procedure**

See R^4RS.

(append *list ...***)** **procedure**

See R^4RS.

(append! *list ...***)** **procedure**

Like *append*, except that the original arguments are modified (destructive *append*). The cdr of each argument is changed to point to the next argument.
Examples:

```
(define x '(a b))
(append x '(c d))                ==>  (a b c d)
x                                ==>  (a b)
(append! x '(c d))               ==>  (a b c d)
x                                ==>  (a b c d)
```

(reverse *list***)** **procedure**

See R^4RS.

(reverse! *list***)** **procedure**

Destructive *reverse*.

(memq *obj list***)** **procedure**
(memv *obj list***)** **procedure**
(member *obj list***)** **procedure**

See R^4RS.

(assq *obj alist***)** **procedure**
(assv *obj alist***)** **procedure**
(assoc *obj alist***)** **procedure**

See R^4RS.

(null? *obj***)** **procedure**
(pair? *obj***)** **procedure**

See R^4RS.

(list? *obj***)** **procedure**

See R^4RS.

17. Numbers

(= z_1 z_2 ...**)** **procedure**
(< z_1 z_2 ...**)** **procedure**
(> z_1 z_2 ...**)** **procedure**
(<= z_1 z_2 ...**)** **procedure**
(>= z_1 z_2 ...**)** **procedure**

See R^4RS.

(1+ *z***)** **procedure**
(-1+ *z***)** **procedure**

Returns *z* plus 1 or *z* minus 1, respectively.

(1- *z***)** **procedure**

A synonym for *-1+* (for backwards compatibility).

(+ z_1 ...**)** **procedure**
(* z_1 ...**)** **procedure**

See R^4RS.

(- z_1 z_2 ...**)** **procedure**
(/ z_1 z_2 ...**)** **procedure**

See R^4RS.

(zero? *z***)** **procedure**
(positive? *z***)** **procedure**
(negative? *z***)** **procedure**
(odd? *z***)** **procedure**
(even? *z***)** **procedure**
(exact? *z***)** **procedure**
(inexact? *z***)** **procedure**

See R^4RS.

(abs *z***)** **procedure**

See R^4RS.

(quotient n_1 n_2**)** **procedure**
(remainder n_1 n_2**)** **procedure**
(modulo n_1 n_2**)** **procedure**

See R^4RS.

(gcd n_1 ...**)** **procedure**
(lcm n_1 ...**)** **procedure**

See R^4RS.

(floor *x***)** **procedure**
(ceiling *x***)** **procedure**
(truncate *x***)** **procedure**
(round *x***)** **procedure**

See R^4RS.

(sqrt *z***)** **procedure**

See R^4RS.

(expt z_1 z_2**)** **procedure**

See R^4RS.

(exp *z***)** **procedure**
(log *z***)** **procedure**
(sin *z***)** **procedure**
(cos *z***)** **procedure**
(tan *z***)** **procedure**
(asin *z***)** **procedure**
(acos *z***)** **procedure**
(atan *z***)** **procedure**
(atan *y x***)** **procedure**

See R^4RS.

(min x_1 x_2 ...**)** **procedure**
(max x_1 x_2 ...**)** **procedure**

See R^4RS.

(random) **procedure**

Returns an integer pseudo-random number in the range from 0 to 2^{31}-1.

(srandom *n***)** **procedure**

Sets the random number generator to the starting point *n*. *srandom* returns *n*.

(number? *obj***)** **procedure**
(complex? *obj***)** **procedure**
(real? *obj***)** **procedure**
(rational? *obj***)** **procedure**
(integer? *obj***)** **procedure**

See R^4RS.

(exact–>inexact *z***)** **procedure**
(inexact–>exact *z***)** **procedure**

See R^4RS.

(number−>string *number***)** **procedure**
(number−>string *number radix***)** **procedure**
See R^4RS.

(string−>number *string***)** **procedure**
(string−>number *string radix***)** **procedure**
See R^4RS.

18. Characters

(char−>integer *char***)** **procedure**
(integer−>char *n***)** **procedure**
See R^4RS.

(char-upper-case? *char***)** **procedure**
(char-lower-case? *char***)** **procedure**
See R^4RS.

(char-alphabetic? *char***)** **procedure**
(char-numeric? *char***)** **procedure**
(char-whitespace? *char***)** **procedure**
See R^4RS.

