unroff 1.0 Programmer's Manual

Oliver Laumann

unroff is a programmable, extensible troff translator that is useful for converting documents with embedded troff markup into another format. Although *unroff* has been designed with higher-level, structure-oriented target languages (such as SGML) in mind, it fully supports all constructs and idiosyncrasies of ordinary troff, so that even low-level formatting requests can be handled correctly if desired.

Translation rules for a specific output format and knowledge about existing troff macro packages are not hard-wired in *unroff*, instead, the translation is controlled by a user-supplied set of procedures written in the *Scheme* programming language. Interpretation of the procedures is facilitated by a full Scheme interpreted embedded in *unroff*. This manual describes the Scheme primitives provided by *unroff* that can be used to customize the translation rules implemented by existing back-ends and to write new ones for new output formats.

1. Additional Documentation

For a general overview of *unroff* and a description from the user's perspective, please read the manual page *unroff*(1) that accompanies the distribution. In addition, there exists one manual page for each output format for which a back-end is provided, and another one for each combination of output format and troff macro package explaining the translation rules associated with the individual macros. For example, the back-end for the Hypertext Markup Language (HTML) that is part of the distribution and that supports the **–man** and **–ms** macros comes with these manual pages:

```
unroff-html(1)
unroff-html-man(1)
unroff-html-ms(1)
```

This text assumes familiarity with the basic troff and Scheme concepts. For a troff manual, refer to the documentation provided by your UNIX system's vendor. As *unroff* supports a number of troff extensions introduced by the free *groff* formatter (which is part of the GNU project), you may want to read the manual page *troff*(1) that is included in the groff distribution.

unroff is centered around *Elk*, the Scheme-based Extension Language Kit. For a description of the Elk-specific Scheme language features please refer to the documentation included in the Elk distribution (which is freely available). An overview of Elk can be found in: Oliver Laumann and Carsten Bormann, Elk: The Extension Language Kit, *USENIX Computing Systems*, vol. 7, no. 4, pp. 419–449, 1994. The Scheme language is described in several textbooks; and the Revised^4 Report on the Algorithmic Language Scheme, on which the IEEE Standard for Scheme is based, can be downloaded from several major FTP sites.

2. Where to Place Scheme Code?

unroff accepts Scheme code in a number of places. First, several Scheme files are loaded on startup:

```
scm/troff.scm
scm/format/common.scm
scm/format/package.scm
~/.unroff
```

The first three path names are relative to a site-specific library directory where the files have been installed by the system administrator. ‘‘troff.scm’’ contains definitions that are independent of the actual output format and troff macro-package; and the file ‘‘.unroff’’ (loaded from the caller’s home directory) typically contains Scheme code to define user-preferences and to tailor and extend the translation rules implemented by the files loaded from a central location. See the manual page *unroff*(1) for more information.

Additional files with user-supplied Scheme definitions (e. g. translation rules for user-defined macros) can be passed to *unroff* by mentioning them in the command line. In general, troff input files and Scheme source files can be mixed arbitrarily when calling *unroff*. Finally, Scheme code can be embedded directly in the troff documents by means of the new ‘‘.##’’ troff request and the corresponding extension to the ‘‘.ig’’ request as explained in the manual page. Such inline Scheme code is executed on-the-fly when it is encountered by the parser while processing the document.

3. Events and Event Handling

unroff interprets a troff document as a sequence of chunks of normal text and interspersed ‘‘events’’. Plain text is usually just copied to the current output (a file or standard output). The output produced for an event is determined by an ‘‘event handler’’ (usually a Scheme procedure) that can be associated with each event. If no event handler can be found for an event encountered in the currently processed document (with a few exceptions), a warning message is displayed and the input that triggered the event is skipped (in case of requests and macros) or treated like normal text. For events such as troff requests, a separate Scheme procedure can be defined for each request, and the name of the request that triggered the event is then passed to the procedure as an argument. An event handling procedure can be defined for

- each troff request, including requests that perform intrinsic troff functions, such as ‘‘.de’’ and ‘‘.if’’
- each troff macro, whether user-defined or part of a macro package
- each troff string
- each number register
- each special character
- each escape sequence
- each character (to provide character translations)
- each inline equation enclosed by the current *eqn*(1) delimiter characters
- each end of sentence (defined as a period, exclamation mark, or question mark, followed by a newline).

When invoked, every Scheme procedure associated with one of the above events receives one or more arguments. For example, a procedure registered for the escape sequence ‘\h’ (horizontal space) is passed the name of the escape sequence (the letter ‘h’) as well as the argument to ‘\h’ (i. e. the amount of space). Likewise, event handling procedures for requests and macros are called with the name of the request or macro as well as any arguments specified in the troff input. The exact arguments passed to each type of event handler will be explained below.

A Scheme procedure associated with an event must return a string which is then output in place of whatever input triggered the event. Here, and in a number of other places, a Scheme symbol or a Scheme is accepted as an alternative to a string return value. Event handling procedures are free to directly produce output in addition to returning it as a result. As procedures associated with events frequently just return a fixed text, the text itself may be defined as the event handler in place of the procedure to save the overhead of the procedure call.

