A Reader Framework for Guile

for Guile-Reader 0.6.2

Ludovic Courtès

Edition 0.6.2
8 March 2017

Table of Contents

A Reader Framework for Guile

This document describes Guile-Reader version 0.6.2, for GNU Guile's 2.0 stable series, as well as the forthcoming 2.2 series and the legacy 1.8 series. It was last updated in March 2017.

This documentation should be mostly complete. Details about the C API are omitted from this manual. They can be found in the public header files.

1 Introduction

Guile currently provides limited extensibility of its reader, by means of `read-hash-extend` (see section "Reader Extensions" in *Guile Reference Manual*), for instance, or `read-enable` (see section "Scheme Read" in *Guile Reference Manual*). SRFI-10 tries to propose a generic, portable, extension mechanism similar to `read-hash-extend` but limited to `#,` sequences. Moreover, while this may not always be desirable, all these extension facilities have a global effect, changing the behavior of the sole reader implementation available at run-time. This makes it impossible to have, for instance, one module consider names starting with `:` as symbols, while another considers them as keywords.

Extensions such as the read syntax for SRFI-4 numeric vectors (see section "Uniform Numeric Vectors" in *Guile Reference Manual*) had to be added to Guile's built-in C reader. Syntactic extensions that did not appeal the majority of users, like Emacs-Lisp vectors, are `#ifdef`'d within the reader code and are not available by default. Moreover, some extensions are incompatible with each other, such as the DSSSL keyword syntax and SCSH block comments (see section "Block Comments" in *Guile Reference Manual*). In short the current reader syntax is hardly extensible.

The idea of Guile-Reader is to provide a framework allowing users to quickly define readers for whatever syntax (or rather: variant of the Scheme syntax) they like. Programs can then provide their own readers and, thanks to Guile's `current-reader` mechanism, have their code read with this reader. When using Guile 2.0, readers produced by Guile-Reader are Unicode-capable; they can read from any ASCII-compatible encoding, such as UTF-8 or ISO-8859-1.

While it is much simpler than a full-blown lexer generator such as Flex, Danny Dubé's SILex and Bigloo's RGC, its simple programming interface should make it very straightforward to implement readers, especially for Scheme-like syntaxes. Best of all, Guile Reader comes with a library of components that can typically be used to construct a reader for the Scheme syntax. And each one of this components may be reused at will when creating other readers. On the other hand, one should be aware that this simpler API comes at the cost of a lack of consistency in some cases, as outlined later in this manual (see Section 4.1.5 [Limitations], page 16).

Common Lisp has a similar mechanism to extend its reader which is called the *read table*. Gambit Scheme, for instance, also provides an implementation of read tables. However, it appears to have limitations similar to Guile's `read-enable` and `read-hash-extend` in terms of possibilities for syntax extension. On the other hand, it allows the reader and writer to be kept consistent, which Guile-Reader does not address.

2 Overview

Guile-Reader allows for the construction of readers capable of understanding various syntactic variants. The simplest way to use it is through its *reader library* that allows one to pick and choose various commonly used syntactic extensions to the standard Scheme syntax (see Section 4.2.2 [Reader Library], page 19). However, Guile-Reader also provides a finer-grain programming interface allowing the construction of virtually any reader, with its own syntactic specificities. The following sections focus primarily on this capability.

Before going into the details of the reader framework API, let us have a quick overview of what this is. Basically, Guile-Reader introduces two objects: *readers* and *token readers*. Readers can be thought of, simply, as procedures like Scheme's `read` (see section "Input" in *Revised^5 Report on the Algorithmic Language Scheme*), i.e., procedures that take one (optional) argument, namely the port to read from. We will see later that readers as defined by Guile-Reader can actually receive two more arguments (see Section 4.2.1 [Defining a New Reader], page 18). A reader, like `read`, reads a sequence of characters (the *external representation* of some object) and returns a Scheme object.

Token readers (TRs, for short) are the building block of a reader. A token reader is basically an association between a character or set of characters and a procedure to read and interpret a sequence of characters starting with one of the former. For instance, in a standard Scheme reader, the character `(` may be associated to a procedure that reads an S-expression. Likewise, lower-case and upper-case letters associated with the appropriate procedure form a token reader for symbols.

In Guile-Reader, TRs may be written either in Scheme or in C, and they can even be a reader produced by Guile-Reader itself. Unless it is a reader, the procedure (or C function) used to create a TR will receive four arguments:

- the character that was read and which triggered its call; in the S-exp example, this would be `(`;
- the port to read from;
- the reader which performed this invocation;
- the top-level reader which yielded this invocation.