(char-upcase *char***)** **procedure**
(char-downcase *char***)** **procedure**
See R^4RS.

(char=? $char_1$ $char_2$**)** **procedure**
(char<? $char_1$ $char_2$**)** **procedure**
(char>? $char_1$ $char_2$**)** **procedure**
(char<=? $char_1$ $char_2$**)** **procedure**
(char>=? $char_1$ $char_2$**)** **procedure**
See R^4RS.

(char-ci=? $char_1$ $char_2$**)** **procedure**
(char-ci<? $char_1$ $char_2$**)** **procedure**
(char-ci>? $char_1$ $char_2$**)** **procedure**
(char-ci<=? $char_1$ $char_2$**)** **procedure**
(char-ci>=? $char_1$ $char_2$**)** **procedure**
See R^4RS.

(char? *obj***)** **procedure**

See R^4RS.

19. Strings

(string *char ...***)** **procedure**

Returns a string containing the specified characters.
Examples:

```
(string)                          ==>  ""
(string #\a #\space #\b)          ==>  "a b"
```

(string? *obj***)** **procedure**

See R^4RS.

(make-string *k char***)** **procedure**

See R^4RS.

(string-length *string***)** **procedure**

See R^4RS.

(string-ref *string k***)** **procedure**

See R^4RS.

(string-set! *string k char***)** **procedure**

See R^4RS.
Returns the previous value of element *k* of the given string.

(substring *string start end***)** **procedure**

See R^4RS.

(string-copy *string***)** **procedure**

See R^4RS.

(string-append *string ...***)** **procedure**

See R^4RS.

(list–>string *chars***)** **procedure**
(string–>list *string***)** **procedure**

See R^4RS.

(string-fill! *string char***)** **procedure**

See R^4RS.
Returns *string*.

(substring-fill! *string start end char***)** **procedure**

Stores *char* in every element of *string* from *start* (inclusive) to *end* (exclusive). Returns *string*.

(string=? *$string_1$ $string_2$***)** **procedure**
(string<? *$string_1$ $string_2$***)** **procedure**
(string>? *$string_1$ $string_2$***)** **procedure**
(string<=? *$string_1$ $string_2$***)** **procedure**
(string>=? *$string_1$ $string_2$***)** **procedure**

See R^4RS.

(string-ci=? *$string_1$ $string_2$***)** **procedure**
(string-ci<? *$string_1$ $string_2$***)** **procedure**
(string-ci>? *$string_1$ $string_2$***)** **procedure**
(string-ci<=? *$string_1$ $string_2$***)** **procedure**
(string-ci>=? *$string_1$ $string_2$***)** **procedure**

See R^4RS.

(substring? *$string_1$ $string_2$***)** **procedure**
(substring-ci? *$string_1$ $string_2$***)** **procedure**

If *$string_1$* is a substring of *$string_2$*, these procedures return the starting position of the first occurrence of the substring within *$string_2$*. Otherwise #f is returned. *substring-ci?* is the case insensitive version of *substring?*.
Examples:

```
(define s "Hello world")
(substring? "foo" x)            ==>  #f
(substring? "hello" x)          ==>  #f
(substring-ci? "hello" x)       ==>  0
(substring? "!" x)              ==>  11
```

20. Vectors

(vector? *obj***)** **procedure**

See R^4RS.

(make-vector *k***)** **procedure**
(make-vector *k fill***)** **procedure**

See R^4RS.

(vector *obj ...***)** **procedure**

See R^4RS.

(vector-length *vector***)** **procedure**

See R^4RS.

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

See R^4RS.

(vector-set! *vector k obj***)** **procedure**

See R^4RS.
Returns the previous value of element *k* of the vector.

(vector–>list *vector***)** **procedure**
(list–>vector *list***)** **procedure**

See R^4RS.

(vector-fill! *vector fill***)** **procedure**

See R^4RS.
Returns *vector*.

(vector-copy *vector***)** **procedure**

Returns a copy of *vector*.

21. Symbols

(string–>symbol *string***)** **procedure**
(symbol–>string *symbol***)** **procedure**

See R^4RS.

(put *symbol key value***)** **procedure**
(put *symbol key***)** **procedure**

Associates *value* with *key* in the property list of the given symbol. *key* must be a symbol. Returns *key*.
If *value* is omitted, the property is removed from the symbol's property list.

