Lexical Analysis

Matthew Might
University of Utah
matt.might.net

Administrative items

- First exam: Tuesday
- Project out: Today

“Exam”

- Topics: Python + Racket
- Worth 5% of exam score
- Laptops are allowed
- Internet banned (except matt.might.net)
- Phones / Tablets not allowed
- Can take exam on your laptop!

You will still fail.

Probably.

Today

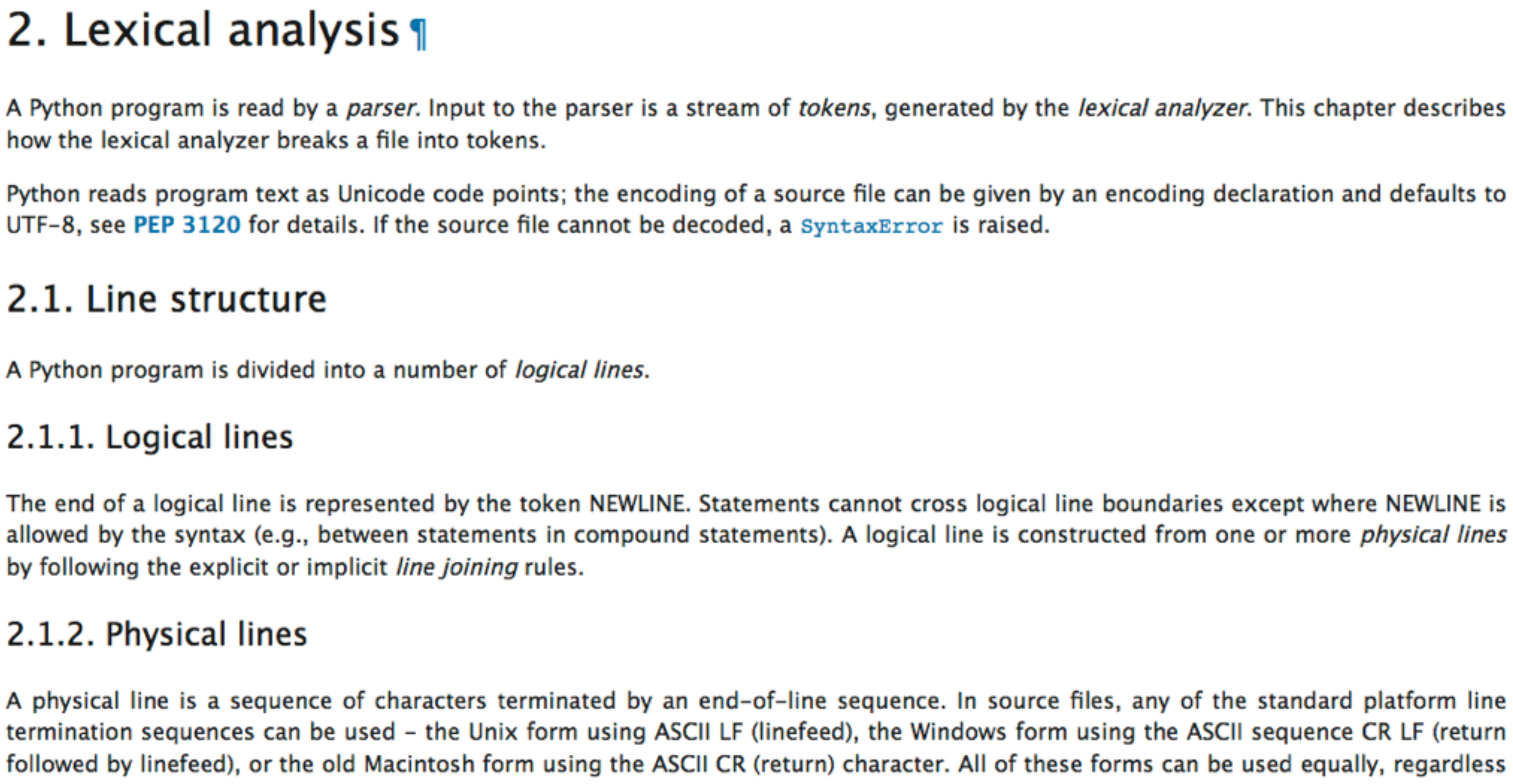
- What is lexical analysis?
- Lexical specifications
- Formal languages
- Regular languages
- Racket lexer technology

Syntax & Semantics

Syntax

Lexical Specification

Formal Grammar



2. Lexical analysis ¶

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8, see **PEP 3120** for details. If the source file cannot be decoded, a `SyntaxError` is raised.

2.1. Line structure

A Python program is divided into a number of *logical lines*.

2.1.1. Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the explicit or implicit *line joining* rules.