Predefined Scheme procedures are supplied for events such as the requests ‘‘.de’’, ‘‘.nr’’, ‘‘.ds’’, and the corresponding escape sequences ‘\n’ and ‘*’ to support user-defined macros, strings, and number registers. In any case, specific event handlers registered for macros, strings, and number registers supersede any user-supplied definitions. Thus, the author of a document can attach a special translation rule to a macro, string, or number register defined in the document to take effect when the document is processed by *unroff*.

This is particularly important for high-level, structure-oriented target languages like SGML, as the the micro-formatting used by typical, more complex troff macros and by many low-level requests may not be expressible in such languages. As a case in point, it would obviously be impossible to translate, for example, the ".IP" macro defined by the "ms" package to a language such as HTML just by looking at the definition of the macro. For this reason, *unroff* does not really load the actual macro definitions for a troff macro package selected via the "−m" option; instead, an event handler is defined for each macro exported by the package to generate whatever represents the corresponding macro's function in the target language.

4. Defining Event Handlers

In the following list of Scheme primitives, the argument *name* denotes the name of a troff request, macro, escape sequence etc. (without any initial period or escape character) and can be supplied in form of a Scheme string, a Scheme symbol, or a Scheme character:

```
(defrequest "ti" ...)

(defrequest 'sp  ...)

(defescape  #\h  ...)
```

(the primitives *defrequest* and *defescape* will be introduced in a moment). An argument named *handler* is either a procedure (usually a lambda expression) which returns a string, a symbol, or a character; or *handler* can itself be specified as a string, symbol, or character. In addition, the literal "#f" (false) can be supplied as a *handler* argument to remove any event handler that is currently associated with that event. Each of the "def" primitives listed below returns the handler that was previously associated with the corresponding event, or "#f" if the event was not handled.

(defrequest *name handler***)**

Associates the given handler with the given troff request. If *handler* is a procedure, it is passed the request's name and arguments as strings when called later. Passing the name of the request as the first argument aids in associating the same procedure with several different requests. *unroff* does not limit the number of arguments to requests, thus, an event handling procedure for a requests that takes a variable number of arguments could be defined like this:

```
(defrequest 'rm
  (lambda (rm . args) ...))
```

If the request is invoked with fewer arguments than the procedure has formal arguments, the remaining arguments are bound to the empty string. If the request is invoked with *more* arguments than the procedure has formal arguments, the last lambda variable is assigned a string consisting of the (space-delimited) arguments left over after the other formal arguments have been bound to the other actual arguments. However, if *handler* has only one formal argument, an error message is displayed when the request is called with any arguments at all and the event is skipped. For example, consider the following handler for the (non-existing) request "xx":

```
(defrequest 'xx
  (lambda (name a b) ...))
```

The procedure's arguments *a* and *b* will be bound as follows when the request is invoked:

```
.xx foo             name="xx"  a="foo"  b=""

.xx foo bar baz     name="xx"  a="foo"  b="bar baz"
```

(defmacro *name handler***)**

Associates *handler* with the given troff macro, superseding any definition for this macro established by the ordinary ".de" request. The only difference between *defrequest* and *defmacro* is the way arguments are bound in case *handler* is a procedure (troff employs slightly different rules when parsing the call to a request and a macro invocation). The quote character can be used in the latter case to surround arguments containing spaces, while quote characters are treated as normal characters in requests, which allows for the following remarkable troff idiom:

```
.ds xy "hello
```

In contrast to event handlers defined for requests, the formal arguments of a handler procedure associated with a macro must match the actual arguments in the normal way, that is, as if the procedure were invoked from within Scheme. A warning message is displayed if the number of macro arguments does not match the number of formal procedure arguments, and the event is skipped.

(defspecial *name handler***)**

Associates *handler* with the special character whose name is *name*. The name must have a length of 2. In addition, an empty name can be specified to define a ‘‘fallback’’ handler that is called for special characters for which no handler exists. Like all event handler procedures, *handler* can have arbitrary side-effects in addition to returning a result; for example, the procedure may display a warning message if the special character cannot be represented in the target language and an approximation must be rendered instead.

(defstring *name handler***)**

Associates a handler with the specified troff string. As *unroff* provides a default handler for the request ‘‘.ds’’ to implement used-defined strings, *defstring* is primarily used to give definitions for strings exported by troff macro packages.

(defnumreg *name handler***)**

This request behaves like *defstring*, except that it works on number registers. Note that the Scheme primitive *number−>string* may have to be used by *handler* (if it is a procedure) to convert a numeric result into a string that can be returned from the handler.

In troff input, number registers as well as strings, special characters, and escape sequences can be denoted using the groff ‘‘long name’’ syntax, unless troff compatibility has been enabled:

```
\n[numreg]   \n[string]   \f[font]   \[em]   ...
```

(defescape *name handler***)**

Associates an event handler with an escape sequence. *name* must have a length of 1, unless the empty string is given to define a ‘‘fallback’’ event handler (as with *defspecial*). Handlers defined for certain escape sequences are passed a second argument in addition to the name of the escape sequence. This is true for all escape sequences that have an argument according to the troff specification:

```
\b  \c  \f  \h  \k  \l  \n  \o  \s  \v  \w  \x  \z
\*  \$  \"
```

In addition, handlers for these groff escape sequences are passed an additional argument unless troff compatibility is enabled:

```
\A  \C  \L  \N  \R  \V  \Y  \Z
```

The form of an escape sequence argument is determined by the troff specification and cannot be programmed; for example, the handler for ‘\z’ is passed a character or a special character, and the handler for ‘\"’ is invoked with the rest of the current input line sans the terminating newline. (The latter can be used to translate troff comments.)