The next section shows how to get started with Guile-Reader, using a high-level API. Details about of the programming interface are given in the API reference (see Chapter 4 [API Reference], page 9).

3 Quick Start

The simplest way to get started and to produce customized readers is through the high-level API provided by the `(system reader library)` module. As the name suggests, this module provides a library of readily usable readers. The `make-alternate-guile-reader` procedure gives access to these readers. It can be passed a list of symbols describing reader options, such as whether you want support for DSSSL keywords, SRFI-62 comments, SRFI-30 block comments, etc.

The following example binds to *my-reader* a Scheme reader that supports DSSSL-style keywords and SRFI-62 comments, and that is case insensitive:

```
(use-modules (system reader library))

(define my-reader
  (make-alternate-guile-reader '(dsssl-keywords
                                 srfi62-sexp-comments
                                 case-insensitive)))
```

This reader can then be used like the regular `read` procedure:

```
(procedure? my-reader)
=> #t

(with-input-from-string "some-symbol" my-reader)
=> some-symbol

(my-reader (open-input-string "MiXeD-CaSe"))
=> mixed-case

(my-reader (open-input-string "(an sexp with a #;srfi-62 comment)"))
=> (an sexp with a comment)

(my-reader (open-input-string "#!some-DSSSL-keyword"))
=> #:some-dsssl-keyword
```

Most of the time, you will want to use it as the current reader, at least in the module where you created it. Fortunately, Guile provides a mechanism for this, the `current-reader` fluid (see section "Loading" in *Guile Reference Manual*). Changing the value of this fluid from a file that is being loaded will affect the reader use to load it. In general, you will want to modify `current-reader` not only at run time, but also at compile time and when the code is evaluated, which can be achieved using `eval-when` (see section "Eval When" in *GNU Guile Reference Manual*). For instance:

```
;;; This is the beginning of my Scheme file.  At this point, Guile's
;;; default reader is used (or, alternatively, the one that was
;;; passed as a second parameter to `load').  So, for instance, DSSSL
;;; keywords are _not_ recognized.

;; Let's create our own customized reader...
(use-modules (system reader library))
```

```
;; 'eval-when' here is needed to make 'my-reader' accessible at
;; compile time, and to have the 'current-reader' change take
;; effect at compile time.
(eval-when (compile load eval)
  (define my-reader
    (make-alternate-guile-reader '(dsssl-keywords
                                   srfi62-sexp-comments
                                   case-insensitive)))

  ;; Let's make it the current reader.
  (fluid-set! current-reader my-reader))

;; From now on, MY-READER is used to read the rest of this file.  Thus
;; we can happily use the syntactic extensions it implements: DSSSL
;; keywords, SRFI-62 comments and case-insensitivity.

(if (not (keyword? #!dsssl-keyword))
    (error "Something went wrong, this should not happen!"))
```

The nice thing is that `current-reader` is reset to its initial value when the dynamic extent of `load` is left. In other words, the loader of the file above is *not* affected by the `fluid-set!` statement. Reader changes are *hygienic* and modules can use their own without risking to harm each other.

The full list of options supported by `make-alternate-guile-reader` is shown in Section 4.2.2 [Reader Library], page 19. However, this option set is quite limited and you may find yourself wanting a syntactic extension not available here. In that case, you will want to build a new reader, possibly reusing existing reader components known as *token readers*, as described in Section 4.2.1 [Defining a New Reader], page 18.

4 API Reference

All the Scheme procedures described below are exported by the `(system reader)` module. In order to be able to use them, you will need to import this module first:

```
(use-modules (system reader))
```

A C variant is also available for most of them by including the declarations available in the `<guile-reader/reader.h>` header file.

4.1 Token Readers

Basically, token readers are the association of a character or set of characters and a function that is able to interpret character sequences that start by one of these characters. We will see below how to define new token readers first, and then how to re-use existing ones.

4.1.1 Defining a New Token Reader

A new token reader object can be created by calling the `make-token-reader` procedure with a *character specification* and a procedure. A character specification defines the set of characters which should trigger an invocation of the corresponding procedure. The character specification may be either:

- a single character;
- a pair of characters, which is interpreted as a character range;
- a list of characters, which is interpreted as a set of characters.

The procedure passed to `make-token-reader` may actually be either a C function or Scheme procedure that takes four arguments (see Section 4.1.2 [TR Calling Convention], page 9), any "object" returned by `token-reader-procedure`, or a reader. This last option turns out to be quite helpful. For example, this is very convenient when implementing the various Scheme read syntaxes prefixed by the `#` character: one can create a reader for `#`, and then turn it into a token reader that is part of the top-level reader.