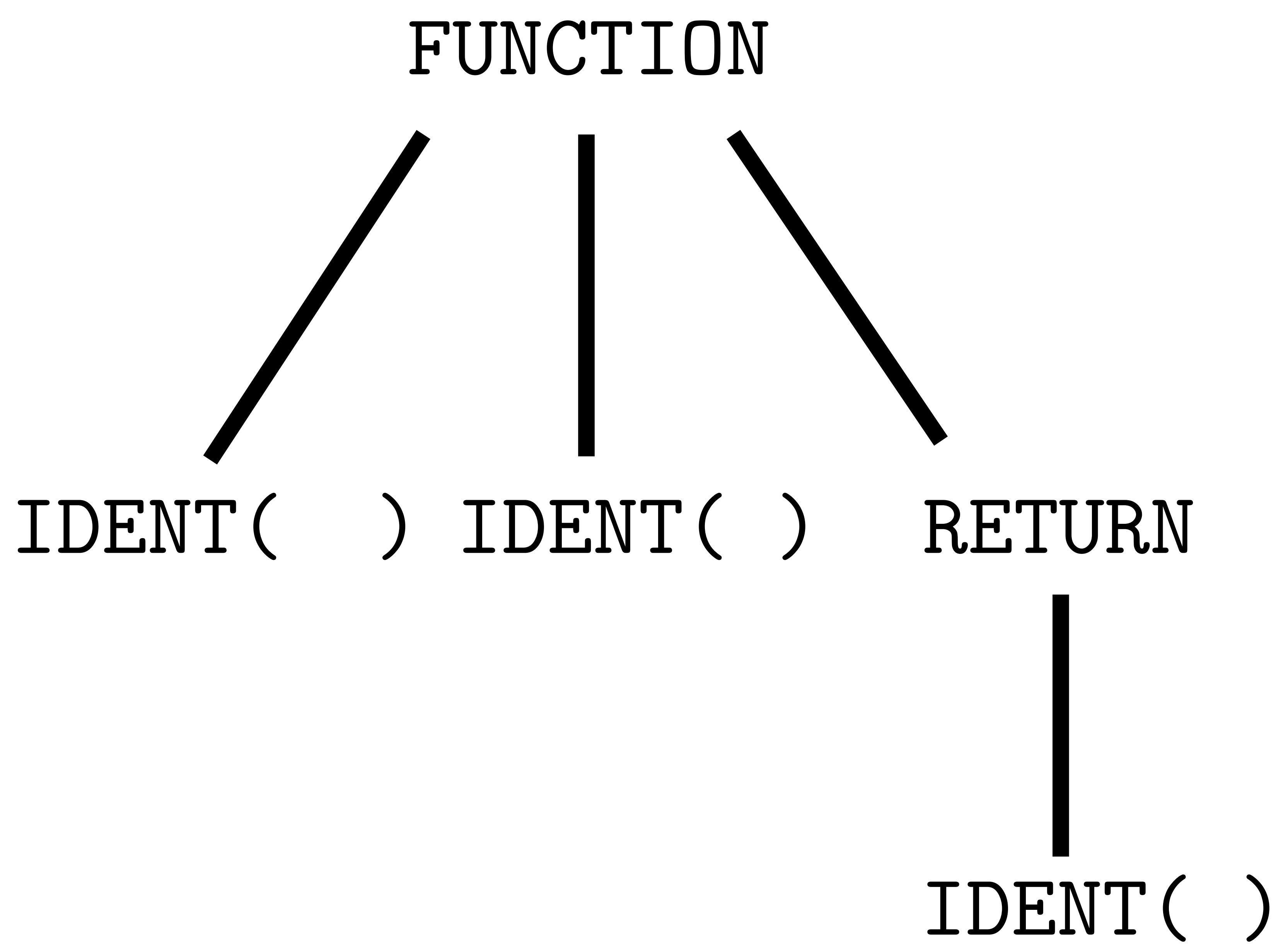
2.1.2. Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used – the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless

https://docs.python.org/3/reference/lexical_analysis.html

```
function id(x)
{
 return x ;  // comment
}
```

```
FUNCTION

IDENT(  )

LPAR

IDENT( )

RPAR

LBRACE

RETURN

IDENT( )

SEMI

RBRACE
```



