Parser Tools: `lex` and `yacc`-style Parsing

Version 8.12

Scott Owens

February 4, 2024

This documentation assumes familiarity with `lex` and `yacc` style lexer and parser generators.

Contents

1 Lexers

```
(require parser-tools/lex)
```
package: parser-tools-lib

1.1 Creating a Lexer

```
(lexer maybe-suppress-warnings [trigger action-expr] ...)

maybe-suppress-warnings =
                        | #:suppress-warnings

               trigger = re
                       | (eof)
                       | (special)
                       | (special-comment)

                    re = id
                       | string
                       | character
                       | (repetition lo hi re)
                       | (union re ...)
                       | (intersection re ...)
                       | (complement re)
                       | (concatenation re ...)
                       | (char-range char char)
                       | (char-complement re)
                       | (id datum ...)
```

Produces a function that takes an input-port, matches the *re* patterns against the buffer, and returns the result of executing the corresponding *action-expr*. When multiple patterns match, a lexer will choose the longest match, breaking ties in favor of the rule appearing first.

The implementation of `syntax-color/racket-lexer` contains a lexer for the `racket` language. In addition, files in the `"examples"` sub-directory of the `"parser-tools"` collection contain simpler example lexers.

An *re* is matched as follows:

- *id* — expands to the named *lexer abbreviation*; abbreviations are defined via `define-lex-abbrev` or supplied by modules like `parser-tools/lex-sre`.
- *string* — matches the sequence of characters in *string*.
- *character* — matches a literal *character*.
- `(repetition` *lo hi re*`)` — matches *re* repeated between *lo* and *hi* times, inclusive; *hi* can be `+inf.0` for unbounded repetitions.

- `(union re ...)` — matches if any of the sub-expressions match
- `(intersection re ...)` — matches if all of the `re`s match.
- `(complement re)` — matches anything that `re` does not.
- `(concatenation re ...)` — matches each `re` in succession.
- `(char-range char char)` — matches any character between the two (inclusive); a single character string can be used as a `char`.
- `(char-complement re)` — matches any character not matched by `re`. The sub-expression must be a set of characters `re`.
- `(id datum ...)` — expands the *lexer macro* named `id`; macros are defined via `define-lex-trans`.

Note that both `(concatenation)` and `""` match the empty string, `(union)` matches nothing, `(intersection)` matches any string, and `(char-complement (union))` matches any single character.

The regular expression language is not designed to be used directly, but rather as a basis for a user-friendly notation written with regular expression macros. For example, `parser-tools/lex-sre` supplies operators from Olin Shivers's SREs, and `parser-tools/lex-plt-v200` supplies (deprecated) operators from the previous version of this library. Since those libraries provide operators whose names match other Racket bindings, such as `*` and `+`, they normally must be imported using a prefix:

```
(require (prefix-in : parser-tools/lex-sre))
```

The suggested prefix is `:`, so that `:*` and `:+` are imported. Of course, a prefix other than `:` (such as `re-`) will work too.

Since negation is not a common operator on regular expressions, here are a few examples, using `:` prefixed SRE syntax:

- `(complement "1")`

  Matches all strings except the string `"1"`, including `"11"`, `"111"`, `"0"`, `"01"`, `""`, and so on.