The reference for `make-token-reader` is given below:

`make-token-reader` *spec proc* [*escape?*] [Scheme Procedure]
`scm_make_token_reader` (*SCM spec, SCM proc, SCM escape_p*) [C Function]
Use procedure (or reader) *proc* as a token reader for the characters defined by *spec*. If *escape_p* is true, then the reader this token reader belongs to should return even if its result is undefined.

The next section explains the token reader calling convention, i.e., how the *proc* argument to `make-token-reader` is invoked.

4.1.2 Token Reader Calling Convention

A token reader's procedure is passed four arguments:

- the character that was read and which triggered its call; in the S-exp example, this would be `(`;
- the port to read from;

- the reader which performed this invocation, i.e., either an `scm_reader_t` object (if the token reader is written in C) or a four-argument Scheme procedure (if the token reader is written in Scheme);
- the top-level reader which yielded this invocation and which may be different from the previous argument in the case a token reader was made from a reader; the use of these two arguments will be detailed in the next section, Section 4.1.3 [Invoking a Reader from a TR], page 10.

It must return a Scheme object resulting from the interpretation of the characters read. It may as well raise an error if the input sequence is corrupt. Finally, it may return `*unspecified*`, in which case the calling reader will not return and instead continue reading. This is particularly useful to define comment token readers: a TR that has just read a comment will obviously not have any sensible Scheme object to return, and a reader is not expected to return anything but a "real" Scheme object. A token reader for Scheme's `;` line comments may be defined as follows:

```
(make-token-reader #\; read-a-line-and-return-unspecified)
```

This behavior may, however, be overridden by passing `make-token-reader` a third argument (called *escape?*):

```
(make-token-reader #\; read-a-line-and-return-unspecified #t)
```

A reader that includes this TR will return `*unspecified*` once a line comment has been read. This is particularly useful, for instance, when implementing `#!` block comments (see section "Block Comments" in *Guile Reference Manual*, for more information) as a TR attached to `#\!` within the `#\#` sub-reader (see Section 4.1.1 [Defining a New Token Reader], page 9).

Finally, the procedure passed to `make-token-reader` may be `#f`, in which case the resulting TR will just have the effect of ignoring the characters it is associated to. For instance, handling white spaces may be done by defining a TR like this:

```
(make-token-reader '(#\space #\newline #\tab) #f)
```

4.1.3 Invoking a Reader from a Token Reader

As seen in section See Section 4.1.1 [Defining a New Token Reader], page 9, token readers are systematically passed to readers when invoked. The reason why this may be useful may not be obvious at first sight.

Consider an S-exp token reader. The TR itself doesn't have sufficient knowledge to read the objects that comprise an S-exp. So it needs to be able to call the reader that is being used to actually read those objects.

The need for the *top-level-reader* argument passed to token readers may be illustrated looking at the implementation of the vector read syntax (see section "Vector Syntax" in *Guile Reference Manual*). One may implement the vector reader as a token reader of the `#` sub-reader (see Section 4.1.1 [Defining a New Token Reader], page 9). The vector token reader may be implemented like this:

```
(lambda (chr port reader top-level-reader)
  ;; At this point, '#' as already been read and CHR is '(',
  ;; so we can directly call the regular S-expression reader
  ;; and convert its result into a vector.
  (let ((sexp-read (token-reader-procedure
```

```
                    (standard-token-reader 'sexp))))
      (apply vector
             (sexp-read chr port reader))))
```

When this procedure is invoked, *reader* points to the `#` sub-reader.

4.1.4 Token Reader Library

Guile-Reader comes with a number of re-usable token readers. Together, they might be assembled to form a complete Scheme reader equivalent to that of Guile (see Section 4.2.2 [Reader Library], page 19). Or they can be used individually in any reader.

The `standard-token-reader` procedure takes a symbol that names a standard TR from the library and returns it (or `#f` if not found). Currently, the available TRs are:

Token Reader	Character Spec.	Description
`boolean`	4 characters, `#\f`... `#\F`	This is a sharp token reader, i.e. it reads an R5RS boolean (`#f` or `#F`, `#t` or `#T`) once a `#` character has been read.
`boolean-srfi-4`	3 characters, `#\t`... `#\F`	This is a sharp token reader, i.e. it reads an R5RS boolean (`#t`, `#T`, `#F`, but *not* `#f`) once a `#` character has been read. Compared to the `boolean` token reader, this one is useful when SRFI-4 floating-point homogeneous vectors are to be used at the same time: the SRFI-4 TR will handle `#f` on its own (see Section 4.1.5.2 [Overlapping Token Readers], page 17).
`brace-free-number`	13 characters, `#\-`... `#\9`	Return a number or a symbol, considering curly braces as delimiters.
`brace-free-symbol-lower-case`	from `#\a` to `#\z`	Read a symbol that starts with a lowercase letter and return a symbol. This token reader recognizes braces as delimiters, unlike R5RS/R6RS.
`brace-free-symbol-misc-chars`	17 characters, `#\[`... `#\$`	Read a symbol that starts with a non-alphanumeric character and return a symbol. This token reader recognizes braces as delimiters, unlike R5RS/R6RS.
`brace-free-symbol-upper-case`	from `#\A` to `#\Z`	Read a symbol that starts with an uppercase letter and return a symbol. This token reader recognizes braces as delimiters, unlike R5RS/R6RS.

`character`	`#\\`	This is a sharp token reader, i.e. it reads an R5RS character once a `#` character has been read.
`curly-brace-sexp`	`#\{`	Read an S-expression enclosed in square brackets. This is already permitted by a number of Scheme implementations and will soon be made compulsory by R6RS.
`guile-bit-vector`	`#\*`	This is a sharp token reader, i.e. it reads a bit vector following Guile's read syntax for bit vectors. See See Info file '`guile`', node '`Bit Vectors`', for details.
`guile-extended-symbol`	`#\{`	This is a sharp token reader, i.e. it reads a symbol using Guile's extended symbol syntax assuming a `#` character was read. See See Info file '`guile`', node '`Symbol Read Syntax`', for details.
`guile-number`	13 characters, `#\-`... `#\9`	Read a number following Guile's fashion, that is, as in R5RS (See Info file '`r5rs`', node '`Lexical structure`', for syntactic details). Because the syntaxes for numbers and symbols are closely tight in R5RS and Guile, this token reader may return either a number or a symbol. For instance, it will be invoked if the string `123.123.123` is passed to the reader but this will actually yield a symbol instead of a number (see Section 4.1.5.2 [Overlapping Token Readers], page 17).
`guile-symbol-lower-case`	from `#\a` to `#\z`	Read a symbol that starts with a lower-case letter in a case-sensitive fashion.
`guile-symbol-misc-chars`	19 characters, `#\[`... `#\$`	Read a symbol that starts with a non-alphanumeric character in a case-sensitive fashion. Note that this token reader does *not* consider square brackets as delimiters, as was the case with Guile 1.8 and earlier.
`guile-symbol-upper-case`	from `#\A` to `#\Z`	Read a symbol that starts with an upper-case letter in a case-sensitive fashion.

`keyword`	`#\:`	This token reader returns a keyword as found in Guile. It may be used either after a `#` character (to implement Guile's default keyword syntax, `#:kw`) or within the top-level reader (to implement `:kw`-style keywords). It is worth noting that this token reader invokes its top-level in order to read the symbol subsequent to the `:` character. Therefore, it will adapt to the symbol delimiters currently in use (see Section 4.1.5.1 [Token Delimiters], page 16).
`number+radix`	12 characters, `#\b`... `#\E`	This is a sharp token reader, i.e. it reads a number using the radix notation, like `#b01` for the binary notation, `#x1d` for the hexadecimal notation, etc., see See Info file '`guile`', node '`Number Syntax`', for details.
`quote-quasiquote-unquote`	3 characters, `#\'`... `#\,`	Read a quote, quasiquote, or unquote S-expression.
`r5rs-lower-case-number`	13 characters, `#\-`... `#\9`	Return a number or a lower-case symbol.
`r5rs-lower-case-symbol-lower-case`	from `#\a` to `#\z`	Read a symbol that starts with a lower-case letter and return a lower-case symbol, regardless of the case of the input.
`r5rs-lower-case-symbol-misc-chars`	19 characters, `#\[`... `#\$`	Read a symbol that starts with a non-alphanumeric character and return a lower-case symbol, regardless of the case of the input.
`r5rs-lower-case-symbol-upper-case`	from `#\A` to `#\Z`	Read a symbol that starts with an upper-case letter and return a lower-case symbol, regardless of the case of the input.
`r5rs-upper-case-number`	13 characters, `#\-`... `#\9`	Return a number or an upper-case symbol.
`r6rs-number`	13 characters, `#\-`... `#\9`	Return a number or a symbol. This token reader conforms to R6RS, i.e. it considers square brackets as delimiters.