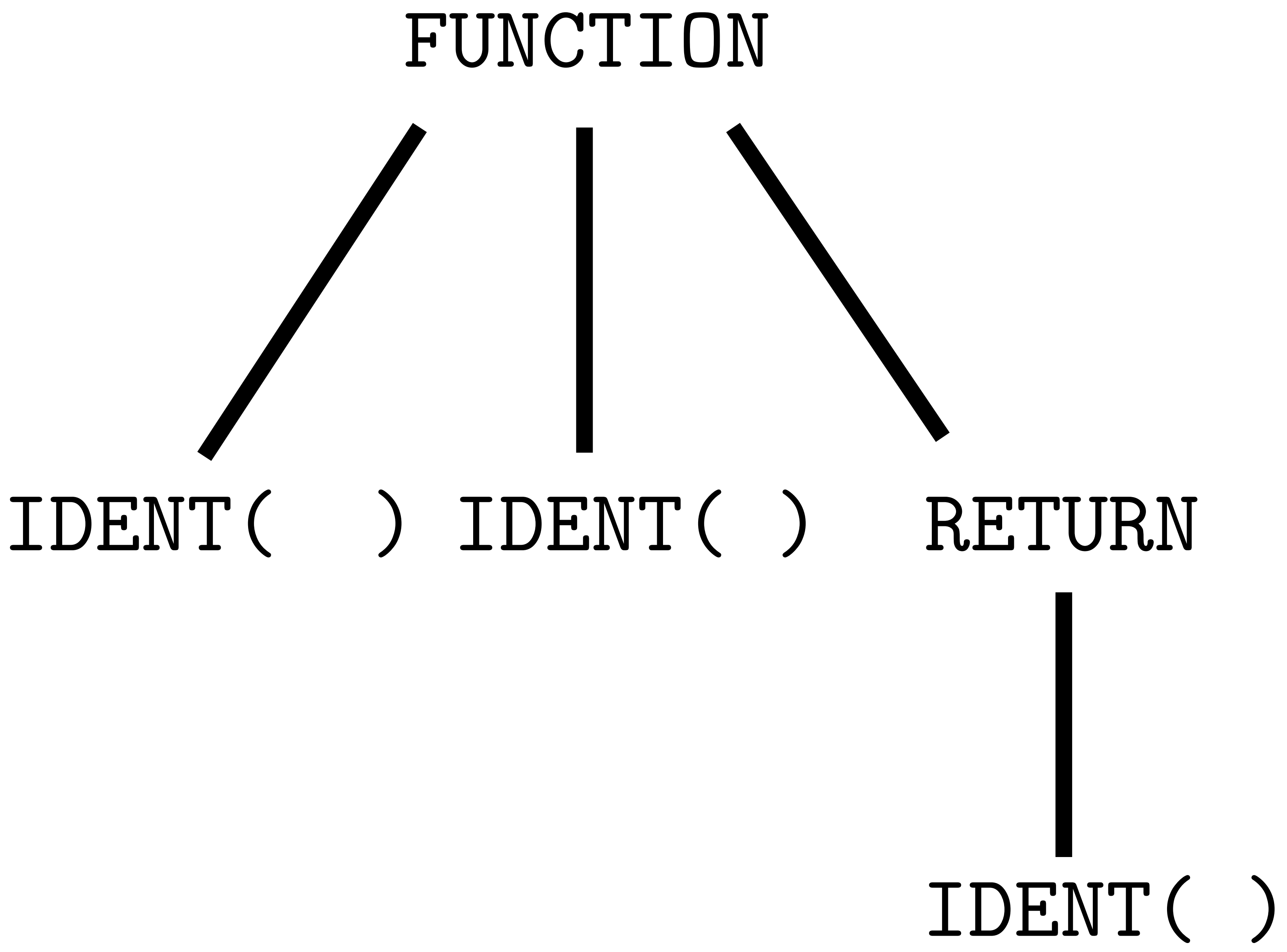
LPAR

RPAR

LBRACE

SEMI

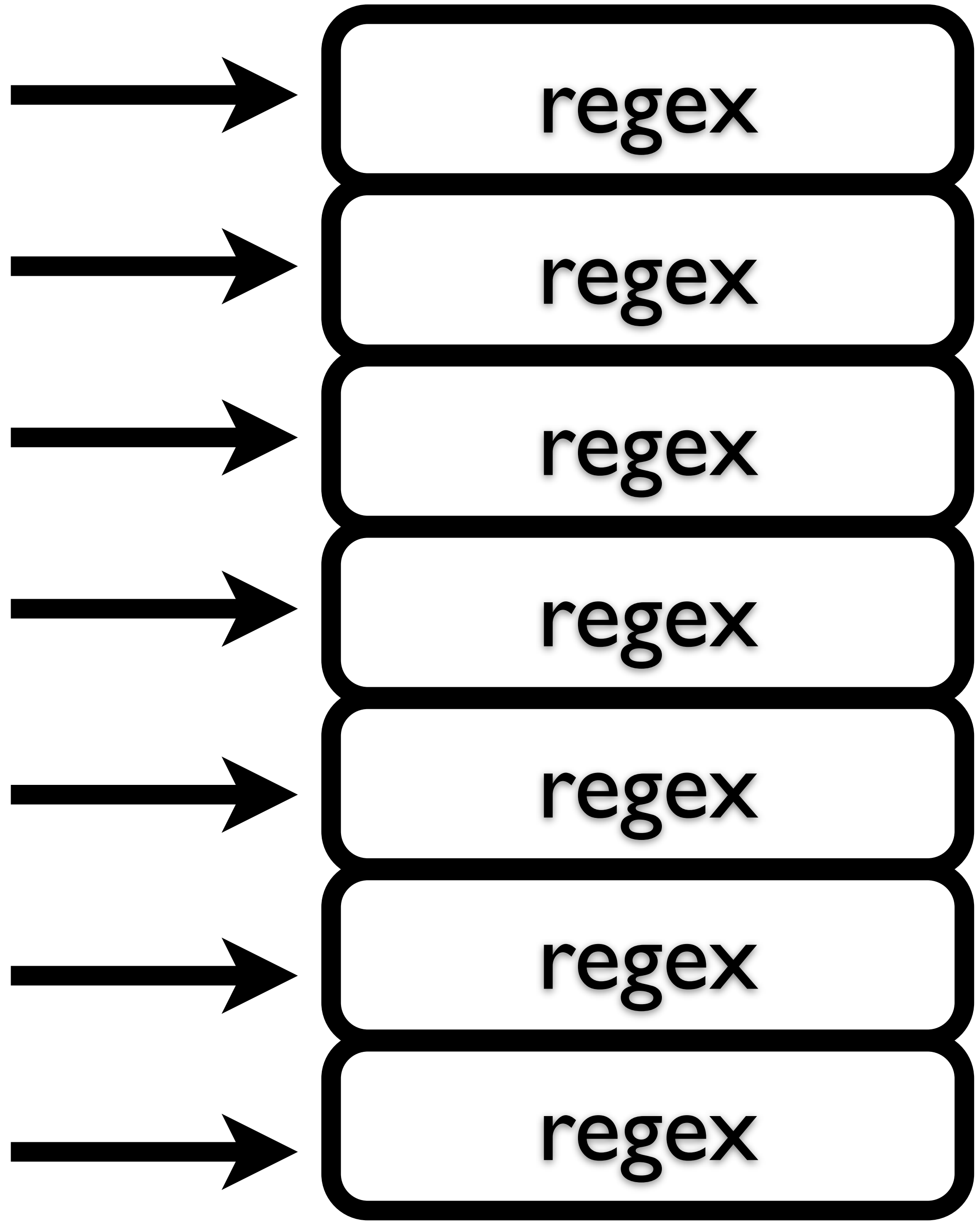
