

Parsing Strings with Scheme

Using the `*parser` library of MIT/GNU Scheme

Aaron S. Hawley (ashawley@gnu.uvm.edu)

Copyright © 2005 Aaron S. Hawley

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being “Free Documentation,” and with the Back-Cover Texts as in (a) below. A copy of the license is available from the Free Software Foundation Web site at <http://www.fsf.org/licenses/fdl.html>.

(a) The Back-Cover Text is: “You have freedom to copy and modify this free document, as you would free software.”

The document was typeset with GNU Texinfo (<http://www.texinfo.org/>). The document’s source file is `parsing-strings.texi`. It is available from <http://www.uvm.edu/~ashawley/scheme/>.

\$Date: 2005/10/06 15:52:06 \$

\$Revision: 1.12 \$

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Overview	1
1.3	Assumptions	2
1.4	External References	2
1.5	Note to Readers Using Emacs	2
2	Getting Started	3
2.1	The MIT Scheme Parser	3
2.2	Using the Parser	3
3	The Tutorial	4
3.1	String Literals	4
3.2	An Initial Parser	4
3.3	Explanation	5
3.4	Testing the Parser	5
3.5	Matching an Escaped Quotation Mark	6
3.6	Removing the Start and End Quotation Marks	7
3.7	Matching Other Escape Characters	7
3.8	Fine-tuning the Return Value More	8
3.9	Comparing the Parser with read	9
3.10	Parsing All Strings	10
4	Conclusion	12
Appendix A	'parse-string.scm'	13
Appendix B	Parsing with read	14

1 Introduction

This manual introduces parsing in MIT/GNU Scheme by giving a tutorial on how to parse string literals.

Impatient readers can skip to the entire source code file of the final parser implementation in Appendix A [parse-string.scm], page 13. The tutorial could then function to help explain confusing aspects of the source code file.

The article begins and concludes with arguments for writing a parser in the Scheme programming language. After the tutorial's scope and intended audience are also introduced, pointers are given to technical and other relevant sources of documentation useful for understanding the tutorial. Information on executing the examples is provided for readers viewing the tutorial from inside an Emacs editor.

1.1 Motivation

The fun in using the *Scheme programming language* doesn't come from only repeating the same tired pedagogical examples, like writing a recursive solution for printing Fibonacci's numbers or writing yet another Scheme interpreter. Scheme can accomplish worthwhile tasks. And counter to the rumor, some Scheme implementations come with libraries to help programmers complete such tasks. These libraries are often as high-level, graceful and useful as the Scheme language itself. One such Scheme implementation—that also comes documented quite well—is MIT/GNU Scheme (hereafter “MIT Scheme”).

1.2 Overview

The following is a tutorial to writing a parser in MIT Scheme. *Parsers* are defined in computer science as software that can extract information from a stream of data. For example, reading the words in a sentence requires a parser. Reading the structure and keywords of computer software code requires a parser. Besides MIT Scheme, the human brain has parsing facilities for reading words and for identifying objects (like shapes or colors).

Most introductions to parsing start with infix arithmetic expressions like ‘1 + 1’. Instead, we shall introduce parsing a simpler construct, the *string literal*. String literals are any number of characters found between two quotation marks (“”). Special scenarios usually exist for representing a quotation mark character inside of string literals. Those are explained later.

By parsing only string literals, the tutorial covers what is more often termed *lexing* by classical computer science. A *lexer* lexes by converting a stream of data into *lexemes*. Lexemes are the individual tokens matched by a parser. In this tutorial, the lexemes are string literals. When a literate human brain reads sentences, it lexes words. This distinction exists likely because conventional programming languages needed the tasks separated to handle parsing and to handle *left recursion* for parsing languages that use *infix notation*. The distinction between lexing and parsing is not acknowledged any further in this document.

To create a parser there must be a defined *grammar*. A grammar describes some desired thing that can be parsed. The desired thing may be a valid sentence in the English language, but is a string literal in this tutorial. Only with a well-defined grammar, the task of designing a parser can be accomplished. Writing a parser

1.3 Assumptions

The tutorial assumes you understand the basic concepts of a grammar and how to represent them. Two common and related notations for grammars, *Regular Expressions* and *Backus-Naur Form* (BNF), are useful to know in this tutorial.