`r6rs-symbol-lower-case`	from `#\a` to `#\z`	Read a symbol that starts with a lowercase letter and return a symbol. This token reader conforms with R6RS in that it is case-sensitive and recognizes square brackets as delimiters (see Section 4.1.5.1 [Token Delimiters], page 16).
`r6rs-symbol-misc-chars`	17 characters, `#\{`... `#\$`	Read a symbol that starts with a non-alphanumeric character and return a symbol. This token reader conforms with R6RS in that it is case-sensitive and recognizes square brackets as delimiters (see Section 4.1.5.1 [Token Delimiters], page 16).
`r6rs-symbol-upper-case`	from `#\A` to `#\Z`	Read a symbol that starts with an uppercase letter and return a symbol. This token reader conforms with R6RS in that it is case-sensitive and recognizes square brackets as delimiters (see Section 4.1.5.1 [Token Delimiters], page 16).
`r6rs-syntax-quote-quasiquote-unquote`	3 characters, `#\'`... `#\,`	Read an R6RS-style syntax, quasisyntax, or unsyntax S-expression.
`scsh-block-comment`	`#\!`	This is a sharp token reader, i.e. it reads a SCSH-style block comment (like `#! multi-line comment !#`) and returns `*unspecified*`, assuming a `#` character was read before. This token reader has its "escape" bit set, meaning that the reader that calls it will return `*unspecified*` to its parent reader. See also See Info file '`guile`', node '`Block Comments`', for details about SCSH block comments.
`semicolon-comment`	`#\;`	Read an R5RS semicolon line-comment and return `*unspecified*`. Consequently, the calling reader will loop and ignore the comment.
`sexp`	`#\(`	Read a regular S-expression enclosed in parentheses.

`skribe-exp`	`#\[`	Read a Skribe markup expression. Skribe's expressions look like this:
		`[Hello ,(bold [World])!]`
		`=> ("Hello " (bold "World") "!")`
		See the Skribe web site or the Skribilo web site for more details.

`square-bracket-sexp`	`#\[`	Read an S-expression enclosed in square brackets. This is already permitted by a number of Scheme implementations and will soon be made compulsory by R6RS.
`srfi-4`	3 characters, `#\s`... `#\f`	This is a sharp token reader, i.e. it reads an SRFI-4 homogenous numeric vector once a `#` character has been read. This token reader also handles the boolean values `#f`.
`srfi30-block-comment`	`#|`	This is a sharp token reader, i.e. it reads an SRFI-30 block comment (like `#| multi-line comment |#`) and returns `*unspecified*`, assuming a `#` character was read before. This token reader has its "escape" bit set. For more details about SRFI-30, see Nested Multi-line Comments.
`srfi62-sexp-comment`	`#\;`	This is a sharp token reader, i.e. it reads an SRFI-62 comment S-expression (as in `(+ 2 #;(comment here) 2)`) and returns `*unspecified*`, assuming a `#` character was read before. This token reader has its "escape" bit set. For more details about SRFI-62, please see S-expression comments specifications.
`string`	`#\"`	Read an R5RS string.
`vector`	`#\(`	This is a sharp token reader, i.e. it reads an R5RS vector once a `#` character has been read.
`whitespace`	from `#\soh` to `#\space`	This is a void token reader that causes its calling reader to ignore (i.e. treat as white-space) all ASCII characters ranging from 1 to 32.

As can be inferred from the above two lists, reading character sequences starting with the `#` characters can easily be done by defining a sub-reader for that character. That reader can then be passed to `make-token-reader` as the procedure attached to `#`:

```
(define sharp-reader
  (make-reader (map standard-token-reader
                    '(boolean character
                      number+radix keyword
                      srfi-4
                      block-comment))))

(define top-level-reader
  (make-reader (list (make-token-reader #\# sharp-reader)
                     ...
                     )))
```

The procedures available to manipulate token readers are listed below:

`token-reader-escape?` *tr* [Scheme Procedure]
`scm_token_reader_escape_p` (*SCM tr*) [C Function]
Return `#t` if token reader *tr* requires the readers that use it to return even if its return value is unspecified.

`token-reader-specification` *tr* [Scheme Procedure]
`scm_token_reader_spec` (*SCM tr*) [C Function]
Return the specification, of token reader *tr*.

`token-reader-procedure` *tr* [Scheme Procedure]
`scm_token_reader_proc` (*SCM tr*) [C Function]
Return the procedure attached to token reader *tr*. When `#f` is returned, the *tr* is a "fake" reader that does nothing. This is typically useful for whitespaces.

`standard-token-reader` *name* [Scheme Procedure]
`scm_standard_token_reader` (*SCM name*) [C Function]
Lookup standard token reader named *name* (a symbol) and return it. If *name* is does not name a standard token reader, then an error is raised.