Handlers registered for the escape sequences ‘\n’ and ’\s’ are passed an optional third argument, one of the Scheme characters #\+ and #\−, if the escape sequence argument begins with a sign. The sign is then stripped from the actual argument.

As ‘\n’ and ‘*’ are treated as ordinary escape sequences, handlers can be defined for them to achieve some form of fallback for number register and strings. *unroff* provides suitable default handlers for ‘\n’, ‘*’, and ’\$’ as part of the implementation of user-defined number registers, strings, and macros. These handlers can be overridden if desired.

(defchar *name handler***)**

Associates *handler* with a character. *name* must have a length of 1. Each time the specified character is encountered in the troff input, the result (or value) of *handler* is output in place of the character. Character translations are not applied to the result of event handlers; event procedures can use the Scheme primitive *translate* (as described below) to execute the character translations established by calls to *defchar* if desired.

defchar currently has a number of weaknesses. The argument cannot be a special character (that is, *name* must be a plain character), and the mechanism cannot be used to achieve true *output* translations as with the troff request ‘‘.tr’’ or the groff request ‘‘.char’’.

(defsentence *handler***)**

Defines a handler to be consulted on end of sentence. If *handler* is a procedure, it is passed the punctuation mark ending the sentence as its argument (in form of a Scheme character). In any case, if an event handler has been specified, its result (or value) is output in place of the end-of-sentence mark and the newline character following it.

(defequation *handler***)**

Defines a handler for *eqn* inline equations. If *handler* is a procedure, it is passed the contents of the inline equation (with the delimiters stripped) as an argument. When an inline equation is encountered in the troff input and a handler has been defined for inline equations, the handler's result (or value) is output in place of the equation.

For inline equations to be recognized, delimiters must be defined first by passing *eqn* input that includes a ‘‘delim’’ directive to the Scheme primitive *filter-eqn-line* (explained below), as is usually done by the event handler associated with the request ‘‘.EQ’’.

5. Querying Event Handlers

In addition to associating event handlers with events by means of the ‘‘def’’ primitives, several primitives exist to query the currently defined handler for a given event:

(requestdef *name***)**
(macrodef *name***)**
(specialdef *name***)**
(stringdef *name***)**
(numregdef *name***)**
(escapedef *name***)**
(chardef *name***)**
(sentencedef)
(equationdef)

Observe that the name of each primitive is derived from the name of the corresponding ‘‘def’’ primitive by exchanging the word ‘‘def’’ and the rest of the name. Each *name* argument is subject to the constraints described under the corresponding ‘‘def’’ primitive above. Each primitive returns whatever object has been registered as the event handler (procedure, string, symbol, character); or #f if no handler has been defined for the event.

6. Event Procedures with Side-Effects

Besides the basic events described in the preceding sections, another group of—slightly different—events exist and can be handled by user-defined Scheme procedures. These events are not related to troff functions, but to a number of other conditions that are encountered when processing documents:

- the end of an input line

- the beginning of a troff input file processed by *unroff*
- the end of a troff input file
- startup of the program
- termination of the program
- a keyword/value option encountered in the command line.

Among other tasks, these events can be used to generate a prologue and epilogue for each input file. In contrast to the events described in the previous section, handlers for these events are called solely for their side-effects. Each event handler must be a Scheme procedure. Their results are ignored, thus the procedures must have side-effects to be useful. Another difference is that more than one event handler can be associated with each request. A numeric *level* (a small integer number) is specified together with each event handler, and when the corresponding event is triggered, all procedures defined for this event are executed in increasing order as indicated by their levels.

(defevent *event level handler***)**

Associates the procedure *handler* with an event and returns the previous event handler registered for this combination of event and level. *level* is an integer between 0 and 99; *handler* is a procedure, or the literal #f to remove a previously defined handler. *event* indicates the type of event and is one of the following Scheme symbols: *line* (end of input line), *prolog* (beginning of input file), *epilog* (end of input file), *start* (program start), *exit* (program termination), *option* (keyword/value command line option).

Procedures defined for the events *prolog* and *epilog* are called with two string arguments: the path name (as specified by the user) and the file name component of the troff input file whose processing has just begun or finished, or the string ‘‘stdin’’ if *unroff* is taking its input from standard input. Procedures defined for the event *option* are passed the option’s name and value as strings. All other event procedures are invoked without arguments. *unroff* provides a default handler for *option* (see the primitives for options below).

Example:

```
(defevent 'exit 50     ; cleanup on exit
  (lambda ()
    ...))
```

The handler defined in this way will be executed on termination, after any handlers with levels 0–49.