(get *symbol key***)** **procedure**

Returns the value associated with *key* in the property list of *symbol*. *key* must be a symbol. If no value is associated with *key* in the symbol's property list, #f is returned.
Examples:

```
(put 'norway 'capital "Oslo")
(put 'norway 'continent "Europe")
(get 'norway 'capital)           ==>  "Oslo"
```

(symbol-plist *symbol***)** **procedure**

Returns a copy of the property list of *symbol* as an *alist*.
Examples:

```
(put 'norway 'capital "Oslo")
(put 'norway 'continent "Europe")
(symbol-plist 'norway)
   ==>  ((capital . "Oslo") (continent . "Europe"))
(symbol-plist 'foo)              ==>  ()
```

(symbol? *obj***)** **procedure**

See R^{4}RS.

(oblist) **procedure**

Returns a list of lists containing all currently interned symbols. Each sublist represents a bucket of the interpreters internal hash array.
Examples:

```
(define (apropos what)
  (let ((ret ()))
    (do ((tail (oblist) (cdr tail))) ((null? tail))
      (do ((l (car tail) (cdr l))) ((null? l))
        (if (substring? what (symbol->string (car l)))
            (set! ret (cons (car l) ret)))))
    ret))

(apropos "let")       ==>  (let* let letrec fluid-let)
(apropos "make")      ==>  (make-list make-vector make-string)
(apropos "foo")       ==>  ()
```

22. Environments

(the-environment) **procedure**

Returns the current environment.

(global-environment) **procedure**

Returns the global environment (the "root" environment in which all predefined procedures are bound).

(environment−>list *environment***)** **procedure**

Returns a list representing the specified environment. The list is a list of *frames*, each frame is a list of bindings (an *alist*). The car of the list represents the most recently established environment. The list returned by *environment−>list* can contain cycles.
Examples:

```
(let ((x 1) (y 2))
  (car (environment->list
    (the-environment))))            ==>  ((y . 2) (x . 1))

((lambda (foo)
   (caar (environment->list
     (the-environment)))) "abc")    ==>  (foo . "abc")

(eq?
  (car (last-pair (environment->list
    (the-environment))))
  (car (environment->list
    (global-environment))))         ==>  #t
```

(procedure-environment *procedure***)** **procedure**
(promise-environment *promise***)** **procedure**
(control-point-environment *control-point***)** **procedure**

Returns the environment in which the the body of the *procedure* is evaluated, the environment in which a value for the *promise* is computed when *force* is applied to it, or the environment in which the *control-point* has been created, respectively.

(environment? *obj***)** **procedure**

Returns #t if *obj* is an environment, #f otherwise.

23. Ports and Files

Generally, a file name can either be a string or a symbol. If a symbol is given, it is converted into a string by applying *symbol–>string*. A tilde at the beginning of a file name is expanded according to the rules employed by the C-Shell (see *csh*(1)).

Elk adds a third type of ports, *input-output* (bidirectional) ports. Both *input-port?* and *output-port?* return #t when applied to an input-output port, and both input primitives and output primitives may be applied to input-output ports. An input-output port (in fact, *any* port) may be closed with any of the primitives *close-input-port* and *close-output-port*.

The only way to create an input-output-port is by means of the procedure *open-input-output-file*. Extensions may provide additional means to create bidirectional ports.

(call-with-input-file *file procedure***)** **procedure**
(call-with-output-file *file procedure***)** **procedure**
See R^4RS.

(input-port? *obj***)** **procedure**
(output-port? *obj***)** **procedure**
See R^4RS.

(current-input-port) **procedure**
(current-output-port) **procedure**
See R^4RS.

(with-input-from-file *file thunk***)** **procedure**
(with-output-to-file *file thunk***)** **procedure**
See R^4RS.
file can be a string as well as a symbol.

(open-input-file *file***)** **procedure**
(open-output-file *file***)** **procedure**
(open-input-output-file *file***)** **procedure**
See R^4RS.
file can be a string as well as a symbol. *open-input-output-file* opens the file for reading and writing and returns an input-output port; the file must exist and is not truncated.

(close-input-port *port***)** **procedure**
(close-output-port *port***)** **procedure**
See R^4RS.
Calls to *close-input-port* and *close-output-port* are ignored when applied to string ports or to ports connected with the standard input or standard output of the process.