It is assumed you are familiar with programming in Scheme, too.

1.4 External References

Information on the Scheme programming language available from MIT Scheme and how to install and use MIT Scheme is available elsewhere as free documentation. See section “Overview” in *MIT/GNU Scheme Reference Manual*, or section “Introduction” in *MIT/GNU Scheme Reference Manual*, respectively.

Developed since at least the 1980s, MIT Scheme is a complete programming environment. It is now distributed as *free software*, was adopted as a *GNU package*, and has been used for decades to teach programming. More information on MIT Scheme is available at the MIT Scheme Web site (<http://www.gnu.org/software/mit-scheme/>).

1.5 Note to Readers Using Emacs

Readers reading the tutorial in the GNU Info format from within GNU Emacs or the Edwin editor (the Emacs-like editor that comes with MIT Scheme) are encouraged to evaluate each piece of example code with the command `C-x C-e`. For Emacs, evaluating Scheme expressions in `Info-mode` requires running an *inferior* Scheme process in a separate buffer followed by binding the key to the `scheme-send-last-sexp` command.

```
M-x run-scheme RET
```

```
C-x b RET
```

```
M-x local-set-key RET C-x C-e scheme-send-last-sexp RET
```

```
M-x local-set-key RET C-M-x scheme-send-definition RET
```

Typing `C-x C-e` with the point (cursor) at the end of the last parenthesis of a Scheme expression evaluates the expression in the `*scheme*` buffer and print its evaluated value. Typing `C-M-x` anywhere inside a Scheme definition evaluates the definition.

2 Getting Started

2.1 The MIT Scheme Parser

In MIT Scheme, the parser and matcher are provided as separate libraries to avoid conflicting with the default standard Scheme language provided by MIT Scheme. This is the scenario for creating parsers in other programming languages as well.

Writing parsers in other programming languages often requires writing in a separate language that must be run through a separate tool. In MIT Scheme, a parser is written entirely in a Scheme-like syntax, does not require any separate utilities and can be combined with any other MIT Scheme code or features of MIT Scheme.

2.2 Using the Parser

The parsing and matching libraries of MIT Scheme are called the *star-parser* (`*parser`) and *star-matcher* (`*matcher`). The syntax for the `*parser` and `*matcher` are inspired by and therefore similar to Regular Expressions and BNF. This makes MIT Scheme's parser and matcher languages high-level and simple to use. The matcher returns true (`#t`) on success and false (`#f`) for a failure to match. The parser differs from this by returning a vector containing each successfully found token. Besides the parsers enhancements to modify the parsed value, the syntaxes of the parser and matcher systems are almost entirely interchangeable.

To use the parser in MIT Scheme you must load the `*parser` using the following:

```
(load-option '*parser)
```

3 The Tutorial

3.1 String Literals

The *string literal* is an object in Scheme and many other programming languages to represent strings of characters. Here are four examples of string literals (comments are inline at the right of each example beyond semicolons as Scheme comments):

```
"foo"

"foo bar"

"" ;; The empty string

"\foo\" ;; The string "foo"
```

Looking at these examples, a parser of string literals would need to find characters with a start quotation mark character and end with a quotation mark character. The actual string literal between the quotation marks could contain zero or more characters.

Notice that putting quotation mark characters in a string literal requires *escaping* them with the *backslash character* (`\`). This is an idiom understood in most programming languages including Scheme, and will later be understood by the parser.

3.2 An Initial Parser

At first, the parser overlooks the escape character issue and therefore ignore string literals containing quotation marks. Instead, any characters between quotation marks is accepted by the parser. We could describe this simple parser as a procedure that:

1. looks for a quotation mark,
2. reads as many characters as possible without reading a quotation mark,
3. and then finds a quotation mark.