- `(complement (:* "1"))`

  Matches all strings that are not sequences of `"1"`, including `"0"`, `"00"`, `"11110"`, `"0111"`, `"11001010"` and so on.

- ```
  (:& (:: any-string "111" any-string)
      (complement (:or (:: any-string "01") (:+ "1"))))
  ```
  ```

Matches all strings that have 3 consecutive ones, but not those that end in `"01"` and not those that are ones only. These include `"1110"`, `"0001000111"` and `"0111"` but not `""`, `"11"`, `"11101"`, `"111"` and `"11111"`.

- `(:: "/*" (complement (:: any-string "*/" any-string)) "*/")`

  Matches Java/C block comments. `"/**/"`, `"/******/"`, `"/*////*/"`, `"/*asg4*/"` and so on. It does not match `"/**/*/"`, `"/* */ */"` and so on. `(:: any-string "*/" any-string)` matches any string that has a `"*/"` in is, so `(complement (:: any-string "*/" any-string))` matches any string without a `"*/"` in it.

- `(:: "/*" (:* (complement "*/")) "*/")`

  Matches any string that starts with `"/*"` and ends with `"*/"`, including `"/* */ */ */"`. `(complement "*/")` matches any string except `"*/"`. This includes `"*"` and `"/"` separately. Thus `(:* (complement "*/"))` matches `"*/"` by first matching `"*"` and then matching `"/"`. Any other string is matched directly by `(complement "*/")`. In other words, `(:* (complement "xx"))` = `any-string`. It is usually not correct to place a `:*` around a `complement`.

The `start-pos`, `end-pos`, `lexeme`, `input-port`, and `return-without-pos` forms have special meaning inside of a lexer.

The lexer raises an exception `(exn:read)` if none of the regular expressions match the input. Hint: If `(any-char` *`custom-error-behavior`*`)` is the last rule, then there will always be a match, and *`custom-error-behavior`* is executed to handle the error situation as desired, only consuming the first character from the input buffer.

In addition to returning characters, input ports can return `eof-object`s. Custom input ports can also return a `special-comment` value to indicate a non-textual comment, or return another arbitrary value (a special). The non-*`re trigger`* forms handle these cases:

- The `(eof)` rule is matched when the input port returns an `eof-object` value. If no `(eof)` rule is present, the lexer returns the symbol `'eof` when the port returns an `eof-object` value.
- The `(special-comment)` rule is matched when the input port returns a `special-comment` structure. If no `special-comment` rule is present, the lexer automatically tries to return the next token from the input port.
- The `(special)` rule is matched when the input port returns a value other than a character, `eof-object`, or `special-comment` structure. If no `(special)` rule is present, the lexer returns `(void)`.

End-of-files, specials, special-comments and special-errors cannot be parsed via a rule using an ordinary regular expression (but dropping down and manipulating the port to handle them is possible in some situations).

Since the lexer gets its source information from the port, use `port-count-lines!` to enable the tracking of line and column information. Otherwise, the line and column information will return `#f`.

When peeking from the input port raises an exception (such as by an embedded XML editor with malformed syntax), the exception can be raised before all tokens preceding the exception have been returned.

Each time the racket code for a lexer is compiled (e.g. when a `".rkt"` file containing a `lexer` form is loaded), the lexer generator is run. To avoid this overhead place the lexer into a module and compile the module to a `".zo"` bytecode file.

If the lexer can accept the empty string, a message is sent to `current-logger`. These warnings can be disabled by giving the `#:suppress-warnings` flag.

Examples:

```
> (define sample-input "( lambda (a ) (add_number a 42  ))")
; A function that partially tokenizes the sample input data
> (define (get-tokens a-lexer)
    (define p (open-input-string sample-input))
    (list (a-lexer p)
          (a-lexer p)
          (a-lexer p)
          (a-lexer p)
          (a-lexer p)))
; A lexer that uses primitive operations directly
> (define the-lexer/primitive
    (lexer
      [(eof) eof]
      ["(" 'left-paren]
      [")" 'right-paren]
      [(repetition 1 +inf.0 numeric) (string->number lexeme)]
      [(concatenation (union alphabetic #\_)
                      (repetition 0 +inf.0 (union alphabetic numeric #\_)))
       lexeme]
      ; invoke the lexer again to skip the current token
      [whitespace (the-lexer/primitive input-port)]))
> (get-tokens the-lexer/primitive)
'(left-paren "lambda" left-paren "a" right-paren)
; Another lexer that uses SRE operators but has the same function-
ality
> (require (prefix-in : parser-tools/lex-sre))
> (define the-lexer/SRE
    (lexer
      [(eof) eof]
      ["(" 'left-paren]
```

```
       [")" 'right-paren]
       [(:+ numeric) (string->number lexeme)]
       [(:: (:or alphabetic #\_) (:* (:or alphabetic numeric #\_)))
        lexeme]
       [whitespace (the-lexer/SRE input-port)]))
> (get-tokens the-lexer/SRE)
'(left-paren "lambda" left-paren "a" right-paren)
```

Changed in version 7.7.0.7 of package `parser-tools-lib`: Add `#:suppress-warnings` flag.