(clear-output-port) **procedure**
(clear-output-port *output-port***)** **procedure**

If the argument is omitted, it defaults to the current output port.
In case of "buffered" output, this procedure is used to discard all characters that have been output to the port but have not yet been sent to the file associated with the port.

(flush-output-port) **procedure**
(flush-output-port *output-port***)** **procedure**

If the argument is omitted, it defaults to the current output port.
In case of "buffered" output, this procedure is used to force all characters that have been output to the port to be printed immediately. This may be necessary to force output that is not terminated with a newline to appear on the terminal. An output port is flushed automatically when it is closed.

(clear-input-port) **procedure**
(clear-input-port *input-port***)** **procedure**

If the argument is omitted, it defaults to the current input port.
In case of "buffered" input, this procedure discards all characters that have already been read from the file associated with the port but have not been processed using *read* or similar procedures.

(port-file-name *port***)** **procedure**

Returns the name of the file associated with *port* if it is a file port, #f otherwise.

(port-line-number) **procedure**

Returns the current line number of a file input port or string input port, i. e. the number of newline characters that have been read from this port plus one. "Unreading" a newline character decrements the line number, but it never drops below one. The result of applying *port-line-number* to an output port is undefined.

(tilde-expand *file***)** **procedure**

If *file* starts with a tilde, performs tilde expansion as described above and returns the result of the expansion (a string); returns *file* otherwise. *file* is a string or a symbol.

(file-exists? *file***)** **procedure**

Returns #t if *file* is accessible, #f otherwise. *file* is a string or a symbol; tilde expansion is not performed.

24. Input

(read) **procedure**
(read *input-port***)** **procedure**

See R^4RS.

(read-char) **procedure**
(read-char *input-port***)** **procedure**

See R^4RS.

(read-string) **procedure**
(read-string *input-port***)** **procedure**

If the argument is omitted, it defaults to the current input port.
Returns the rest of the current input line as a string (not including the terminating newline).

(unread-char *char***)** **procedure**
(unread-char *char input-port***)** **procedure**

If the second argument is omitted, it defaults to the current input port.
Pushes *char* back on the stream of input characters. It is *not* an error for *char* not to be the last character read from the port. It is undefined whether more than one character can be pushed back without an intermittent read operation, and whether a character can be pushed back before something has been read from the port. The procedure returns *char*.

(peek-char) **procedure**
(peek-char *input-port***)** **procedure**

See R^4RS.

peek-char uses *unread-char* to push back the character.

(eof-object? *obj***)** **procedure**

See R^4RS.

(char-ready? *input-port***)** **procedure**

See R^4RS.

char-ready cannot be implemented correctly based on C FILE pointers. In the current version, *char-ready* can return #f although a call to *read-char* would not block.

25. Output

print-length **variable**
print-depth **variable**

These variables are defined in the global environment. They control the maximum length and maximum depth, respectively, of a list or vector that is printed. If one of the variables is not bound to an integer, or if its value exceeds a certain, large maximum value (which is at least 2^20), a default value is taken. The default value for *print-length* is 1000, and the default value for *print-depth* is 20. Negative values of *print-length* and *print-depth* are treated as ‘‘unlimited’’, i. e. output is not truncated.

(write *obj***)** **procedure**
(write *obj output-port***)** **procedure**

See R^4RS.

(display *obj***)** **procedure**
(display *obj output-port***)** **procedure**

See R^4RS.

(write-char *char***)** **procedure**
(write-char *char output-port***)** **procedure**

See R^4RS.

(newline) **procedure**
(newline *output-port***)** **procedure**

See R^4RS.

(print *obj***)** **procedure**
(print *obj output-port***)** **procedure**

If the second argument is omitted, it defaults to the current output port.
Prints *obj* using *write* and then prints a newline. *print* returns *void*.

(format *destination format-string obj ...***)** **procedure**

Prints the third and the following arguments according to the specifications in the string *format-string*. Characters from the format string are copied to the output. When a tilde is encountered in the format string, the tilde and the immediately following character are replaced in the output as follows:

~s is replaced by the printed representation of the next *obj* in the sense of *write*.

~a is replaced by the printed representation of the next *obj* in the sense of *display*.

~~ is replaced by a single tilde.

~% is replaced by a newline.