This parser is described with MIT Scheme in the following definition:

```
;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (seq
    (match "\"")
    (* (match (not-char #\")))
    (match "\""))))
```

Notice that `parse-string` is not defined as a typical Scheme function with specified function arguments (historically called a “defun”) nor assigned a `lambda` expression. It is not immediately clear how to use the `parse-string` function created with the `*parser`. Instead of a function, `parse-string` looks like a variable assigned the value resulting from the `*parser`-syntax. The single argument to `*parser` is called a “syntax” even though it appears `*parser` is a function taking a single argument. According to the documentation in section “Parser Language” in *MIT/GNU Scheme Reference Manual*, `*parser` is implemented as a *macro*.

What does it mean that `parse-string` is a macro? The parser definition `parse-string` could be thought of as a description of another parser. The `*parser` macro uses the parser language syntax in the sub-expression to create a corresponding larger (thus “macro”), parser function. The *macro* parser created handles the specific details of reading and matching input, and is applied to a *parser buffer*—a buffer of data that parser functions are capable of operating on (This is what is hinted in the source code comment before the definitions of `parse-string`). Intelligently, all of these complicated details of actually parsing the data in the buffer are abstracted and hidden. Before explaining how to use the `parse-string` parser function on an parser buffer and testing it, the parser language syntax is explained first.

3.3 Explanation

Inside the above `*parser` expression is a `seq` expression. The `seq` expression guarantees the sub-expressions are matched *sequentially* on the data. The `seq` used above matches a quotation mark followed by anything but a quotation mark followed by a closing quotation mark. The order of this sequence is critical.

The `match` expressions, as intuition hints, match their sub-expression. If the sub-expression is a string, then the data are matched against the string, but other `*matcher` expressions are allowed in `match`. For instance, the `not-char` expression matches any character other than the character provided to `not-char`.

The star (*) expression matches its sub-expression zero or more times. In the parser above, the `*` expression matches any character—that is not a quotation mark—zero or more times.

For those familiar with BNF, the source code for the parser maps quite closely to a BNF representation of the grammar. The syntax differs with its “reverse Polish notation” as adopted generally by Scheme.

```
<string> ::= "\"" <not-quote>* "\""
<not-quote> ::= "a" | "b" | "c" | ... | (anything but a "\"" ....)
```

Not even Extended BNF (EBNF) can easily capture the grammar the `parse-string` parser accepts, without adding the ellipsis “hack”.

3.4 Testing the Parser

The small parser above attempts to match a simple string literal. Testing the parser requires having a parser buffer. Fortunately, the `string->parser-buffer` procedure can create parser buffers from strings. The parser buffer created by `string->parser-buffer` can then be passed as an argument to the `parse-string` procedure.

```
(parse-string (string->parser-buffer "\"foo\""))
=> #("\"" "f" "o" "o" "\"") ;; "foo"

(parse-string (string->parser-buffer "foo"))
=> #f

(parse-string (string->parser-buffer "foo\""))
=> #f
```

```

(parse-string (string->parser-buffer "\"foo\""))
=> #f

(parse-string (string->parser-buffer "\"\""))
=> #("\"" "\"") ;; ""

(parse-string (string->parser-buffer ""))
=> #f

(parse-string (string->parser-buffer "\"foo\" \"foo\""))
=> #("\"" "f" "o" "o" "\"") ;; "foo"

(parse-string (string->parser-buffer "bar \"foo\""))
=> #f

(parse-string (string->parser-buffer "\"foo\" bar"))
=> #("\"" "f" "o" "o" "\"") ;; "foo"

```

The tests of the initial parser displays some noteworthy behavior. When a leading quotation mark was not found right away or a closing quotation mark was never found the parser fails and returns false. When there were extra characters, including string literals, after the string literal, the parsing would succeed and ignore the trailing characters. On success, the parser function returns a vector an element for each successful match with the `match` function. The element returned is each character matched by the three uses of `match` in `parse-string`. To the right of the result in Scheme comments is a more readable version of the result.

These tests, although maybe not exhaustive, have verified that the `parse-string` parser works for simple string literals. The tutorial will improve the parser by having it accept a broader definition of string literals and tune the values returned from parsing successfully.

3.5 Matching an Escaped Quotation Mark

The next logical step is to allow quotation marks to exist in a string literal. To have the parser accept quotation characters in strings, we need to match “\” —an *escaped* quotation mark. When writing an escaped quotation mark in Scheme, quotation marks *and* backslashes need to be escaped. The escaped quotation mark is represented in a string value as “\\” in Scheme—two backslashes to make a backslash, and another backslash to escape the quotation mark.