(eventdef *event level***)**

Returns the procedure defined as a handler for *event* and *level*, or #f if no such handler exists. See *defevent* above for a description of the arguments.

7. How Troff Input is Processed

To be able to write non-trivial event handling procedures, it helps to have a look at how troff input is processed, especially since the parser of *unroff* works somewhat differently than ordinary troff. In particular, the parser cannot blindly rescan the result of handlers for escape sequences or special characters, as these handlers will probably generate text in the *target language* that cannot be interpreted as troff input any longer. Here is a brief overview of the parsing process.

Each input line is first scanned for references to troff strings and number registers (this scanning pass will later be referred to as the ‘‘expansion phase’’). For each ‘*’ or ‘\n’ sequence found in the input line, *unroff* checks whether a handler for the string or number register has been defined with *defstring* or *defnumreg*, and if this is the case, replaces the string or number register reference by the result (or value) of the handler. Otherwise, if a handler for the escape sequence ‘*’ or ‘\n’ proper has been defined, that handler is called. Otherwise the reference is left untouched and scanning resumes behind it[1]. Comments are recognized in this phase, too, by calling the handler for the ‘\"’ escape sequence if there is one.

[1] Although the result of specific event handlers defined for strings is not rescanned, the handler for ‘*’ that is supplied by *unroff* to implement user-defined strings does rescan the contents of a string when it is expanded.

Next, the parser checks whether the result of the first phase is a request or macro invocation (that is, begins with a period or an apostrophe). If this is the case, the arguments are parsed mimicking the behavior of ordinary troff. The rules for macro arguments are employed if a handler has been defined for the token after the period with *defmacro*, else the rules for requests are used. The handler for the macro or request is then used, or applied to the arguments if it is a procedure.

If the input line does not contain a request or macro invocation, it is scanned a second time to take care of escape sequences and special characters (for lack of a better term, we will call this phase ‘‘escape parsing’’). Every escape character reference, special character, and inline equation is replaced by the result (or value) of the event handler registered for it, or left in place if there is no handler. Character translations defined by means of *defchar* are also executed in this phase.

Finally, the result of the escape parsing phase or of the request or macro invocation is checked whether it constitutes the end of a sentence, and if so, the handler for this event is called (actually, in the former case, the check is applied before *and* after the escape parsing and must succeed both times). As the final step the line is output, and any handlers for the *line* event are invoked.

An important thing to note is that the arguments passed to a handler defined for a request or macro are not scanned for escape sequences and special characters. Therefore event procedures must explicitly parse their arguments if desired by calling the Scheme primitive *parse* (which will be described in the next section). Consider, for example, an event procedure associated with a macro ‘‘IP’’:

```
(defmacro 'IP
  (lambda (IP tag . indent)
    ...))
```

and a call to the macro with an argument containing a special character:

```
.IP \(bu
```

As the argument to the event procedure is only scanned for strings and number registers, the variable *tag* will be bound to the string ‘‘\(bu’’. Applying *parse* to the argument will turn it into whatever is the target language representation for the special character ‘‘\(bu’’ (that is, the result of the event handler for the special character). Whether or not arguments will have to be parsed depends on the particular request or macro; the procedure implementing the request ‘‘.tm’’, for instance, will print its ‘‘raw’’ argument (a sample event handler for the request ‘‘.tm’’ is supplied by *unroff*).

8. Calling the Parser

The following Scheme primitives are used by event procedures for requests, macros, and escape characters to parse their arguments or to parse lines of text that have been read from an input source. Each of the primitives can be invoked with zero or more arguments of type string, symbol, or character. The arguments are concatenated to form a Scheme string which is then passed to the parser, and the result is returned as a new string.

(parse . *args***)**

This primitive feeds its arguments to the ‘‘escape parsing’’ pass as described in the previous section. It scans its arguments for special characters and escape sequences and replaces them by the corresponding event values (or results), and it executes character translations.

(translate . *args***)**

Like *parse* above, except that only output character translations (defined by calls to *defchar*) are executed.

(parse-expand . *args***)**

This primitive applies the ‘‘expansion parsing’’ phase (as described in the previous section) to its arguments. Compared to *parse*, *parse-expand* is only used rarely, as input lines read in the normal way are scanned for string and number register references anyway. The sample implementation supplied by *unroff* for the requests ‘‘.ds’’, ‘‘.as’’, and '*' makes use of this primitive to rescan the contents of user-defined strings upon interpolation.

(parse-line . *args***)**

This primitive parses an entire input line, which may contain a call to a request or macro, as described in the previous section. The line made up by the primitive's arguments is treated exactly as it if were read from an input file, although it need not have a terminating newline. Two places where this primitive is required are the handler for the request ‘‘.so’’ and the code that expands user-defined macros.

(parse-copy-mode . *args***)**

The primitive *parse-copy-mode* parses its arguments in a manner similar to troff ‘‘copy mode’’. In this mode, escape sequences beginning with '\$' are dealt with (by calling their event procedures), the sequence ‘\\’ is replaced by a single ‘\’, and each occurrence of ‘\.’ is replaced by a period. Macro bodies are parsed in copy mode during macro definition and again when the macros are expanded.