An error is signaled if fewer *obj*s are provided than required by the given format string. If the format string ends in a tilde, the tilde is ignored.

If *destination* is #t, the output is sent to the current output port; if #f is given, the output is returned as a string; otherwise, *destination* must be an output or input-output port.
Examples:

```
(format #f "Hello world!")        ==>  "Hello world"
(format #f "~s world!" "Hello")   ==>  "\"Hello\" world"
(format #f "~a world!" "Hello")   ==>  "Hello world"
(format #f "Hello~a")             ==>  "Hello!"

(define (flat-size s)
  (fluid-let ((print-length 1000) (print-depth 100))
    (string-length (format #f "~a" s))))

(flat-size 1.5)                   ==>  3
(flat-size '(a b c))              ==>  7
```

26. String Ports

String ports are similar to file ports, except that characters are appended to a string instead of being sent to a file, or taken from a string instead of being read from a file. It is not necessary to close string ports. When an string input port has reached the end of the input string, successive read operations return end-of-file.

(open-input-string *string***)** **procedure**

Returns a new string input port initialized with *string*.
Examples:

```
(define p (open-input-string "Hello world!"))
(read-char p)                    ==>  #\H
(read p)                         ==>  ello
(read p)                         ==>  world!
(read p)                         ==>  end of file

(define p (open-input-string "(cons 'a 'b)"))
(eval (read p))                  ==>  (a . b)
```

(open-output-string) **procedure**

Returns a new string output port.

(get-output-string *string-output-port***)** **procedure**

Returns the string currently associated with the specified string output port. As a side-effect, the string is reset to zero length.
Examples:

```
(define p (open-output-string))
(display '(a b c) p)
(get-output-string p)            ==>  "(a b c)"
(get-output-string p)            ==>  ""

(define (flat-size s)
  (let ((p (open-output-string)))
    (display s p)
    (string-length (get-output-string p))))
```

27. Loading

(load *file***)** **procedure**
(load *file environment***)** **procedure**

Loads a source file or one or more object files. If the file contains source code, the expressions in the file are read and evaluated. If a file contains object code, the contents of the file is linked together with the running interpreter and with additional libraries that are specified by the variable *load-libraries* (see below). Names of object files must have the suffix ‘‘.o’’. *load* returns *void*.

file must be either a string or a symbol or a list of strings or symbols. If it is a list, all elements of the list must be the names of object files. In this case, all object files are linked by a single run of the linker.
If an optional *environment* is specified, the contents of the file is evaluated in this environment instead of the current environment.

Loading of object files is not supported on some platforms. On the platforms where it is supported, the feature *elk:load-object* is provided by the interpreter on startup (see ‘‘Features’’ below).
Example:

```
(fluid-let ((load-noisily? #t))
  (load 'test.scm))
```

load-path **variable**

This variable is defined in the global environment. It is bound to a list of directories in which files to be loaded are searched for. Each element of the list (a string or a symbol) is used in turn as a prefix for the file name passed to *load* until opening succeeds. Elements of *load-path* that are not of type string or symbol are ignored.

If the value of *load-path* is not a list of at least one valid component, or if the name of the file to be loaded starts with ‘‘/’’ or with ‘‘~’’, it is opened directly.

The initial value of *load-path* is a list of the three elements ‘‘.’’ (i. e. the current directory), ‘‘$scheme_dir’’, and ‘‘$lib_dir’’; $scheme_dir and $lib_dir are the directories into which the runtime Scheme files and object files are installed (typically ‘‘/usr/elk/runtime/scm’’ and ‘‘/usr/elk/runtime/obj’’; defined in the installation’s site file).

load-noisily? **variable**

This variable is defined in the global environment. When a file is loaded and the value of *load-noisily?* is true, the result of the evaluation of each expression is printed. The initial value of *load-noisily?* is #f.

load-libraries **variable**

This variable is defined in the global environment. If *load-libraries* is bound to a string, its value specifies additional load libraries to be linked together with an object file that is loaded into the interpreter (see *load* above). Its initial value is ''−lc''.

(autoload *symbol file***)** **procedure**

Binds *symbol* in the current environment (as with *define*). When *symbol* is evaluated the first time, *file* is loaded. The definitions loaded from the file must provide a definition for *symbol* different from *autoload*, otherwise an error is signaled.

file must be either a string or a symbol or a list of strings or symbols, in which case all elements of the list must be the names of object files (see *load* above).

autoload-notify? **variable**

This variable is defined in the global environment. If the value of *autoload-notify?* is true, a message is printed whenever evaluation of a symbol triggers autoloading of a file. *autoload-notify?* is bound to #t initially.