Where the parser used to match zero or more non-quotation mark characters needs to match both escaped-quotation characters and non-quotation characters. The “both” relation can be handled with the `alt` parser expression. Unlike `seq`, `alt` allows either subexpression to match and satisfies the first one that matches.

```

; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (seq

```

```
(match "\"")
(* (alt (match "\\\"")
      (match (not-char #\"))))
(match "\""))
```

Tests of the new definition confirm that it works.

```
(parse-string (string->parser-buffer "\"\\\\"foo\\\\"\""))
=> #("\"" "\\\" "f" "o" "o" "\\\" "\") ;; "\"foo\""
```

Running the above test on the old definition of `parse-string` would return a different value and an incorrect parsing of the string literal.

```
(parse-string (string->parser-buffer "\"\\\\"foo\\\\"\"\"\""))
=> #("\"" "\\\" \"\"") ;; "\""
```

Incorrectly, it returns only the starting quotation mark, the escaping backslash, and then ends prematurely on the escaped quotation mark of the string literal.

3.6 Removing the Start and End Quotation Marks

The parser matches the leading and ending quotation mark characters and returns a vector value containing the delimiting quotation marks. Really, only the contained string of characters should be returned. To avoid this annoyance we can use the `noise` expression. The `noise` parser expression is equivalent to the `match` expression, except the match isn't included in the returned vector value. The use of `match` is replaced with `noise` for the start end ending quotation marks.

```
;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (seq
    (noise "\"")
    (* (alt (match "\\\"")
          (match (not-char #\"))))
    (noise "\""))))
```

Here are some tests to make sure it worked.

```
(parse-string (string->parser-buffer "\"foo\""))
=> #("f" "o" "o") ;; foo
```

```
(parse-string (string->parser-buffer "\"\\\\"foo\\\\"\""))
=> #("\\\" "f" "o" "o" "\\\"") ;; "\"foo\""
```

3.7 Matching Other Escape Characters

The parser now allows the quotation mark and removes the delimiting quotation marks, but it is returning any escaped quotation marks (“\\”) with slashes. The backslash before the quote should be removed in the returned value.

This should even be generalized to all escaped characters. For instance, the backslash character even needs to be escaped by another backslash character. This is not general-

ized for all characters. Some backslash character sequences represent special characters with specific meanings in computing. These include the *newline* (`'\n'`) and the *tab* (`'\t'`) characters.

To accept the sequence of a backslash character followed by either another backslash character or a quotation mark, the escaping backslash character is discarded using `noise`.

```
;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (seq
    (noise "\\")
    (* (alt (seq (noise "\\")
                 (alt (match "\\")
                      (match "\\")
                      (match "\\")
                      (match (not-char #\"))))
        (noise "\\")))))
```

In the following tests, the resulting vector has strings printed by MIT Scheme with escaped quotation mark characters. The backslashes used for escaping in the original string shall no longer be present in the parsed string.

```
(parse-string (string->parser-buffer "\\\"foo\\\""))
=> #("\\ " "f" "o" "o" "\\") ;; "foo"
```

```
(parse-string (string->parser-buffer "\\\"\\\"\\\""))
=> #("\\") ;; \
```

3.8 Fine-tuning the Return Value More

Another common expectation of a parser is the ability to determine the returned value of the parsed input. The parser currently returns a vector composed of each individually matched element. This is not a useful return value for use by other programs. Usually, the atomic “token” of the grammar should be the return value, not some representation determined by the underlying MIT Scheme parsing system. In this tutorial, a token should be the matched string literal, not each result coming from the `match` call.

Provided by the `*parser`, the `encapsulate` parser expression can modify a return value by applying a function to the vector returned by a parser function. For parsing string literals, `encapsulate` will need a function for creating a single string from each of the matched string elements in the vector.