RBRACE

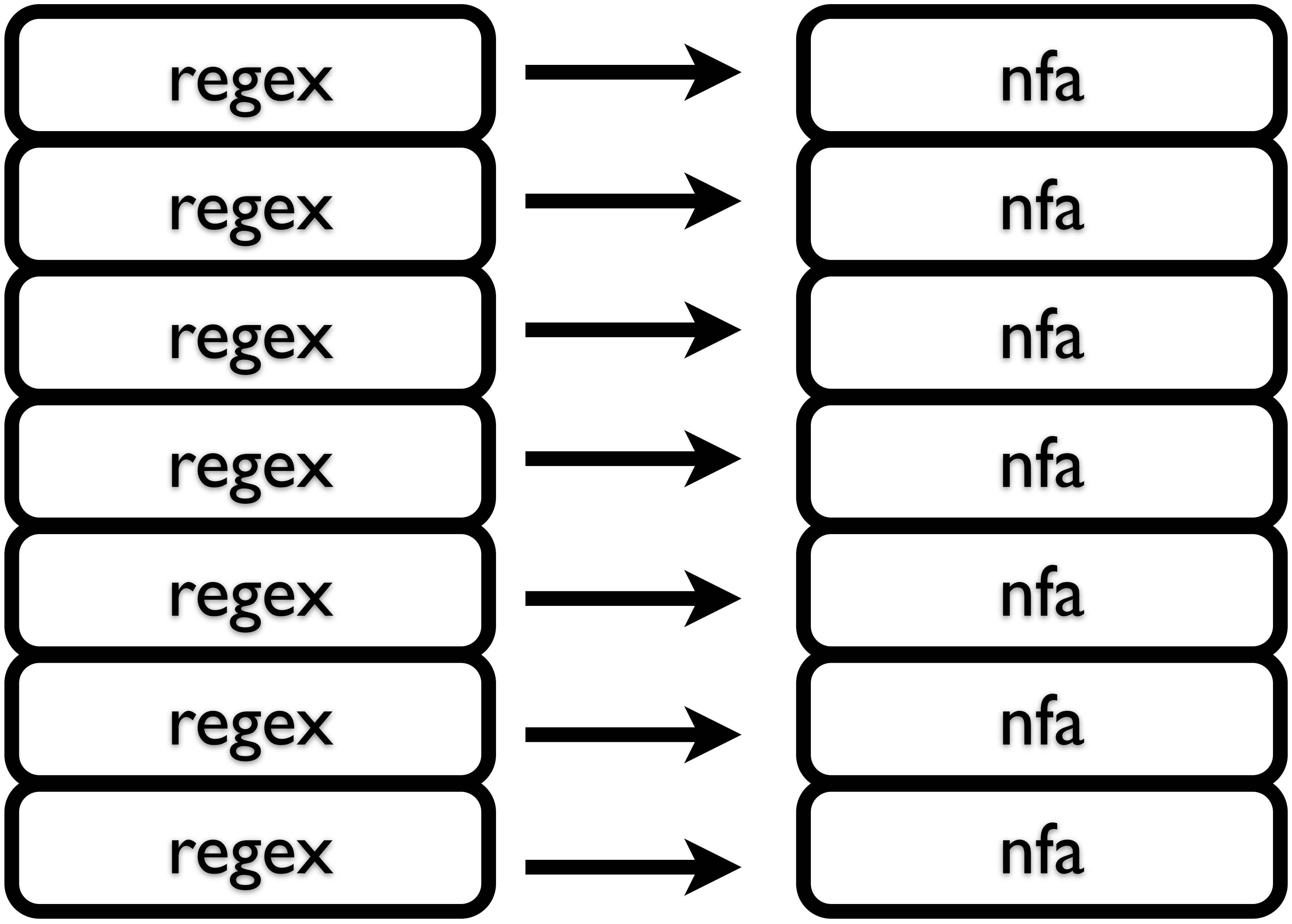

FUNCTION
IDENT( )
IDENT( )
RETURN
IDENT( )