The sample implementation of user-defined macros supplied by *unroff* defines suitable event handlers for the usual

```
\$1  \$2 ...
```

escape sequences (there is no limit to the number of arguments, and the groff long name convention may be used to denote an argument number), and in addition for the groff extensions

```
\$0   \$*   \$@
```

as explained in the manual page *unroff*(1).

(parse-expression *expr fail scale***)**
(parse-expression-rest *expr fail scale***)**

These primitives evaluate the numeric expression specified by the string argument *expr* and return the result as an exact number. The usual troff expression syntax, operators, and scale indicators are supported. If an error occurs during evaluation (for instance, if *expr* is not a syntactically valid expression), a warning message is displayed and *fail* (which may be an arbitrary Scheme object) is returned. The character argument *scale* is the default scale indicator, for example ‘#\m’, or ‘#\u’ for basic units.

The primitive *parse-expression-rest* is identical to *parse-expression*, except that its return value is a cons cell whose car consists of the result of the evaluation and whose cdr is the rest of *expr* starting at the character position where parsing of the expression stopped. In other words, the primitive evaluates the portion of *expr* that constitutes a valid expression, and it returns the result and whatever is left over. Warning messages are also suppressed, except if an overflow occurs during evaluation. *parse-expression-rest* is useful for tasks like parsing the argument of the escape sequences ‘\l’ and ‘\L’ where an expression is immediately followed by another character. Examples:

```
(parse-expression "(2+8)/5" 0 #\u)      ☞  2
(parse-expression "foo" #f #\u)         ☞  #f; prints warning

(parse-expression-rest "1+1" #f #\u)         ☞  (2 . "")
(parse-expression-rest "(2+8)/5foo" 0 #\u)   ☞  (2 . "foo")
(parse-expression-rest "15\&-" 0 #\u)        ☞  (15 . "\&-")
```

(char-expression-delimiter? *char***)**

Returns #t if the character argument *char* is valid as the first character of a numeric expression (e. g. a digit), otherwise #f.

(set-scaling! *scale factor divisor***)**
(get-scaling *scale***)**

These primitives set and read the scale factor and divisor for the specified scale indicator. *scale* is the scale indicator (a character); *factor* and *divisor* are integers. *get-scaling* returns the scaling for the specified scale indicator as a pair of integers. The factors and divisors are initially set to 1 for all scale indicators; they must be assigned useful values by each back-end.

9. Streams

Input, output, and storage of text lines in *unroff* are centered around a new Scheme data type named *stream* and a set of primitives that work on streams. A stream can act as a source (input stream) or as a sink (output stream) for lines of text. Streams not only serve as the basis for input and output operations and for the exchange of text with shell commands, but can also be used to temporarily buffer lines of text (e.g. footnotes or tables of contents) and to implement user-defined macros in a simple way. Each input or output stream can be connected to one of the following three types of *targets*:

- a file, or the program's standard input or standard output
- a UNIX pipe connected to a shell running a shell command
- an internal *buffer* whose lifetime is limited to that of the current invocation of *unroff*.

Buffers act similar to (initially empty) files, except that they are not visible from the outside and that they are destroyed automatically on exit of the program. Once a buffer has been filled with text through an output stream, it can be reopened and read through an input stream multiple times. However, if a buffer is currently written through an output stream, no more streams may refer to the same buffer. As the contents of buffers kept in memory, input and output operations on buffers are fast. The sample implementation of user-defined macros utilizes buffers to store the macro bodies; a macro can then be expanded simply by redirecting the current input source to the corresponding buffer temporarily.

Both the parser and all input and output primitives operate on a *current input stream* and a *current output stream*; input and output is always performed using these two streams. On startup, *unroff* initializes the current output stream to either point to standard output or to a newly created output file (usually depending on the value of the **document** option). If the current output stream is assigned the literal #f, output is sent to standard output[2]. Likewise, for each input file mentioned in the command line, a stream pointing to that file is created and assigned to the current input stream before the parser starts processing the file. The rest of this section lists the Scheme primitives operating on streams.

(stream? *obj***)**

The type predicate for the new data type. It returns #t if *obj* is a member of the type *stream*, otherwise #f.

(input-stream)
(output-stream)

Returns the current input stream, or output stream respectively.

(open-input-stream *target***)**
(open-output-stream *target***)**
(append-output-stream *target***)**

These primitives create a new input stream or output stream pointing to the specified target. The argument *target* is a string or a symbol. If the target is enclosed in square brackets, it names a buffer; if it begins with the pipe symbol '|', a pipe to a shell running the rest of the target as a shell command is established; otherwise *target* is interpreted as a file name. *append-output-stream* rewinds to the end of the specified output buffer or file before the first output operation; it acts like *open-output-stream* in case of a pipe. Examples:

---

[2] While #f indicates "standard output" when assigned to the current output stream, it is an error to call an input primitive after #f has been assigned to the current *input* stream. This may be considered a mis-feature; the current input and output streams should be treated similarly with respect to standard input and standard output.

```
(let* ((buffer (open-output-stream '[temp]))
       (pipe   (open-input-stream "|ls -l /usr/lib/tmac"))
       (file   (open-input-stream "/etc/passwd")))
  ...)
```

(set-input-stream! *stream***)**
(set-output-stream! *stream***)**

These primitives make the specified stream the *current* input stream (or output stream respectively). *stream* must be the result of a call to one of the three primitives that open a stream, or #f. An error is signaled if *set-input-stream!* is applied to an output stream or vice versa, or if the stream has been closed in the meantime.