```
(lexer-src-pos maybe-suppress-warnings [trigger action-expr] ...)
```

Like `lexer`, but for each *action-result* produced by an *action-expr*, returns `(make-position-token` *action-result* `start-pos end-pos)` instead of simply *action-result*.

```
start-pos
```

Produces a `position` struct for the first character matched. Its use outside of a `lexer` action is a syntax error.

```
end-pos
```

Produces a `position` struct for the character after the last character in the match. Its use outside of a `lexer` action is a syntax error.

```
lexeme
```

Produces the matched string. Its use outside of a `lexer` action is a syntax error.

```
input-port
```

Produces the input port being processed, which is particularly useful for matching input with multiple lexers. Its use outside of a `lexer` action is a syntax error.

```
return-without-pos
```

Produces a function (continuation) that immediately returns its argument from the lexer. This is useful in `lexer-src-pos` to prevent the lexer from adding source information. For example:

```
(define get-token
  (lexer-src-pos
    ...
    [(comment) (get-token input-port)]
    ...))
```

would wrap the source location information for the comment around the value of the recursive call. Using `[(comment) (return-without-pos (get-token input-port))]` will cause the value of the recursive call to be returned without wrapping position around it. Its use outside of a `lexer` action is a syntax error.

```
(struct position (offset line col)
    #:extra-constructor-name make-position)
  offset : exact-positive-integer?
  line : exact-positive-integer?
  col : exact-nonnegative-integer?
```

Instances of `position` are bound to `start-pos` and `end-pos`. The `offset` field contains the offset of the character in the input. The `line` field contains the line number of the character. The `col` field contains the offset in the current line. The fields are indexed as in the position-counting of ports.

```
(struct position-token (token start-pos end-pos)
    #:extra-constructor-name make-position-token)
  token : any/c
  start-pos : position?
  end-pos : position?
```

Lexers created with `lexer-src-pos` return instances of `position-token`.

```
(file-path) → any/c
(file-path source) → void?
  source : any/c
```

A parameter that the lexer uses as the source location if it raises a `exn:fail:read` error. Setting this parameter allows DrRacket, for example, to open the file containing the error.

1.2 Lexer Abbreviations and Macros

```
(char-set string)
```

A lexer macro that matches any character in *string*.

```
any-char
```

A lexer abbreviation that matches any character.

```
any-string
```

A lexer abbreviation that matches any string.

```
nothing
```

A lexer abbreviation that matches no string.

```
alphabetic
lower-case
upper-case
title-case
numeric
symbolic
punctuation
graphic
whitespace
blank
iso-control
```

Lexer abbreviations that match `char-alphabetic?` characters, `char-lower-case?` characters, etc.

```
(define-lex-abbrev id re)
```

Defines a lexer abbreviation by associating a regular expression to be used in place of the *id* in other regular expression. The definition of name has the same scoping properties as a other syntactic binding (e.g., it can be exported from a module).

```
(define-lex-abbrevs (id re) ...)
```

Like `define-lex-abbrev`, but defines several lexer abbreviations.

```
(define-lex-trans id trans-expr)
```

Defines a lexer macro, where *trans-expr* produces a transformer procedure that takes one argument. When `(id datum ...)` appears as a regular expression, it is replaced with the result of applying the transformer to the expression.

1.3 Lexer SRE Operators

```
(require parser-tools/lex-sre)
```

package: `parser-tools-lib`

```
(* re ...)
```

Repetition of *re* sequence 0 or more times.

```
(+ re ...)
```

Repetition of *re* sequence 1 or more times.

```
(? re ...)
```

Zero or one occurrence of *re* sequence.