28. Macros

(macro *formals body***)** **syntax**

This special form creates a macro. The syntax is identical to the syntax of *lambda* expressions. When a macro is called, the actual arguments are bound to the formal arguments of the *macro* expression *in the current environment* (they are *not* evaluated), then the *body* is evaluated. The result of this evaluation is considered the *macro expansion* and is evaluated in place of the macro call.

(define-macro *(variable formals) body***)** **syntax**
(define-macro *(variable . formal) body***)** **syntax**

Like *define*, except that *macro* is used instead of *lambda*.
Examples:

```
(define-macro (++ x) `(set! ,x (1+ ,x)))
(define foo 5)
foo          ==>  5
(++ foo)
foo          ==>  6
```

```
(define-macro (while test . body)
  `(let loop ()
     (cond (,test ,@body (loop)))))
```

(macro? *obj***)** **procedure**

Returns #t if *obj* is a macro, #f otherwise.

(macro-body *macro***)** **procedure**

Returns a copy of the *macro* expression which has been evaluated to created the given macro (similar to *procedure-lambda*).
Examples:

```
(define-macro (++ x) `(set! ,x (1+ ,x)))

(macro-body ++)
  ==>  (macro (x) (quasiquote (set! (unquote x) (1+ (unquote x)))))
```

(macro-expand *list***)** **procedure**

If the expression *list* is a macro call, the macro call is expanded.
Examples:

```
(define-macro (++ x) `(set! ,x (1+ ,x)))

(macro-expand '(++ foo))          ==>  (set! foo (1+ foo))
```

The following function can be used to expand *all* macro calls in an expression, i. e. not only at the outermost level:

```
(define (expand form)
  (if (or (not (pair? form)) (null? form))
      form
      (let ((head (expand (car form)))
            (args (expand (cdr form)))
            (result))
        (if (and (symbol? head) (bound? head))
            (begin
              (set! result (macro-expand (cons head args)))
              (if (not (equal? result form))
                  (expand result)
                  result))
              (cons head args)))))
```

29. Error and Exception Handling

error-handler **variable**

This variable is defined in the global environment. When an error occurs or when the procedure *error* is invoked and the variable *error-handler* is bound to a compound procedure (the *error handler*), the interpreter invokes this procedure. The error handler is called with an object (either the first argument that has been passed to *error* or a symbol identifying the primitive procedure that has caused the error), and an error message consisting of a format string and a list of objects suitable to be passed to *format*.

Typically, a user-defined error handler prints the error message and then calls a control point that has been created outside the error handler. If the error handler terminates normally or if *error-handler* is not bound to a procedure, the error message is printed in a default way, and then a *reset* is performed.

interrupt-handler **variable**

This variable is defined in the global environment. When an interrupt occurs (typically as a result of typing the interrupt character on the keyboard), and the variable *interrupt-handler* is bound to a procedure (the *interrupt handler*), this procedure is called with no arguments. If *interrupt-handler* is not bound to a procedure or if the procedure terminates normally, a message is printed, and a *reset* is performed.
Examples:

```
(set! interrupt-handler
  (lambda ()
    (newline)
    (backtrace)
    (reset)))
```

(disable-interrupts) **procedure**
(enable-interrupts) **procedure**

disable-interrupts causes signals to be blocked from delivery to the interpreter; *enable-interrupts* enables delivery of signals. These functions control delivery of keyboard-generated interrupt signals (see *interrupt-handler* above) as well as additional signals used by extensions (such as the alarm signal). The interpreter automatically blocks delivery of signals during critical operations, such as garbage collection. Signals are enabled on startup after initialization has completed.

A call to *enable-interrupts* immediately delivers signals that have been generated while signals were disabled, but blocked signals are not queued. On platforms that support neither POSIX-style nor BSD-style reliable signals, *disable-interrupts* causes signals to be ignored (as opposed to blocking them until the next call to *enable-interrupts*).

Calls to *disable-interrupts* and *enable-interrupts* can be nested. The functions maintain a count indicating the number of calls to *enable-interrupts* that it takes to return from a nested *disable-interrupts* invocation to the topmost level (i. e. to actually enable delivery of signals again). Both functions return this nesting level as an integer.