(close-stream *stream***)**

Closes the specified stream. An error is signaled if the stream is still the current input stream or current output stream. Once an output stream pointing to a buffer has been closed, the buffer can be reopened for reading. A stream that is no longer reachable is closed automatically during the next run of the garbage collector.

(stream-buffer? *stream***)**
(stream-file? *stream***)**
(stream-pipe? *stream***)**

These predicates return #t if the specified stream points to a buffer, a file, or a pipe respectively, otherwise #f.

(stream-target *stream***)**

This primitive returns the target to which the specified stream points. The return value is a string. In case of a pipe, the target is truncated at the first space, that is, only the command name is included. The target of the current input stream (together with the current line number) is displayed as a prefix of error messages and can also be obtained through the primitive *substitute* described below.

(stream-position *stream***)**

Returns the current character position of the specified output stream, that is, the offset at which the next character will be written. The return value for input streams is currently always zero. This primitive is useful in conjunction with *file-insertions* (described below).

(stream–string *target***)**

This primitive opens an input string to the specified target, reads from the stream until end-of-stream is reached, closes the stream, and returns the concatenation of all the lines that have been read as a string[3].

10. Input and Output Primitives

unroff provides one new input primitive and one new output primitive that work with the current input stream and current output stream (and a third primitive which is just an optimization of the latter, as well as a few auxiliary functions).

(emit . *args***)**

emit is the only stream-based output primitive. It receives any number of strings, symbols, and characters, concatenates its arguments, and sends the resulting string to the current output stream (to standard output if the the current output stream has been assigned #f). *emit* is primarily used in situations where text has to be output without rescanning it and without applying any character translations. It is also used from

[3] *stream–>string* is a misnomer, because the argument of the primitive is not a stream, nor does the primitive actually *convert* a stream to a string as suggested by the '–>' sign.

within the event procedures that are called for their side-effects, for example, by the *prolog* and *epilog* event procedures to generate a header and trailer for each output file. The primitive returns the empty symbol so that it can be called as the last form in an event procedure whose result is used.

Example: the new troff request for transparent output, as explained in the manual page *unroff*(1), can be implement like this:

```
(defrequest '>>
  (lambda (>> code)
    (emit code #\newline)))
```

(read-line)

This primitive reads the next input line from the current input stream and returns it as a string. An error is signaled if the current input stream has been bound to #f, which is the case, for example, when *unroff* has been called with the option **–t** to start an interactive top level. If an incomplete last line (i. e. a line without a terminating newline) is returned by the target pointed to by the current input stream, a newline is appended. Thus, *read-line* always returns at least a string containing a newline character.

(read-line-expand)

This primitive is nothing more than an optimization for

```
(parse-expand (read-line))
```

which has been provided to speed up frequently used functions like macro expansion.

(unread-line *string***)**

This primitive pushes back an input line to the current input stream, which will then be returned by the next call to *read-line* or *read-line-expand*, or it will be read by the parser in the normal way when processing the current input file. *string* need not have a terminating newline. Strings pushed back by multiple calls to *unread-line* are coalesced and returned as a whole by the next input operation.

(error-port)

Returns a Scheme output port that is bound to the program's standard error output. This primitive is used by the default Scheme error handler provided by *unroff* and by the *warn* utility function[4]. Note that *error-port* returns an ordinary Scheme port, not a stream.

11. String Functions

Most of the string handling primitives described in this section could as well have been implemented in Scheme based on the standard Scheme string primitives. They are provided as built-in primitives by *unroff* mainly as optimizations or because writing them as Scheme procedures would have been significantly more cumbersome. All the string functions return new strings, that is, they do not modify their arguments.

(concat . *args***)**

concat can be called with any number of Scheme strings, symbols, and characters. The primitive concatenates its arguments and returns the result as a string.

(spread)

This primitive is identical to *concat*, except that it delimits its arguments by a space character. For example, the event procedure for a macro that just returns a line consisting of its arguments could be define like this:

---

[4] The primitive *error-port* should actually be provided by Elk proper to avoid having to reinvent it for each extensible application.

```
(defmacro 'X
  (lambda (X . words)
    (parse (apply spread words) #\newline)))
```

(repeat-string *num string***)**

Returns a string consisting of the string argument *string* repeated *num* times.

(string-prune-left *string prefix fail***)**

This primitive checks whether *string* starts with the given string prefix, and if so, returns the rest of *string* beginning at the first character position after the initial prefix. If the strings do not match, *fail* is returned (which may an arbitrary object). Example:

```
(string-prune-left "+foo" "+" #f)      ☞  "foo"
(string-prune-left "gulp" "+" #f)      ☞  #f
```

(string-prune-right *string suffix fail***)**

This primitive is identical to *string-prune-left*, except that it checks for a suffix rather than a prefix, that is, whether *string* ends with *suffix*.