4.1.5 Limitations

This section describes the main limitations and common pitfalls encountered when using Guile-Reader.

4.1.5.1 Token Delimiters

As can be seen from the previous section, there exist, for instance, an surprisingly high number of symbol token readers. The reason for this is that different syntax variants define different *token delimiters*. Token delimiters are characters that help the reader determine where tokens that require implicit termination do terminate. Quoting R5RS (see section "Lexical structure" in *Revised^5 Report on the Algorithmic Language Scheme*):

> Tokens which require implicit termination (identifiers, numbers, characters, and dot) may be terminated by any <delimiter>, but not necessarily by anything else.

R5RS defines token delimiters as one of the following: a whitespace, a parentheses, a quotation mark (`"`) or a semi-colon (`;`) character. On the other hand, R6RS, which is to support the ability to use square brackets instead of parentheses for S-expressions, also considers square brackets as token delimiters. Likewise, if we were to support curly braces to enclose S-expressions, then curly braces would need to be considered as token delimiters too.

For this reason, the token reader library comes with several symbol token readers: the `guile-symbol-` family does not consider square brackets as delimiters while the `r6rs-symbol-` family does, the `brace-free-` TR family considers curly braces as delimiters but not square brackets, etc. Similarly, several variants of number TRs are available. This is due to the fact that number TRs may return symbols in corner cases like symbol names starting with a number.

However, although keywords must also comply with the token delimiters rules, there is only one keyword TR (called `keyword`). The reason for this is that this TR relies on the top-level reader's symbol reader to read the symbol that makes up the keyword being read.

In the current design of Guile-Reader, this token delimiter issue creates a number of pitfalls when one is willing to change the current delimiters. In particular, one has to be very careful about using TRs that consistently assume the same token delimiters.

A "real" lexer generator such as Danny Dubé's SILex avoids such issues because it allows the definition of tokens using regular expressions. However, its usage may be less trivial than that of Guile-Reader.

4.1.5.2 Overlapping Token Readers

As can be seen from the descriptions of the standard token readers (see Section 4.1.4 [Token Reader Library], page 11), token readers sometimes "overlap", i.e., the set of input strings they match overlap. For instance, the `boolean` token reader should match `#t`, `#T`, `#f` or `#F`. However, the `srfi-4` token reader also needs to match floating-point numeric vectors such as `#f32(1.0 2.0 3.0)`. Similarly, strings like `1` are, logically, handled by the `guile-number` (or similar) token reader; however, since a string like `1+` should be recognized as a *symbol*, rather than a number, it must then be passed to one of the symbol token readers.

In those two cases, the input sets of those two token readers *overlap*. In order for the resulting reader to work as expected, the two overlapping token readers need to somehow *cooperate*. In the first example, this is achieved by having the `srfi-4` TR read in strings starting with `#f` or `#F` and passing them to the `boolean-srfi-4` TR if need be. In the second case, this is done by having number TRs (e.g., `guile-number`) explicitly check for non-digit characters and return a symbol instead of a number when a non-digit is encountered.

It should be obvious from these two examples that this limitation impedes full separation of the various TRs. Fortunately, there are not so many cases where such overlapping occurs when implementing readers for R5RS-like syntaxes. The implementation of `make-alternate-guile-reader` (see Section 4.2.2 [Reader Library], page 19) shows how such problems have been worked around.

Lexer generators such as Flex, SILex and Bigloo's RGC (see section "Regular Parsing" in *Bigloo, A "Practical Scheme Compiler"—User Manual*) obviously do not have this problem: all possible "token" types are defined using regular expressions and the string-handling code

(e.g., code that converts a string into a Scheme number) is only invoked once a full matching string has been found.

4.2 Readers

Guile-Reader is about defining readers. Continuing to read this manual was definitely a good idea since we have finally reached the point where we will start talking about how to define new readers.