```
(= n re ...)
```

Exactly *n* occurrences of *re* sequence, where *n* must be a literal exact, non-negative number.

```
(>= n re ...)
```

At least *n* occurrences of *re* sequence, where *n* must be a literal exact, non-negative number.

```
(** n m re ...)
```

Between *n* and *m* (inclusive) occurrences of *re* sequence, where *n* must be a literal exact, non-negative number, and *m* must be literally either `#f`, `+inf.0`, or an exact, non-negative number; a `#f` value for *m* is the same as `+inf.0`.

```
(or re ...)
```

Same as `(union re ...)`.

```
(: re ...)
(seq re ...)
```

Both forms concatenate the *re*s.

```
(& re ...)
```

Intersects the *re*s.

```
(- re ...)
```

The set difference of the *re*s.

```
(~ re ...)
```

Character-set complement, which each *re* must match exactly one character.

```
(/ char-or-string ...)
```

Character ranges, matching characters between successive pairs of characters.

1.4 Lexer Legacy Operators

```
(require parser-tools/lex-plt-v200)
```
package: parser-tools-lib

The parser-tools/lex-plt-v200 module re-exports *, +, ?, and & from parser-tools/lex-sre. It also re-exports :or as :, :: as @, :~ as ^, and :/ as -.

```
(epsilon)
```

A lexer macro that matches an empty sequence.

```
(~ re ...)
```

The same as (complement re ...).

1.5 Tokens

Each *action-expr* in a lexer form can produce any kind of value, but for many purposes, producing a *token* value is useful. Tokens are usually necessary for inter-operating with a parser generated by parser-tools/yacc or parser-tools/cfg-parser, but tokens may not be the right choice when using lexer in other situations.

Examples:

```
> (define-tokens basic-tokens (NUM ID))
> (define-empty-tokens punct-tokens (LPAREN RPAREN EOF))
```

```
> (define the-lexer/tokens
    (lexer
      [(eof) (token-EOF)]
      ["(" (token-LPAREN)]
      [")" (token-RPAREN)]
      [(:+ numeric) (token-NUM (string->number lexeme))]
      [(:: (:or alphabetic #\_)
           (:* (:or alphabetic numeric #\_)))
       (token-ID (string->symbol lexeme))]
      [whitespace (the-lexer/tokens input-port)]))
; Use get-tokens defined in §1.1 ‘‘Creating a Lexer’’
> (get-tokens the-lexer/tokens)
(list 'LPAREN (token 'ID 'lambda) 'LPAREN (token 'ID 'a) 'RPAREN)
```

```
(define-tokens group-id (token-id ...))
```

Binds *group-id* to the group of tokens being defined. For each *token-id*, a function token-*token-id* is created that takes any value and puts it in a token record specific to *token-id*. A token can be inspected using token-name and token-value.

A token cannot be named error, since error has a special use in the parser.

```
(define-empty-tokens group-id (token-id ...))
```

Like define-tokens, but each token constructor token-*token-id* takes no arguments and returns (quote *token-id*).

```
(token-name t) → symbol?
  t : (or/c token? symbol?)
```

Returns the name of a token that is represented either by a symbol or a token structure.

```
(token-value t) → any/c
  t : (or/c token? symbol?)
```

Returns the value of a token that is represented either by a symbol or a token structure, returning #f for a symbol token.

```
(token? v) → boolean?
  v : any/c
```

Returns #t if val is a token structure, #f otherwise.

2 LALR(1) Parsers

```
(require parser-tools/yacc)
```
package: parser-tools-lib

```
(parser clause ...)