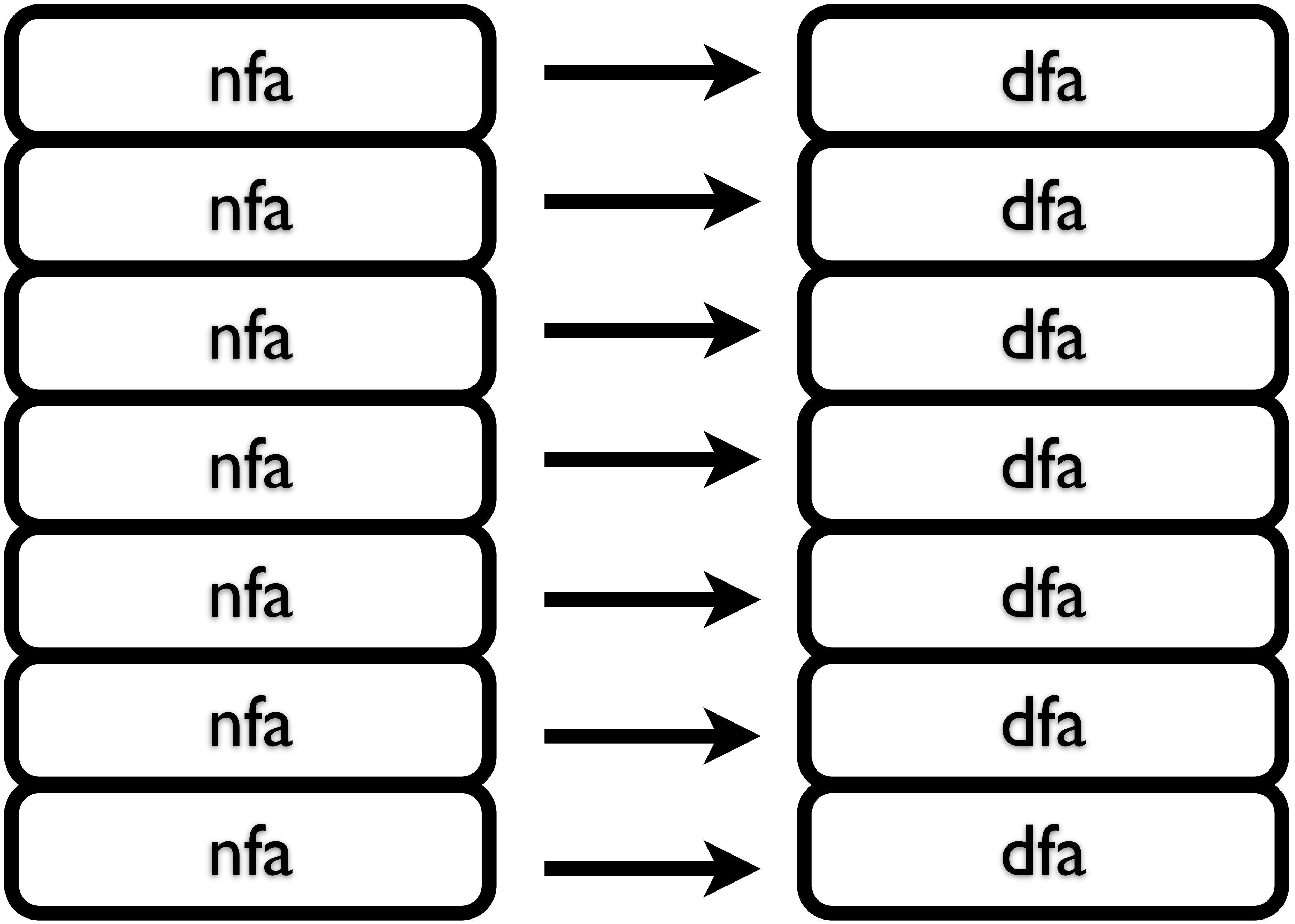
Lexical specification

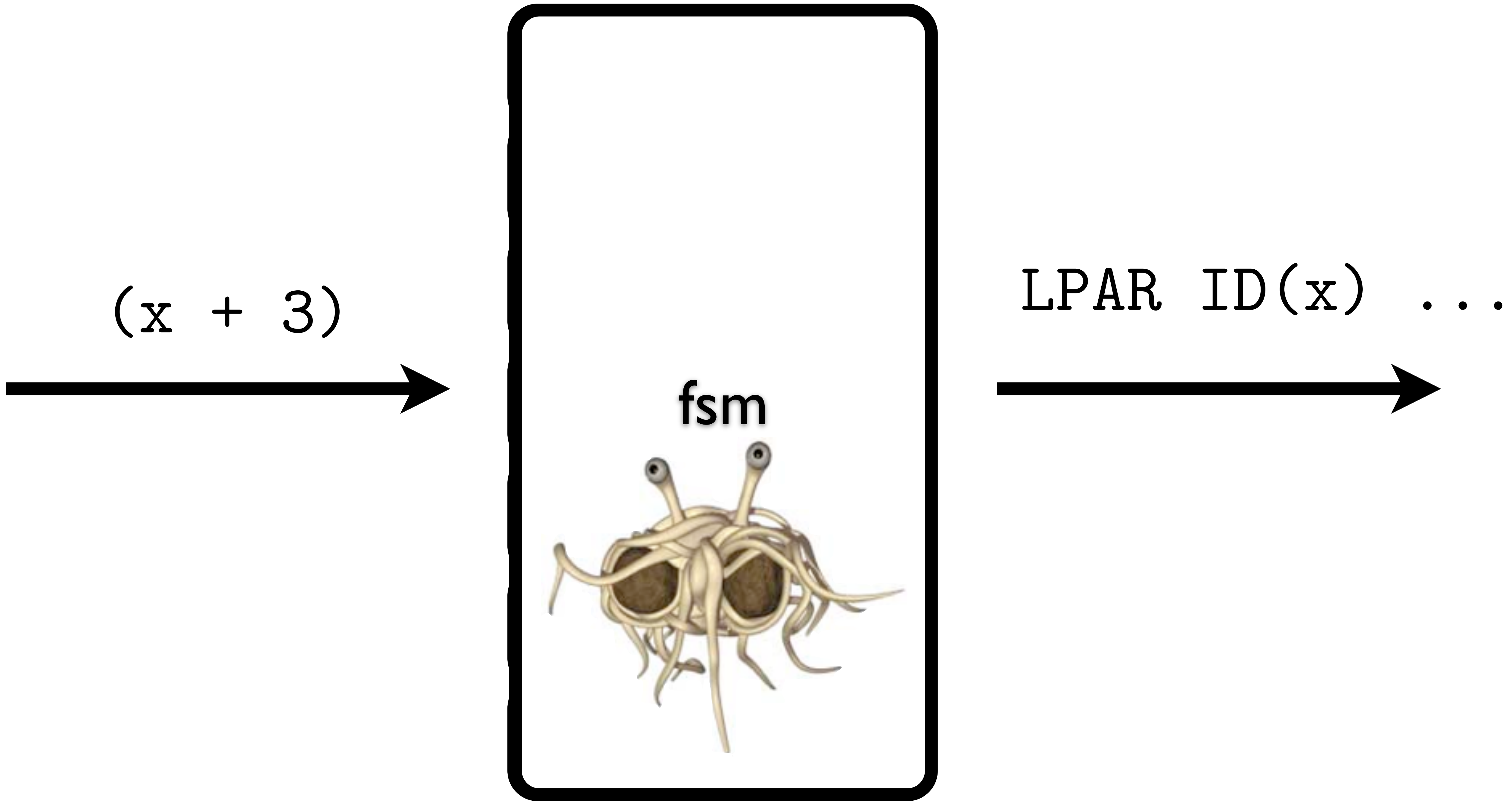
- Token types
- Whitespace
- Keywords
- Operators
- Comments
- Identifiers
- Punctuation