To convert a vector of strings, the vector needs to be converted to a list with the function `vector->list`. Then, the list of strings created from the vector are converted into a single string. Lists of strings can't be converted into strings automatically in Scheme (Lists of characters can). Reducing a list of strings to a single string is done with the higher-order function `reduce`. The `reduce` procedure is often introduced with the example of adding a list of numbers together.

```
(reduce + 0 '(1 2 3 4))
=> 10
```

Instead of addition, the list of strings created with `vector->list` will be concatenated by the `reduce` function to “sum” the strings into a single final string.

```
(lambda (v) (reduce string-append "" (vector->list v)))
```

This function can be inserted in the parser as part of the `encapsulate` expression.

```
;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (encapsulate
    (lambda (v) (reduce string-append "" (vector->list v)))
    (seq
     (noise "\\")
     (* (alt (seq (noise "\\")
                  (alt (match "\\")
                       (match "\\"))))
        (match (not-char #\\"))))))
    (noise "\\")))))
```

This allows `parse-string` to give a clear result.

```
(parse-string (string->parser-buffer "\"foo\""))
=> #("foo")
```

The procedure to `encapsulate` the vector can be alternatively defined as a function. It will be called `vector-string->string`.

```
;; vector-string->string : string vector -> string
(define (vector-string->string v)
  (reduce string-append "" (vector->list v)))
```

With this definition we can define a more concise parser composed of reusable parts.

```
;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (encapsulate vector-string->string
    (seq
     (noise "\\")
     (* (alt (seq (noise "\\")
                  (alt (match "\\")
                       (match "\\"))))
        (match (not-char #\\"))))))
    (noise "\\")))))
```

3.9 Comparing the Parser with read

Ironically, the utility of this parsing tutorial emphasized in the introduction (see Chapter 1 [Introduction], page 1) was misleading. The example parser can successfully parse string literals. The parser is not an astonishing breakthrough. It is actually duplicating what is already available in Scheme.

Programming languages, including Scheme, parse string literals all the time. Scheme, unlike other programming languages, makes the internal Scheme parser available to the

user, to even use in their program. Therefore, somewhere in Scheme there exists the string literal parser, the exact capability of this tutorial's parser.

The `read` procedure available in MIT Scheme and all Scheme implementations can parse Scheme “objects” (a string literal is a Scheme object). Instead of reading from a parser buffer, `read` reads from a port. A port can be created from a string with MIT Scheme's `open-input-string` procedure. So, the tutorial's parser is entirely unoriginal research.

```
(read (open-input-string "\"foo\""))
=> "foo"
```

The parser defined in this tutorial is still useful, though. For instance, some programming tasks require parsers for parsing *only* strings (or numbers or some other value). Scheme's `read` function doesn't satisfy such a requirement. It accepts objects other than strings, including symbols, characters, numbers, lists and all other valid Scheme values and expressions. It also returns the objects as their Scheme types (strings as string type, symbols as symbol type, ...) and not as a vector of strings.

```
(read (open-input-string "foo"))
=> foo
```

A string literal parser written with MIT Scheme's `*parser` syntax would return false if it met these objects. Further, `read` does not return false for failed matches but on occasion will give a parse error.

```
(read (open-input-string ""))
=> PARSE-ERROR: Unmatched close paren #\)
```

The parser for literal strings could be rewritten using `read`. We've added the code for doing just that at the end of this tutorial in Appendix B [Parsing with read], page 14.

A parser written with MIT Scheme's `*parser` syntax and not with Scheme's `read` is a useful exercise.

The parser is taken further by having it match and return all string literals available in the parser buffer, allowing *whitespace* characters to exist between each string literal.

3.10 Parsing All Strings

To match all string literals, the entire parser need only match one or more string literals. The plus (+) expression is similar to the star (*) expression used elsewhere in the parser. Except, + must match *one* or more subexpressions rather than *zero* or more.