4.2.1 Defining a New Reader

Roughly, a reader is no more than a loop which reads characters from a given port, and dispatches further interpretation to more specific procedures. Written in Scheme, it could resemble something like:

```
(define (my-reader port)
  (let loop ((result *unspecified*))
    (let ((the-char (getc port)))
      (case the-char
        ((#\()  (my-sexp-token-reader the-char port my-reader)))
        ((#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)
         (my-number-token-reader the-char port my-reader))
        (else
         (error "unexpected character" the-char))))))
```

Using Guile-Reader, this is done simply by providing a list of token readers to the `make-reader` procedure, as in the following example:

```
(define my-reader
  (make-reader (list (make-token-reader #\( my-sexp-token-reader)
                     (make-token-reader '(#\0 . #\9)
                                        my-number-token-reader))))
```

However, the procedure returned by `make-reader` is different from the hand-written one above in that in takes two additional optional arguments which makes it look like this:

```
(define (my-reader port faults-caller-handled? top-level-reader)
  (let loop ((the-char (getc port)))
    (case the-char
      ...
      (else
        (if (not faults-caller-handled?)
            (error "unexpected character" the-char)
            (ungetc the-char) ;; and return *unspecified*
            )))))
```

Therefore, by default, `my-reader` will raise an error as soon as it reads a character that it does not know how to handle. However, if the caller passes `#t` as its `faults-caller-handled?` argument, then `my-reader` is expected to "unget" the faulty character and return `*unspecified*`, thus allowing the caller to handle the situation.

This is useful, for instance, in the S-exp token reader example: the S-exp token reader needs to call its calling reader in order to read the components between the opening and

closing brackets; however, the calling reader may be unable to handle the `#\)` character so the S-exp token reader has to handle it by itself and needs to tell it to the reader.

`%reader-standard-fault-handler` *chr port reader* [Scheme Procedure]
`scm_reader_standard_fault_handler` (*SCM chr, SCM port, SCM reader*) [C Function]
Throw a `read-error` exception indicating that character *chr* was read from *port* and could not be handled by *reader*.

`make-reader` *token-readers* [*fault-handler-proc* [*flags...*]] [Scheme Procedure]
`scm_make_reader` (*SCM token_readers, SCM fault_handler_proc, SCM flags*) [C Function]
Create a reader made of the token readers listed in *token_readers*. *token_readers* should be a list of token readers returned by `make-token-reader` or `standard-token-reader` for instance. The *fault_handler_proc* argument is optional and may be a three-argument procedure to call when an unexpected character is read. When *fault_handler_proc* is invoked, it is passed the faulty character, input port, and reader; its return value, if any, is then returned by the reader. If *fault_handler_proc* is not specified, then `%reader-standard-fault-handler` is used. *flags* is a rest argument which may contain a list of symbols representing reader compilation flags.

Currently, the flags that may be passed to `make-reader` are the following:

- `reader/record-positions` will yield a reader that records the position of the expression read, which is mostly useful for debugging purposes; this information may then be accessed via source properties (see section "Procedure Properties" in *Guile Reference Manual*).
- `reader/lower-case` will have the yielded reader convert to lower-case all the letters that it reads; note that this is not sufficient to implement symbol case-insensitivity as shown in section "Reader options" in *Guile Reference Manual*. For this, the token reader(s) that read symbols must also convert all subsequent characters to lower-case.
- `reader/upper-case` will have the yielded reader convert to upper-case all the letters that it reads; again, that is not sufficient to implement case-insensitivity.
- `reader/debug` causes the generated reader to produce debugging output.

4.2.2 Reader Library

The `(system reader)` module exports the `default-reader` procedure which returns a reader equivalent to Guile's built-in default reader made of re-usable token readers written in C (see Section 4.1.4 [Token Reader Library], page 11).

`default-sharp-reader-token-readers` [Scheme Procedure]
`scm_default_sharp_reader_token_readers` (*void*) [C Function]
Return the list of token readers that comprise Guile's default reader for the `#` character.

`default-reader-token-readers` [Scheme Procedure]
`scm_default_reader_token_readers` (*void*) [C Function]
Return the list of token readers that comprise Guile's default reader.

`default-sharp-reader` [Scheme Procedure]
`scm_default_sharp_reader` (*void*) [C Function]
Returns Guile's default reader for the `#` character.

Additionally, the `(system reader library)` module exports a number of procedures that ease the re-use of readers.

`make-guile-reader` [*fault-handler* [*flags...*]] [Scheme Procedure]
`scm_make_guile_reader` (*SCM fault_handler, SCM flags*) [C Function]
Make and return a new reader compatible with Guile 2.0's `read`, with its default settings. This function calls `make-reader` with *flags*. Note that the sharp reader used by the returned reader is also instantiated using *flags*. The value of *fault-handler* defaults to `%reader-standard-fault-handler`.

`make-alternate-guile-reader` [Scheme Procedure]
Return a newly created Guile reader with options *options* (a list of symbols, as for `alternate-guile-reader-token-readers`), with fault handler *fault-handler* and flags *flags*. The *fault-handler* and *flags* arguments are the same as those passed to `make-reader`. By default, *fault-handler* is set to *%reader-standard-fault-handler*.