- Token types
- Whitespace
- Keywords
- Operators
- Comments
- Identifiers
- Punctuation




regex
regex
regex
regex
regex
regex
regex
nfa
nfa
nfa
nfa
nfa
nfa
nfa

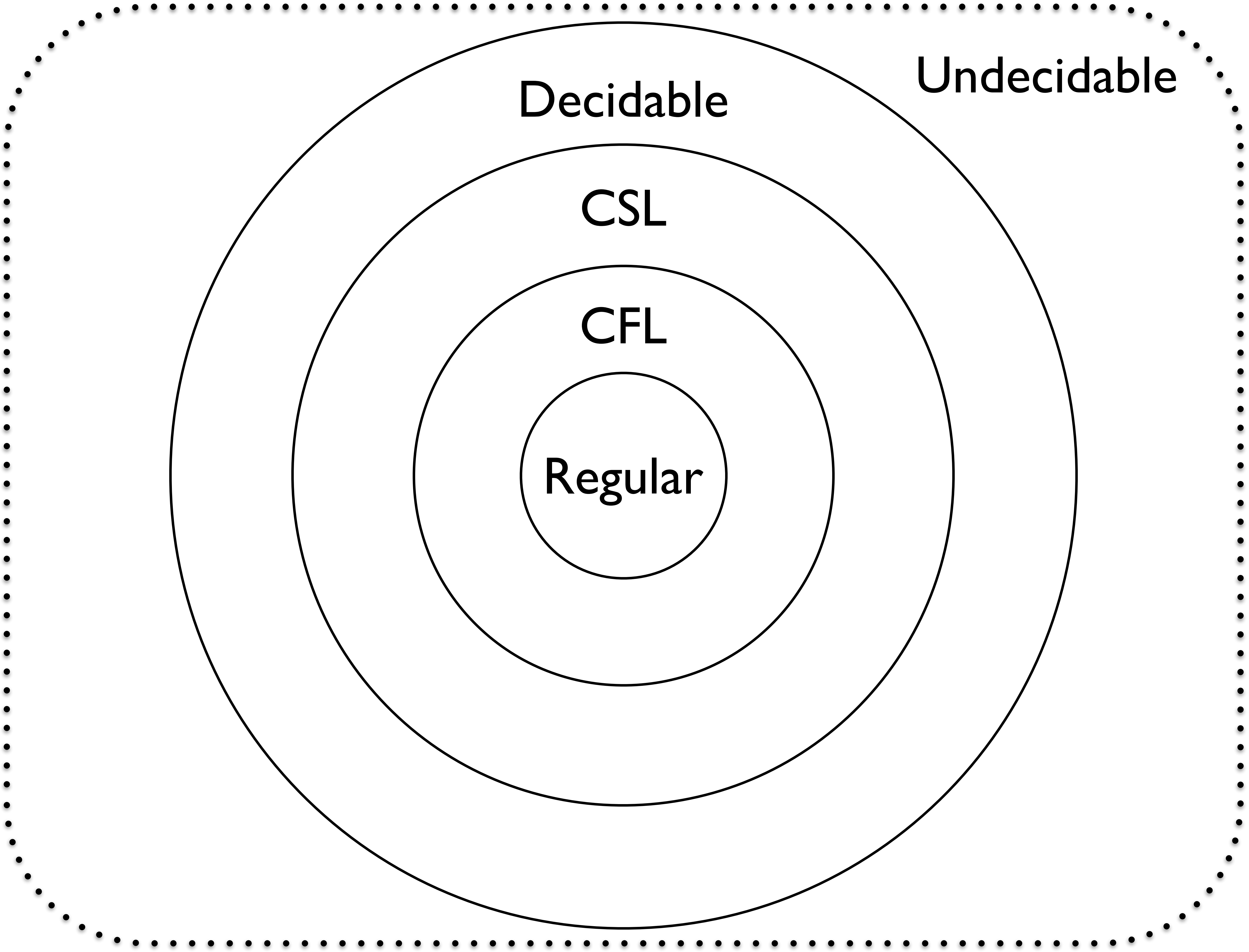

nfa
nfa
nfa
nfa
nfa
nfa
nfa
dfa
dfa
dfa
dfa
dfa
dfa
dfa


(x + 3)
fsm
LPAR ID(x) ...