(string-compose *string1 string2***)**

If the argument *string2* begins with a plus sign, *string-compose* returns the concatenation of *string1* and *string2* with the initial plus sign stripped. If *string2* begins with a minus sign, it returns a string consisting of *string1* with all characters occurring in *string2* removed. Otherwise, *string-compose* just returns *string2*. This primitive is used for the implementation of the option type *dynstring*.

(parse-pair *string***)**

If *string* consists of two parts separated and enclosed by an arbitrary delimiter character, *parse-pair* returns a cons cell holding the two substrings. Otherwise, it returns #f. Example:

```
(parse-pair "'foo'bar'")     ☞  ("foo" . "bar")
(parse-pair "hello")         ☞  #f
```

(parse-triple *string***)**

This primitive is identical to *parse-pair*, except that it breaks up a three-part string rather than a two-part string and returns an improper list whose car, cadr, and cddr consist of the three substrings[5]. *parse-pair* and *parse-triple* are useful mainly for parsing the arguments to troff requests such as ‘‘.if’’ and ‘‘.tl’’.

(substitute *string . args***)**

This primitive returns a copy of *string* in which each sequence of a percent sign, a *substitution specifier*, and another percent sign is replaced by another string according to the specifier. Two adjacent percent signs are replaced by a single percent sign. The following list describes all substitution specifiers together with their respective replacements.

macros
The name of the troff macro package whose macros are recognized, that is, the argument to the option **–m** (or the empty string if none was specified).

format
The output format, that is, the argument to the option **–f** (or the default output format if the option was omitted).

directory
The name of the library directory from which *unroff* loads its Scheme files.

[5] The primitive *parse-triple* should probably return a proper list rather than an improper list.

progname
The name of the running program (this is used as a prefix in error messages and warning messages).

filepos
A space character followed by the target of the current input stream, a colon, the number of the last input line read from the stream, and another colon. If the current input stream is bound to #f, the empty string is substituted. This specifier is useful for displaying error messages or warning messages.

tmpname
A file name that can be used for a temporary file. Each use of this specifier creates a new, unique file name.

version
The program's major and minor version numbers separated by a period.

weekday
The abbreviated weekday name.

weekday+
The full weekday name.

weekdaynum
The weekday (0–6, Sunday is 0).

monthname
The abbreviated month name.

monthname+
The full monthname.

day The day of the month (01−31).

month
The month (01−12).

year The year.

date The date (in the local environment's representation).

time The time (in the local environment's representation).

a positive number *n*
The *n*th additional argument in the call to the *substitute* primitive, which must be a string.

a *string*
string is interpreted as the name of an environment variable, and the value of this variable is substituted (or the empty string if the environment variable is undefined).

Examples:

```
(substitute "%date% %HOME%")      ☞  "04/09/95 /home/kbs/net"

(substitute "%progname%:%filepos% %1%" "hello")
                     ☞ "unroff: manual.ms:21: hello"

(load (substitute "%directory%/scm/%format%/m%macros%.scm"))
```

12. Tables

unroff provides simple hash tables as a new first class data type *table*. Each table entry associates an arbitrary Scheme object with a key (a Scheme string or symbol). Tables are useful for various purposes; for example, the Scheme code delivered with *unroff* maintains hash tables to store information about number registers, options, fonts, and for other bookkeeping tasks.

(table? *obj***)**

The type predicate for the new type; it returns #t if *obj* is a member of the type *table*, otherwise #f.

(make-table *size***)**

Returns a new table of the specified size. *size* is a positive integer. The smaller the size, the more collisions occur as entries are added to the table. However, the hash function employed by the table primitives ensures that no collisions occur in tables of size 256ˆ*n* if all keys have a length less than or equal to *n*.

(table-store! *table key obj***)**

This primitive stores the Scheme object *obj* under the given *key* in the given *table*. The key argument must be a string or a symbol.

(table-lookup *table key***)**

This primitive checks whether an object is stored in the given *table* under the specified *key*, and if so, returns the object. If no object is stored under *key*, *table-lookup* returns #f.

(table-remove! *table key***)**

Removes the entry selected by *key* from the specified table.

13. Miscellaneous Primitives

The first two primitives described in this section are not essential, as the same function could be achieved with pipe streams, although with greater overhead. The remaining primitives perform a number of troff-specific operations and are only useful in a few specialized contexts.

(shell-command *command***)**

Runs the specified *command* (which must be a string) as a shell command by passing it to a call to *system*(3). The return value is that of *system()* (an integer).

(remove-file *filename***)**

Removes the specified file; *filename* must be a string or a symbol.

(troff-compatible?)

This predicate returns #t if troff compatibility mode has been enabled (i. e. if the option **–C** has been given), otherwise #f.

(set-escape! *char***)**

Sets the troff escape character (initially '\') to the specified character argument. This primitive is used to implement the ".ec" request.