`alternate-guile-reader-token-readers` [Scheme Procedure]
Given *options*, a list of symbols describing reader options relative to the reader returned by `(default-reader)`, return two lists of token readers: one for use as a sharp reader and the other for use as a top-level reader. Currently, the options supported are the following:

`no-sharp-keywords`
Remove support for `#:kw`-style keywords.

`dsssl-keywords`
Add support for DSSSL-style keywords, like `#!kw`. This option also has the same effect as `no-scsh-block-comments`.

`colon-keywords`
Add support for `:kw`-style keywords. This is equivalent to `(read-set! keywords 'prefix)`.

`no-scsh-block-comments`
Disable SCSH-style block comments (see See Info file '`guile`', node '`Block Comments`', for details).

`srfi30-block-comments`
Add support for SRFI-30 block comments, like:

```
(+ 2 #| This is an #| SRFI-30 |# comment |# 2)
```

`srfi62-sexp-comments`
Add support for SRFI-62 S-expression comments, like:

```
(+ 2 #;(a comment) 2)
```

`case-insensitive`
Read symbols in a case-insensitive way.

`square-bracket-sexps`
`square-brackets`
: Allow for square brackets around S-expressions.

`read-options->extended-reader-options` [Scheme Procedure]
: Read *read-opts*, a list representing read options following Guile's built-in representation (see See Info file '`guile`', node '`Scheme Read`', for details), and return a list of symbols represented "extended reader options" understood by `make-alternate-guile-reader` et al.

4.2.3 Compatibility and Confinement

Guile's core read subsystem provides an interface to customize its reader, namely via the `read-options` (see section "Scheme Read" in *Guile Reference Manual*) and `read-hash-extend` (see section "Reader Extensions" in *Guile Reference Manual*) procedures.

The main problem with this approach is that changing the reader's options using these procedures has a global effect since there is only one instance of `read`. Changing the behavior of a single function at the scale of the whole is not very "schemey" and can be quite harmful. Suppose a module relies on case-insensitivity while another relies on case-sensitivity. If one tries to use both modules at the same time, chances are that at least one of them will not work as expected. Risks of conflicts are even higher when `read-hash-extend` is used: imagine a module that uses DSSSL-style keywords, while another needs SCSH-style block comments.

In `(system reader confinement)`, guile-reader offers an implementation of `read-option-interface` and `read-hash-extend` that allows to *confine* such settings on a per-module basis. In order to enable reader confinement, one just has to do this:

```
(use-modules (system reader confinement))
```

Note that this must be done before the suspicious modules are loaded, that is, typically when your program starts. This will redefine `read-options-interface` and `read-hash-extend` so that any future modification performed via Guile's built-in reader option interface will be confined to the calling module.

Starting from Guile 1.8.0, `current-reader` is a core binding bound to a fluid whose value should be either `#f` or a reader (i.e., a `read`-like procedure). The value of this fluid dictates the reader that is to be used by `primitive-load` and its value can be changed dynamically (see section "Loading" in *Guile Reference Manual*).

The confined variants of `read-options-interface` and `read-hash-extend` rely on this feature to make reader customizations local to the file being loaded. This way, invocations of these functions from within a file being loaded by `primitive-load` take effect immediately.

5 Internals

In order to not have to trade too much performance for flexibility, Guile-Reader dynamically compiles code for the readers defined using GNU *lightning* (see section "Overview" in *Using and Porting GNU lightning*). As of version 1.2c, GNU *lightning* can generate code for the PowerPC, SPARC, and IA32 architectures. For other platforms, Guile-Reader provides an alternative (slower) C implementation that does not depend on it. Using the *lightning*-generated readers typically provides a 5% performance improvement over the static C implementation. However, note that *lightning* 2.x is not supported yet.

Re-using token readers written in C, as explained in See Section 4.1.4 [Token Reader Library], page 11, does not imply any additional cost: the underlying C function will be called directly by the reader, without having to go through any marshalling/unmarshalling stage.

Additionally, on the C side, token readers may be initialized *statically* (except, obviously, token readers made out of a dynamically-compiled reader). Making good use of it can improve the startup time of a program. For example, `make-guile-reader` (see Section 4.2.2 [Reader Library], page 19) is implemented in C and it uses statically initialized arrays of token readers. It still needs to invoke `scm_c_make_reader ()`, but at least, token readers themselves are "ready to use".

Scanners as generated by Flex or similar tools should theoretically be able to provide better performance because the input reading and pattern matching loop is self-contained, may fit in cache, and only has to perform function calls once a pattern has been fully recognized.

Concept Index

Function Index