Writing lexical spec?

You need regular languages.

You need formal languages.


Undecidable
Decidable
CSL
CFL
Regular

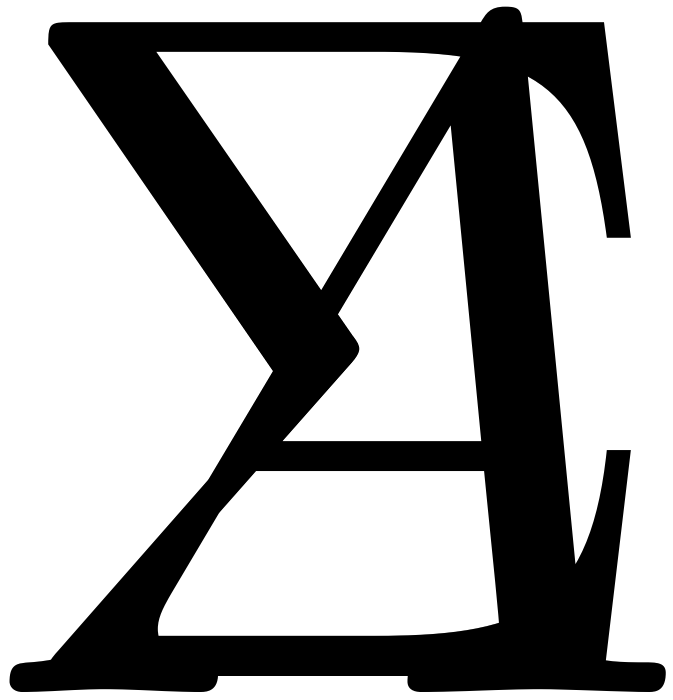
A formal language is a set of strings over an alphabet.

A string is a sequence of characters over an alphabet.

An alphabet is a set of characters.

Notation

Alphabets



Languages

$$L$$

Languages

$$L \subseteq A^*$$

Examples

$$A = \{0, 1\}$$

$$A = \{\mathtt{a}, \mathtt{b}\}$$

$$L_1 = \{\texttt{bab}, \texttt{abba}\}$$

$$L_2 = \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$$

Atomic languages

$$\emptyset$$

$$\emptyset = \{\}$$

{"" ""}

$$\epsilon = \{\text{""}\}$$

$$\epsilon = ""$$

(Sometimes!)

Primitive languages

C

$$c = \{"c"\}$$

Operations

Concatenation

$$L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

Exercise

$$L_1 = \{\texttt{a},\texttt{b}\}$$

$$L_2 = \{\texttt{c},\texttt{d}\}$$

Exercise

$$L_1 \cdot L_2 = \{\texttt{ac},\texttt{ad},\texttt{bc},\texttt{bd}\}$$

Union

$$L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}$$

Exercise

$$L_1 = \{\texttt{a}, \texttt{b}\}$$

$$L_2 = \{\texttt{c}, \texttt{d}\}$$

Exercise

$$L_1 \cup L_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$$

Exponentiation

$$L^n = \{w_1 \ldots w_n : w_i \in L\}$$

Exponentiation

$$L^0 = \epsilon$$

Exercise

$$\{\texttt{a},\texttt{b}\}^3$$

$$\{\texttt{aaa}, \texttt{aab}, \texttt{aba}, \texttt{abb}, \texttt{baa}, \texttt{bab}, \texttt{bba}, \texttt{bbb}\}$$

Kleene star

$$L^\star = \bigcup_{n=0}^{\infty} L^n$$

Exercise

$$\{\mathtt{a}, \mathtt{b}\}^*$$

Concatenation,

union,

Kleene star.

Regular Languages

More regular operations!

Option

$$L^{?} \equiv L \cup \{\epsilon\}$$

Kleene plus

$$L^+ = \bigcup_{n=1}^{\infty} L^n$$

Intersection

$$L_1 \cap L_2 = \{w : w \in L_1 \text{ and } w \in L_2\}$$

Difference

$$L_1 - L_2 = \{w : w \in L_1 \text{ and } w \notin L_2\}$$

Complement

$$\overline{L} = \{w : w \notin L\}$$

Reversal

$$L^R = \{\langle a_n, \ldots, a_1 \rangle : \langle a_1, \ldots, a_n \rangle \in L\}$$