Example: the following loop ensures that delivery of signals is enabled, regardless of the current nesting depth of *disable-interrupts* calls:

```
(let loop ((intr-level (enable-interrupts)))
     (if (positive? intr-level)
         (loop (enable-interrupts))))
```

dynamic-wind can be used to write a macro *with-interrupts-disabled* to protect a critical section of code from being interrupted by a signal:

```
(define-macro (with-interrupts-disabled . body)
  `(dynamic-wind
     (lambda () (disable-interrupts))
     (lambda () ,@body)
     (lambda () (enable-interrupts))))
```

(error *obj string obj ...***)** **procedure**

Signals an error. The arguments of *error* are passed to the *error-handler*.
Examples:

```
(define (foo sym)
  (if (not (symbol? sym))
      (error 'foo "argument not a symbol: ~s" sym))
  ...
```

top-level-control-point **variable**
(reset) **procedure**

reset performs a reset by calling the control point to which the variable *top-level-control-point* is bound in the global environment. The control point is called with the argument #t. If *top-level-control-point* is not bound to a control point, or does not exist at all, an error message is printed and the interpreter is terminated.
Examples:

```
(if (call-with-current-continuation
      (lambda (x)
        (fluid-let ((top-level-control-point x))
          do something
          #f)))
    (print "Got a reset!"))
```

(exit) **procedure**
(exit *n***)** **procedure**

Terminates the interpreter. The optional argument *n* indicates the exit code; it defaults to zero.

30. Garbage Collection

The interpreter supports two garbage collectors: the stop-and-copy garbage collector that was part of older versions of Elk, and a generational, incremental garbage collector.

If generational garbage collection has been selected, Scheme objects surviving two garbage collections will not be touched again until there is only a certain amount of memory left on the heap, triggering a full garbage collection. Particularly in applications with large amounts of Scheme code or constant data, partial garbage collections run much faster than full garbage collections. In contrast to the stop-and-copy garbage collector, the generational garbage collector is not limited to a pre-allocated amount of heap; it will expand the heap in steps of 1 MB if the free space left after a full garbage collection falls below a certain amount.

Another feature of the generational garbage collector (available on some platforms only) is the ability to do incremental garbage collection. Starting a garbage collection does not interrupt the application until the garbage collector is done. Instead, the collector returns control to the application almost immediately. To synchronize between the garbage collection and the running application, the code makes use of the *mprotect* system call.

(garbage-collect-status *strategy mode***)** **procedure**

garbage-collect-status is used to select a garbage collector and an optional, garbage collector specific mode of operation, and to query the currently enabled garbage collector and mode.

strategy is a symbol identifying a garbage collector. Permitted values are *stop-and-copy* and *generational* (future version of Elk may support additional garbage collectors). The optional *mode* argument may be specified if the *strategy* argument is equal to *generational*. Currently, only the symbol *incremental* may be used for the *mode* argument to enable incremental garbage collection.

The current version of the interpreter does not support dynamic switching between the stop-and-copy and the generational, incremental garbage collector at runtime. Instead, a garbage collector has to be selected at compile time (by setting the *generational_gc* variable in the installation's site file to either *yes* or *no*). Thus, *garbage-collect-status* can currently only be used to query the garbage collector and, if the generational, incremental garbage collector has been selected, to enable and disable incremental garbage collection (this restriction may be removed in future versions).

garbage-collect-status returns a list of symbols indicating the currently enabled garbage collector and mode. This list resembles the arguments to *garbage-collect-status*, i. e. the first element of the list one of the symbols *stop-and-copy* and *generational*, and an optional, second symbol (*incremental*) may be present if the first symbol is equal to *generational*.

If *garbage-collect-status* is invoked with no arguments, or if the desired garbage collector or mode of operation cannot be enabled (either because selection of a strategy at runtime is not supported, of because the mode of operation cannot be supported), the primitive just returns the currently active strategy and mode.

(collect) **procedure**

Causes a garbage collection. Even if incremental garbage collection has been enabled, *collect* always performs a full garbage collection run.