The + expression is added before the `encapsulate` expression. If it is put after, then the `encapsulate` expression would concatenate together all the matched string literals into one string literal rather than keeping them separate. Also, the matching of whitespace is also outside of the `encapsulate` command to also avoid being “encapsulated” into the returned value.

```
;; parse-all-strings : parser-buffer -> ( vector | #f )
(define parse-all-strings
  (*parser
   (+
    (alt
     (noise " ")
     (encapsulate vector-string->string
```

```

      (seq
        (noise "\"")
        (* (alt (seq (noise "\\")
                    (alt (match "\\")
                        (match "\""))))
           (match (not-char #\"))))))
      (match "\""))))
      (match (not-char #\"))))
      (noise "\"")))))))

```

The following tests show how `parse-all-strings` returns a vector element for each individual literal string matched.

```

(parse-all-strings (string->parser-buffer "\"\""))
=> #("")

```

```

(parse-all-strings (string->parser-buffer ""))
=> #f

```

```

(parse-all-strings (string->parser-buffer "\"foo\\\\"foo\""))
=> #("foo" "foo")

```

```

(parse-all-strings (string->parser-buffer "\"foo\" \\\bar\""))
=> #("foo" "foo")

```

Really, we don't need to duplicate the entire code of `parse-string` in `parse-all-strings`. The *modularity* of the MIT Scheme parser language, as is accomplished with the Scheme programming language generally, allow us to reuse parsers as recursive calls.

```

;; parse-all-strings : parser-buffer -> ( vector | #f )
(define parse-all-strings
  (*parser
   (+
    parse-string)))

```

4 Conclusion

The parsing capabilities in MIT Scheme are powerful, yet succinct and they are useful. Below are some of the qualities parsers created with MIT Scheme exhibit.

Modular parsing

A parser can be reused inside another parser, or separated into smaller manageable pieces. There is a macro language for adding features to a parser like seamless and implied error handling making a completed parser but still clearly represent its grammar.

Clear syntax

Instead of specifying *start condition* flags for enforcing parsing sequences or being limited by Regular Expressions, the MIT Scheme parsing language can be used to write human-readable parsers. This benefits the original author and others needing to make modifications or improvements to the parser. Complex Regular Expressions are notorious for being difficult to read, maintain and modify by other programmers. The parser syntax is verbose but is more powerful and lucid than Regular Expressions.

Intelligent technology

In commonly used programming languages, matching input requires reading in data to a variable and then operating on the variable. MIT Scheme requires only specifying the match and what to do with matched values. The detailed implementation that accomplishes the parsing of input is hidden. The resulting parser is optimized by MIT Scheme for performance and scale by reading input in smaller sequences to avoid bounds on the input size or the need to *backtrack* over the input. This makes a parser created with MIT Scheme not only useful and simple but extremely powerful.

Intelligent philosophy

There exists the slight possibility for a parser written in MIT Scheme and generated by MIT Scheme to be inadequate for any number of technical reasons, including parsing accuracy or runtime performance. This offers two possible solutions. One could *hack* their parser to correct for this situation by writing some or all of the parser in low-level Scheme code. Alternatively, one could investigate ways to improve MIT Scheme's underlying parser generation macro. Because MIT Scheme is *free software*, access is given to the source code for making or suggesting improvements to the package.

Appendix A 'parse-string.scm'

The following is the entire source code for the final version of the string literal parser written in MIT Scheme.

```
(load-option '*parser)

;; vector-string->string : string vector -> string
(define (vector-string->string v)
  (reduce string-append "" (vector->list v)))

;; parse-string : parser-buffer -> ( vector | #f )
(define parse-string
  (*parser
   (encapsulate vector-string->string
    (seq
     (noise "\"")
     (* (alt (seq (noise "\\")
                  (alt (match "\\")
                       (match "\""))
                       (match (not-char #\"))))
        (noise "\"")))))

;; parse-all-strings : parser-buffer -> ( vector | #f )
(define parse-all-strings
  (*parser
   (+
    parse-string)))
```

Appendix B Parsing with read

Here is how to define a string literal parser by using the `read` function provided by Scheme as mention in Section 3.9 [Comparing the Parser with ‘read’], page 9.

```
(define (parse-string #!optional port)
  (let* ((port (if (input-port? port) port
                  current-input-port))
        (object (read port)))
    (if (string? object) object
        #f)))
```

A version of the function `parse-all-strings` that was introduced in Section 3.10 [Parsing All Strings], page 10 can also be rewritten using `read`. It is left to the reader as an exercise.