Prefix

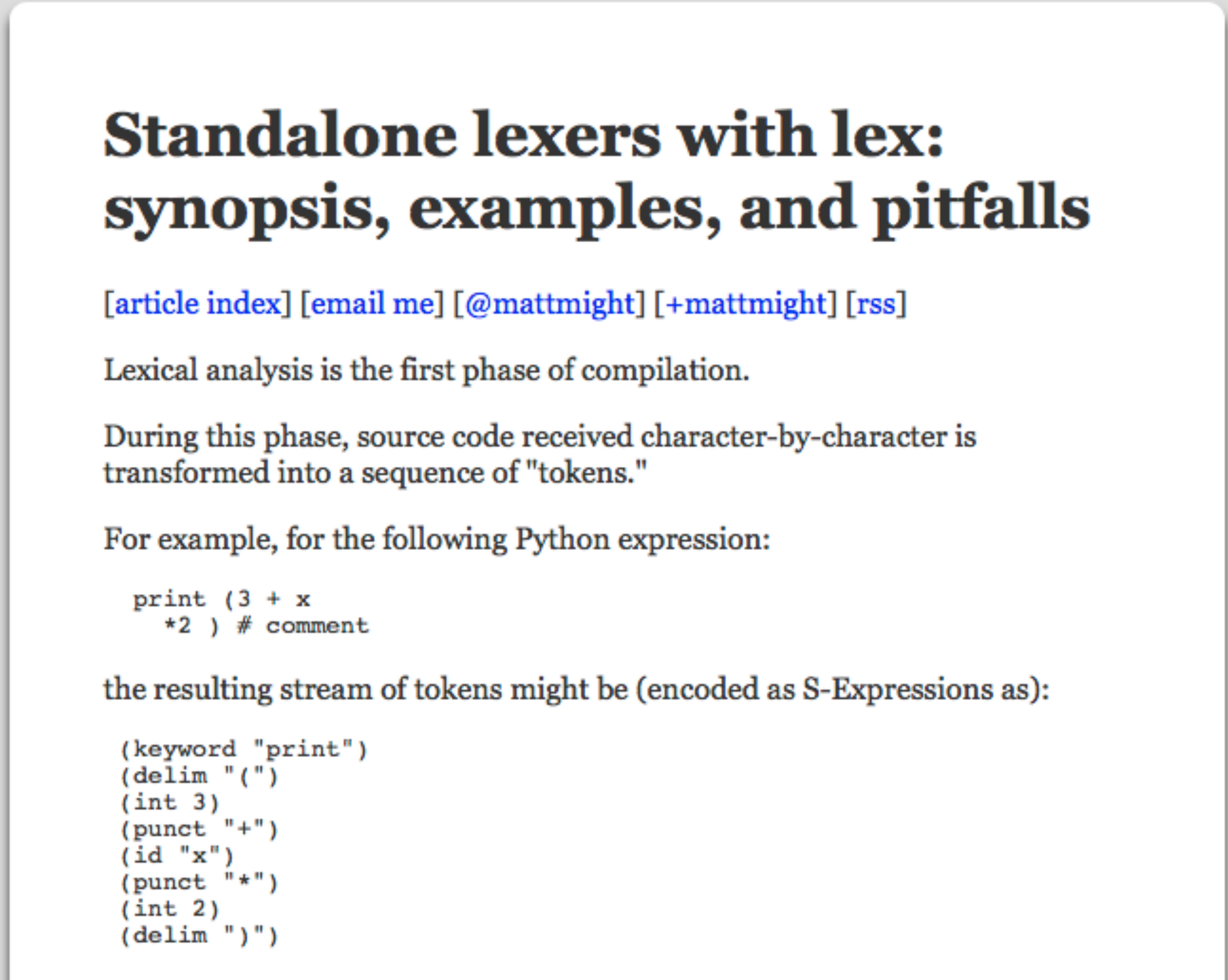
$$L^{\leq} = \{w_1 : \text{there exists } w_2 \text{ such that } w_1 w_2 \in L\}$$

Recognition

$$w \in L?$$

RegEx => NFA => DFA

Derivatives!



Standalone lexers with lex: synopsis, examples, and pitfalls

[article index] [email me] [@mattmight] [+mattmight] [rss]

Lexical analysis is the first phase of compilation.

During this phase, source code received character-by-character is transformed into a sequence of "tokens."

For example, for the following Python expression:

```
print (3 + x
  *2 ) # comment
```

the resulting stream of tokens might be (encoded as S-Expressions as):

```
(keyword "print")
(delim "(")
(int 3)
(punct "+")
(id "x")
(punct "*")
(int 2)
(delim ")")
```

`parser-tools/lex`

Now, in Racket…

Project 1

Python tokens

Integer literals

```
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"
```

Floating point

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

Operators

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

How do you lexically analyze indentation?

Off-side rule

Track levels of indentation with a stack.

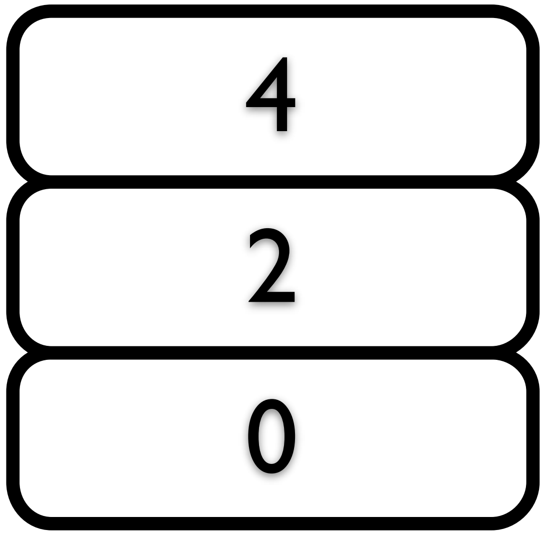
Emit { for increase in indentation.

Emit `}` for increase in indentation.

```
foo
  bar
  baz
    qux
quux
```

```
foo { bar baz { qux } } quux
```

```
foo
  bar
  baz
    qux
quux
```



```
foo { bar baz { qux } } quux
```

Questions?