1 Lexers

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

1.1 Creating a Lexer

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

```
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 following binding have special meaning inside of a lexer action:

- `start-pos` — a `position` struct for the first character matched.
- `end-pos` — a `position` struct for the character after the last character in the match.
- `lexeme` — the matched string.
- `input-port` — the input-port being processed (this is useful for matching input with multiple lexers).
- `(return-without-pos x)` is a function (continuation) that immediately returns the value of `x` from the lexer. This useful in a src-pos lexer 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.

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-objects`. 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 the-lexer
    (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 input-port)]))
> (define s (open-input-string "( lambda (a ) (add_number a 42))"))
> (list (the-lexer s) (the-lexer s) (the-lexer s) (the-lexer s) (the-lexer s))
'(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***] ...)** syntax

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` syntax

`end-pos` syntax

`lexeme` syntax

`input-port` syntax

`return-without-pos` syntax

Use of these names outside of a `lexer` action is a syntax error.

```
(struct position (offset line col)                                  struct
    #: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.

```
(struct position-token (token start-pos end-pos)                    struct
    #: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                                                 parameter
(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)` syntax

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

`any-char` syntax

A lexer abbreviation that matches any character.

---

any-string syntax

A lexer abbreviation that matches any string.

---

nothing syntax

A lexer abbreviation that matches no string.

---

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

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

(**define-lex-abbrev** *id re*) syntax

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*) ...) syntax

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

(**define-lex-trans** *id trans-expr*) syntax

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* ...) syntax

Repetition of *re* sequence 0 or more times.

(+ *re* ...) syntax

Repetition of *re* sequence 1 or more times.

(? *re* ...) syntax

Zero or one occurrence of *re* sequence.

(= *n* *re* ...) syntax

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

(>= *n* *re* ...) syntax

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

(** *n* *m* *re* ...) syntax

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* ...) syntax

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

(: *re* ...) syntax
(seq *re* ...) syntax

Both forms concatenate the *re*s.

(& *re* ...) syntax

Intersects the *re*s.

(- *re* ...) syntax

The set difference of the *re*s.

`(~ re ...)` syntax

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

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

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)` syntax

A lexer macro that matches an empty sequence.

`(~ re ...)` syntax

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 (number id))
> (define-empty-tokens punct-tokens (left-paren right-paren the-end))
> (define the-lexer
    (lexer
     [(eof) (token-the-end)]
     ["(" (token-left-paren)]
     [")" (token-right-paren)]
     [(repetition 1 +inf.0 numeric) (token-number (string->number lexeme))]
     [(concatenation (union alphabetic #\_)
                     (repetition 0 +inf.0 (union alphabetic numeric #\_)))
      (token-id (string->symbol lexeme))]
```

```
      ; invoke the lexer again to skip the current token
      [whitespace (the-lexer input-port)]))
> (define s (open-input-string "( lambda (a ) (add_number a 42))"))
> (list (the-lexer s) (the-lexer s) (the-lexer s) (the-lexer s) (the-lexer s))
(list 'left-paren (token 'id 'lambda) 'left-paren (token 'id 'a) 'right-paren)
```

(**define-tokens** *group-id* (*token-id* ...)) syntax

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*. The token value is inspected using *token-id* and `token-value`.

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

(**define-empty-tokens** *group-id* (*token-id* ...)) syntax

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

(**token-name** *t*) → `symbol?` procedure

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` procedure

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?` procedure

v : `any/c`

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