    clause = (grammar (non-terminal-id
                       ((grammar-id ...) maybe-prec expr)
                       ...)
                      ...)
           | (tokens group-id ...)
           | (start non-terminal-id ...)
           | (end token-id ...)
           | (error expr)
           | (precs (assoc token-id ...) ...)
           | (src-pos)
           | (suppress)
           | (expected-SR-conflicts num)
           | (expected-RR-conflicts num)
           | (debug filename)
           | (yacc-output filename)

maybe-prec =
           | (prec token-id)

     assoc = left
           | right
           | nonassoc
```

Creates a parser. The clauses may be in any order, as long as there are no duplicates and all non-*OPTIONAL* declarations are present:

- `(grammar (non-terminal-id ((grammar-id ...) maybe-prec expr) ...) ...)`

  Declares the grammar to be parsed. Each *grammar-id* can be a *token-id* from a *group-id* named in a `tokens` declaration, or it can be a *non-terminal-id* declared in the `grammar` declaration. The optional `prec` declaration works with the `precs` declaration. The *expr* is a "semantic action," which is evaluated when the input is found to match its corresponding production.

  Each action is Racket code that has the same scope as its parser's definition, except that the variables `$1`, ..., `$`*i* are bound, where *i* is the number of *grammar-id*s in the corresponding production. If the k^{th} grammar symbol on the right of the production

is a non-terminal, $k is bound to the result of its action; if it is a terminal, $k is bound to the value stored in the token. If the src-pos option is present in the parser, then variables $1-start-pos, ..., $i-start-pos and $1-end-pos, ..., $i-end-pos and are also available, and they refer to the position structures corresponding to the start and end of the corresponding grammar-symbol. Grammar symbols defined as empty-tokens have no $k associated, but do have $k-start-pos and $k-end-pos. Also $n-start-pos and $n-end-pos are bound to the largest start and end positions, (i.e., $i-start-pos and $i-end-pos).

An *error production* can be defined by providing a production of the form (error α), where α is a string of grammar symbols, possibly empty.

All of the productions for a given non-terminal must be grouped with it. That is, no *non-terminal-id* may appear twice on the left hand side in a parser.

- (tokens *group-id* ...)

  Declares that all of the tokens defined in each *group-id*—as bound by define-tokens or define-empty-tokens—can be used by the parser in the grammar declaration.

- (start *non-terminal-id* ...)

  Declares a list of starting non-terminals for the grammar.

- (end *token-id* ...)

  Specifies a set of tokens from which some member must follow any valid parse. For example, an EOF token would be specified for a parser that parses entire files and a newline token for a parser that parses entire lines individually.

- (error *expr*)

  The *expr* should evaluate to a function which will be executed for its side-effect whenever the parser encounters an error.

  If the src-pos declaration is present, the function should accept 5 arguments:

  ```
  (lambda (tok-ok? tok-name tok-value start-pos end-pos)
    ....)
  ```

  Otherwise it should accept 3:

  ```
  (lambda (tok-ok? tok-name tok-value)
    ....)
  ```

  The first argument will be #f if and only if the error is that an invalid token was received. The second and third arguments will be the name and the value of the token at which the error was detected. The fourth and fifth arguments, if present, provide the source positions of that token.

  In both cases, the function is allowed to accept an additional keyword argument named #:stack. This argument is a representation of the parsing automaton's stack. This can, for example, be used to generate context-sensitive error messages as described in Generating LR syntax error messages from examples, by Clinton L. Jeffrey.

- `(precs (assoc token-id ...) ...)` *OPTIONAL*

  Precedence declarations to resolve shift/reduce and reduce/reduce conflicts as in `yacc`/`bison`. An *assoc* must be one of *left*, *right* or *nonassoc*. States with multiple shift/reduce or reduce/reduce conflicts (or some combination thereof) are not resolved with precedence.

- `(src-pos)` *OPTIONAL*

  Causes the generated parser to expect input in the form `(make-position-token token start-pos end-pos)` instead of simply *token*. Include this option when using the parser with a lexer generated with `lexer-src-pos`.

- `(debug filename)` *OPTIONAL*

  Causes the parser generator to write the LALR table to the file named *filename* (unless the file exists), where *filename* is a literal string. Additionally, if a debug file is specified, when a running generated parser encounters a parse error on some input file, after the user specified error expression returns, the complete parse stack is printed to assist in debugging the grammar of that particular parser. The numbers in the stack printout correspond to the state numbers in the LALR table file.

- `(yacc-output filename)` *OPTIONAL*

  Causes the parser generator to write a grammar file in approximately the syntax of `yacc`/`bison`. The file might not be a valid `yacc` file, because the Racket grammar can use symbols that are invalid in C.

- `(suppress)` *OPTIONAL*

  Causes the parser generator not to report shift/reduce or reduce/reduce conflicts.

- `(expected-SR-conflicts n)` *OPTIONAL*

  Causes the parser generator to expect exactly *num* shift/reduce conflicts, where *num* must be a literal number. The `suppress` option overrides this option.

- `(expected-RR-conflicts n)` *OPTIONAL*

  Causes the parser generator to expect exactly *num* reduce/reduce conflicts, where *num* must be a literal number. The `suppress` option overrides this option.

The result of a `parser` expression with one `start` non-terminal is a function, *parse*, that takes one argument. This argument must be a zero argument function, *gen*, that produces successive tokens of the input each time it is called. If desired, the *gen* may return symbols instead of tokens, and the parser will treat symbols as tokens of the corresponding name (with `#f` as a value, so it is usual to return symbols only in the case of empty tokens). The *parse* function returns the value associated with the parse tree by the semantic actions. If the parser encounters an error, after invoking the supplied error function, it will try to use error productions to continue parsing. If it cannot, it raises `exn:fail:read`.

If multiple non-terminals are provided in `start`, the `parser` expression produces a list of parsing functions, one for each non-terminal in the same order. Each parsing function is like the result of a parser expression with only one `start` non-terminal.

Each time the Racket code for a `parser` is compiled (e.g. when a `".rkt"` file containing a `parser` form is loaded), the parser generator is run. To avoid this overhead place the parser into a module and compile the module to a `".zo"` bytecode file.

Examples:

```
; Use the lexer and tokens from §1.5 "Tokens"
; and the sample input from §1.1 "Creating a Lexer"
> (define the-parser
    (parser
      [start expr]
      [end EOF]
      [error void]
      [tokens basic-tokens punct-tokens]
      [grammar
       [expr [(LPAREN exprs RPAREN) $2]
             [(NUM) $1]
             [(ID) $1]]
       [exprs [() '()]
              [(expr exprs) (cons $1 $2)]]]))
> (define p (open-input-string sample-input))
> (the-parser (λ () (the-lexer/tokens p)))
'(lambda (a) (add_number a 42))
```

3 Context-Free Parsers

```
(require parser-tools/cfg-parser)
            package: parser-tools-lib
```

The parser-tools/cfg-parser library provides a parser generator that is an alternative to that of parser-tools/yacc.

```
(cfg-parser clause ...)

clause = (grammar (non-terminal-id
                   ((grammar-id ...) maybe-prec expr)
                   ...)
                  ...)
       | (tokens group-id ...)
       | (start non-terminal-id ...)
       | (end token-id ...)
       | (error expr)
       | (src-pos)
```

Creates a parser similar to that of `parser`. Unlike `parser`, `cfg-parser`, can consume arbitrary and potentially ambiguous context-free grammars. Its interface is a subset of parser-tools/yacc, with the following differences:

- `(start non-terminal-id)`

  Unlike `parser`, `cfg-parser` only allows for a single non-terminal-id.

- The `cfg-parser` form does not support the `precs`, `suppress`, `expected-SR-conflicts`, `expected-RR-conflicts`, `debug`, or `yacc-output` options of `parser`.

4 Converting `yacc` or `bison` Grammars

```
(require parser-tools/yacc-to-scheme)
```
package: `parser-tools-lib`

```
(trans file) → any/c
  file : path-string?
```

Reads a C `yacc`/`bison` grammar from *file* and produces an s-expression that represents a Racket parser for use with `parser`.

This function is intended to assist in the manual conversion of grammars for use with `parser`, and not as a fully automatic conversion tool. It is not entirely robust. For example, if the C actions in the original grammar have nested blocks, the tool will fail.

Annotated examples are in the `"examples"` subdirectory of the `"parser-tools"` collection.

Index