(filter-eqn-line *string***)**

This primitive scans the string argument (which is supposed to be passed to the *eqn* preprocessor afterwards) for occurrences of the "delim" directive. If a "delim" directive is found, the current inline equation delimiters maintained by the parser are changed or disabled as specified by the directive. The primitive returns #f if *string* is empty or consists just of white space, or if it contains a valid "delim" or "define" directive, otherwise #t. The inline equation delimiters are disabled initially.

The primitive is supposed to be used by implementations of the request ".EQ" and inline equation event handlers to intercept the *eqn* input. In this case, the *eqn* preprocessor need only be invoked if *filter-eqn-line* returned #t at least once.

(skip-group)

This primitive reads input lines from the current input stream and scans them for the escape sequences '\{' and '\}' until the nesting level of conditional input is balanced (i. e. until a matching closing brace for an initial opening brace has been found). The primitive is only useful for the implementation of the troff requests for conditional input.

14. File Insertions

The primitive *file-insertions* is a general-purpose utility for inserting strings into files at specified locations in a fast and robust way. One application is to resolve forward references of any kind among a group of files when all files have been processed. In this case, the insertions would be executed by an *exit* event handler.

(file-insertions *insertions***)**

insertions is a list specifying the parameters for the file insertions. Each element of the list is itself a list consisting of a file name (a string), a file offset (an integer between zero and the size of the file), and a string to be inserted in the given file at the given offset. *file-insertions* sorts the list to ensure that each file is only processed once and that the offsets for each file are in increasing order. Then each file is copied to a temporary file

filename`.new`

(where *filename* is the original file name), and the specified insertions are carried out as the file is copied. When processing of a file is finished, the temporary file is renamed to its original name. If there exist links to a file, a warning is displayed and the insertion is skipped.

15. Utilities for Back-Ends

Writers of new back-ends (either for new output formats or for new troff macro packages) can benefit from a number of Scheme procedures and macros that are exported by the file "scm/troff.scm" which is loaded from the library directory on startup. The first two, *eval-if-mode* and *set-option!* are exceptions in that they are typically used by the user's initialization file "~/.unroff" to customize *unroff*, rather than by programmers of *unroff*.

(set-option! *name value***)**

This procedure assigns *value* to the option *name*. The value must be appropriate for the option's type.

(eval-if-mode *mode . forms***)**

This macro is typically used to evaluate a sequence of expressions, *forms*, depending on the output format and macro package specified in the command line. *mode* is a list of two symbols, an output format and a macro package name; the wildcard '*' can be used for both elements. The *forms* are evaluated if the first symbol matches the value of the option **–f** and the second symbol matches the value of the option **–m**; in this case the result of the last sub-expression is returned. Otherwise the forms are ignored and #f is returned. Example:

```
(eval-if-mode (* html)
  (set-option! 'mail-address "net@cs.tu-berlin.de"))
```

(quit *message . args***)**
(warn *message . args***)**

These procedures print *message* and the optional *args* on the port returned by *error-port* using the primitive *format*. The message is prefixed by the program name, current input file name and line number, and, in case of *warn*, the word "warning". A newline is appended. *quit* causes the program to exit with an exit code of 1, and *warn* returns the empty string (and can therefore be used as the last form in event procedures).

(option *name***)**

Returns the value of the specified option.

(define-option *name type initial***)**

Defines a new option with the specified name, type, and initial value. *name* and *type* are strings or symbols. There exist a number of predefined, basic option types as described in the manual page *unroff*(1). The initial value need not match the option's type; for example, the following expression is valid:

```
(define-option 'author 'string #f)
```

(define-option-type *name pre-check pre-msg converter post-check post-msg***)**

This procedure defines a new option type named *name* which can then be used in calls to *define-option*. If an option of this type is specified in the command line, the procedure *pre-check* is applied to the option's value (a string). In this case, if *pre-check* returns #f, *quit* is called with an error message including the string *pre-msg*, which should describe the expected option value format (e. g. ‘‘a character’’). If the check succeeds, the procedure *converter* is called with the option's current value and with the string as given in the command line. The job of the converter procedure is to convert the option value from a string representation to a Scheme object matching the option's actual Scheme type.

Finally, the predicate *post-check* is applied either to the result of *converter* or, if the option was set through a call to *set-option!*, to this procedure's argument. If the predicate returns #f, a error is signaled with an error message including *post-msg* as described in the previous paragraph. For example, the predefined option type ‘‘boolean’’ is defined as follows:

```
(define-option-type 'boolean
  (lambda (x) (member x '("0" "1")))     "0 or 1"
  (lambda (old new) (string=? new "1"))
  boolean? "a boolean")
```

(with-input-from-stream *target . forms***)**
(with-output-to-stream *target . forms***)**
(with-output-appended-to-stream *target . forms***)**

These macros open an input stream (first macro) or output stream to the specified target and assign it to the current input stream (first macro) or current output stream. Then the specified *forms* are evaluated, the stream is reassigned its previous value, and the result of the last sub-expression in *forms* is returned. The macros recur on the primitives *open-input-stream*, *open-output-stream*, and *append-output-stream*, respectively.

(skip-lines *stop***)**

Reads input lines using *read-line-expand* until either end-of-stream is reached (in this case a warning is displayed) or a line matching the string argument *stop* is encountered.