

# Lexical Analysis with Racket

Matt Might
University of Utah
matt.might.net

▼ Parser Tools: lex and yacc-style Parsing
1 Lexers
2 LALR(1) Parsers
3 Context-Free Parsers
4 Converting yacc or bison Grammars
Index

▶ 1 Lexers

ON THIS PAGE:

1.1 Creating a Lexer
lexer
lexer-src-pos
start-pos
end-pos
lexeme
input-port

# 1 Lexers

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

## 1.1 Creating a Lexer

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

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

    re = id
       | string
       | character
       | (repetition lo hi re)
       | (union re ...)
```

# Lexical analysis in Racket



In compilers and interpreters, lexers transform source code (a sequence of characters) into a sequence of tokens.

By stripping out insignificant characters (often whitespace and comments), the lexer is the first increase in the level of abstraction.

For instance, a lexer might transform:

```
  3 + (x * 10)
```

into:

```
(NUM 3)
(OP +)
(LPAR)
(IDENT x)
(OP *)
(NUM 10)
(RPAR)
```

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

```
(lexer [trigger action] ...)
```

```
(define my-lexer (lexer [trigger action] ...))
```

# Triggers

$$
\begin{aligned}
\mathit{trigger} ::= {} & \mathit{re} \\
& \mid \texttt{(eof)}
\end{aligned}
$$

```
re ::= id
     |  string
     |  character
     |  (concatenation re …)
     |  (union re ...)
     |  (repetition lo hi re)
```

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

# Examples

foo

"foo"

```
foo|bar
```

```
(union "foo" "bar")
```

(foo|bar)*

```
(repetition 0 +inf.0 (union "foo" "bar"))
```

# Actions

lexeme

start-pos  end-pos

input-port

# Simple lexer

```
(define basic-printing-lexer
  (lexer
   [(repetition 1 +inf.0 (char-range #\a #\z))

    (begin (display "found an id: ")
           (display lexeme)
           (newline))]

[(union #\space #\newline)

 (basic-printing-lexer input-port)]))
```

# I/O

(open-input-string *string*)

(open-input-file *file-name*)

# Too verbose?

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

```
(* re ...)
(+ re ...)
(? re ...)
(= n re ...)
(>= n re ...)
(** n m re ...)
(or re ...)
(: re ...)
(seq re ...)
(& re ...)
(- re ...)
(~ re ...)
(/ char-or-string ...)
```

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

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

```
(* re ...)
(+ re ...)
(? re ...)
(= n re ...)
(>= n re ...)
(** n m re ...)
(or re ...)
(: re ...)
(seq re ...)
(& re ...)
(- re ...)
(~ re ...)
(/ char-or-string ...)
```

```
(:* re ...)
(:+ re ...)
(:? re ...)
(:= n re ...)
(:>= n re ...)
(:** n m re ...)
(:or re ...)
(:: re ...)
(:seq re ...)
(:& re ...)
(:- re ...)
(:~ re ...)
(:/ char-or-string ...)
```

foo

"foo"

```
foo|bar
```

```
(:or "foo" "bar")
```

(foo|bar)*

```
(:* (:or "foo" "bar"))
```

# Abbreviations

(define-lex-abbrev *id re*)

# Examples

```
integer          ::=  decimalinteger |
                      octinteger |
                      hexinteger | bininteger
decimalinteger   ::=  nonzerodigit digit* | "0"+
nonzerodigit     ::=  "1"..."9"
digit            ::=  "0"..."9"
octinteger       ::=  "0" ("o" | "O") octdigit+
hexinteger       ::=  "0" ("x" | "X") hexdigit+
bininteger       ::=  "0" ("b" | "B") bindigit+
octdigit         ::=  "0"..."7"
hexdigit         ::=  digit | "a"..."f" | "A"..."F"
bindigit         ::=  "0" | "1"
```

```
(define-lex-abbrev nonzerodigit (char-range #\1 #\9))

(define-lex-abbrev digit (char-range #\0 #\9))

(define-lex-abbrev octdigit (char-range #\0 #\7))


(define-lex-abbrev hexdigit (union digit
                                   (char-range #\a #\f)
                                   (char-range #\A #\F)))

(define-lex-abbrev bindigit (union #\0 #\1))

(define-lex-abbrev octinteger (:: #\0 (:or #\o #\O) (:+ octdigit)))

(define-lex-abbrev hexinteger (:: #\0 (:or #\x #\X) (:+ hexdigit)))

(define-lex-abbrev bininteger (:: #\0 (:or #\b #\B) (:+ bindigit)))

(define-lex-abbrev decimalinteger (:or (:: nonzerodigit (:* digit)) (:+ #\0)))
```

# Questions?