(collect-incremental) **procedure**

This primitive is only present if the generational garbage collector has been selected. An error is signaled if *collect-incremental* is invoked and incremental garbage collection has not been enabled, i. e. if a call to *garbage-collect-status* would return the list *(generational).*

collect-incremental starts an incremental garbage collection and then returns immediately. If an incremental garbage collection is already in progress, *collect-incremental* triggers one incremental garbage collection step, i. e. scans a few more pages of memory, and then returns immediately. The primitive returns true if the incremental garbage collection has been finished, false otherwise.

If incremental garbage collection is disabled by a call to *(garbage-collect-status 'generational)* while an incremental garbage collection run is in progress, the next call to *collect-incremental* finishes the incremental garbage collection run and returns #t; further calls to *collect-incremental* will signal an error.

garbage-collect-notify? **variable**

This variable is defined in the global environment. If the value of *garbage-collect-notify?* is true, a message indicating the amount of free memory on the heap and the size of the heap are displayed whenever a stop-and-copy garbage collection is performed. If the generational, incremental garbage collector has been enabled, the amount of reclaimed memory is displayed on each garbage collection run, and a message is displayed each time the heap is expanded by the garbage collector. *garbage-collect-notify?* is bound to #t initially.

31. Features

(feature? *symbol***)** **procedure**

Returns #t if *symbol* is a feature, i. e. *provide* has been called to indicate that the feature *symbol* is present; #f otherwise.

(provide *symbol***)** **procedure**

Indicates that the feature *symbol* is present. Returns *void.*

(require *symbol***)** **procedure**
(require *symbol file***)** **procedure**
(require *symbol file environment***)** **procedure**

If the feature *symbol* is not present (i. e. (feature? *symbol*) evaluates to #f), *file* is loaded. A message is displayed prior to loading the file if the value of the global variable *autoload-notify?* is true. If the feature is still not present after the file has been loaded, an error is signaled.

If the *file* argument is omitted, it defaults to *symbol*; if *symbol* does not end in a suffix (i. e. does not contain a dot character), the suffix *.scm* is appended to obtain a file name.

If an *environment* argument is supplied, the file is loaded into given environment. If the *environment* argument is omitted, it defaults to the current environment.

file must be either a string or a symbol or a list of strings or symbols, in which case all elements of the list must be the names of object files (see *load* above).

(features) **procedure**

Returns the currently provided features a list of symbols.

32. Miscellaneous

(dump *file***)** **procedure**

Writes a snapshot of the running interpreter to *file* and returns #f. When *file* is executed, execution of the interpreter resumes such that the call to *dump* returns #t (i.e., *dump* actually returns twice). *dump* closes all ports except the current input and current output port.

This primitive is not supported on platforms that are not capable of creating an executable file from the memory image of the running process. If *dump* is available, the feature *elk:dump* is provided by the interpreter on startup (see ‘‘Features’’ above).

(eval *list***)** **procedure**
(eval *list environment***)** **procedure**

Evaluates the expression *list* in the specified environment. If *environment* is omitted, the expression is evaluated in the current environment.
Examples:

```
(let ((car 1))
  (eval 'car (global-environment)))      ==>  primitive car

(define x 1)
(define env
  (let ((x 2)) (the-environment)))
(eval 'x)        ==>  1
(eval 'x env)    ==>  2
```

(bound? *symbol***)** **procedure**

Returns #t if *symbol* is bound in the current environment, #f otherwise.

(type *obj***)** **procedure**

Returns a symbol indicating the type of *obj*.
Examples:

```
(type 13782343423544)              ==>  integer
(type 1.5e8)                       ==>  real
(type (lambda (x y) (cons x y)))   ==>  compound
(type #\a)                         ==>  character
(type '(a b c))                    ==>  pair
(type '())                         ==>  null
(type (read
  (open-input-string "")))         ==>  end-of-file
```

(void? ***obj*****)** **procedure**

Returns true if *obj* is the non-printing object, false otherwise.

(command-line-args) **procedure**

Returns the command line arguments of the interpreter's invocation, a list of strings.

33. R^4RS Language Features not Implemented by Elk

- Rational and complex numbers are not supported.
- Radix prefixes (#b, #o, #d, and #x) for real numbers are currently not implemented.
- The exponent markers *s*, *f*, *d*, and *l* are not implemented; the character # is not permitted in place of digits in numerical constants.
- *char-ready* is not implemented correctly (see above).
- *transcript-on* and *transcript-off* are not implemented.

Index

Table of Contents