Lexical Analysis

Matthew Might
University of Utah
matt.might.net

Plan

- What is lexical analysis?
- Formal languages
- Regular languages
- Deterministic automata
- Nondeterministic automata
- Derivatives
- GNU `flex`

Homework questions?

lexing, scanning, tokenizing

Characters in;

tokens out.

Example

```
function identity(x)
{
 return x ; // Comment
}
```

Example

FUNCTION ID(`identity`) LPAR ID(`x`) RPAR

```
{
 return x ; // Comment
}
```

Example

FUNCTION ID(`identity`) LPAR ID(`x`) RPAR
LBRACE

```
 return x ; // Comment
}
```

Example

FUNCTION ID(`identity`) LPAR ID(`x`) RPAR
LBRACE
RETURN ID(`x`) SEMI
`}`

Example

FUNCTION ID(`identity`) LPAR ID(`x`) RPAR
LBRACE
RETURN ID(`x`) SEMI
RBRACE

Lexical specification

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

- Identifiers match `[A-Za-z_][A-Za-z0-9_]*`
- Delimiters match `[();]`
- Operators match `[-*+/^=]`
- Integers match `-?([1-9][0-9]*|0)`
- Whitespace is ignored
- Comments match `^#[^\n]*`

Formal languages

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

A **string** is a sequence of characters from an alphabet.

Notation

A, Σ – Alphabets

L – Language

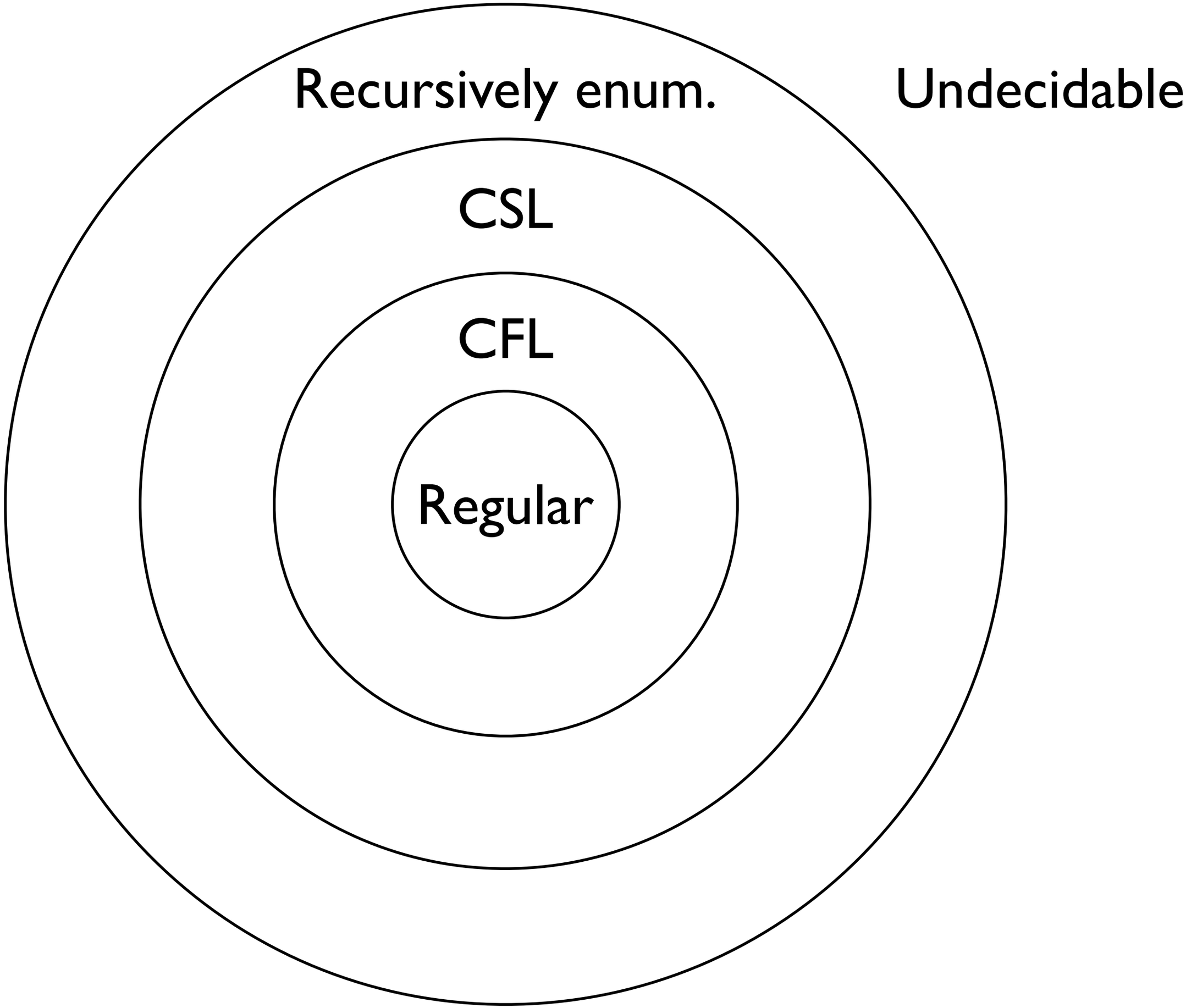
$L \subseteq A^*$ – Language over alphabet A

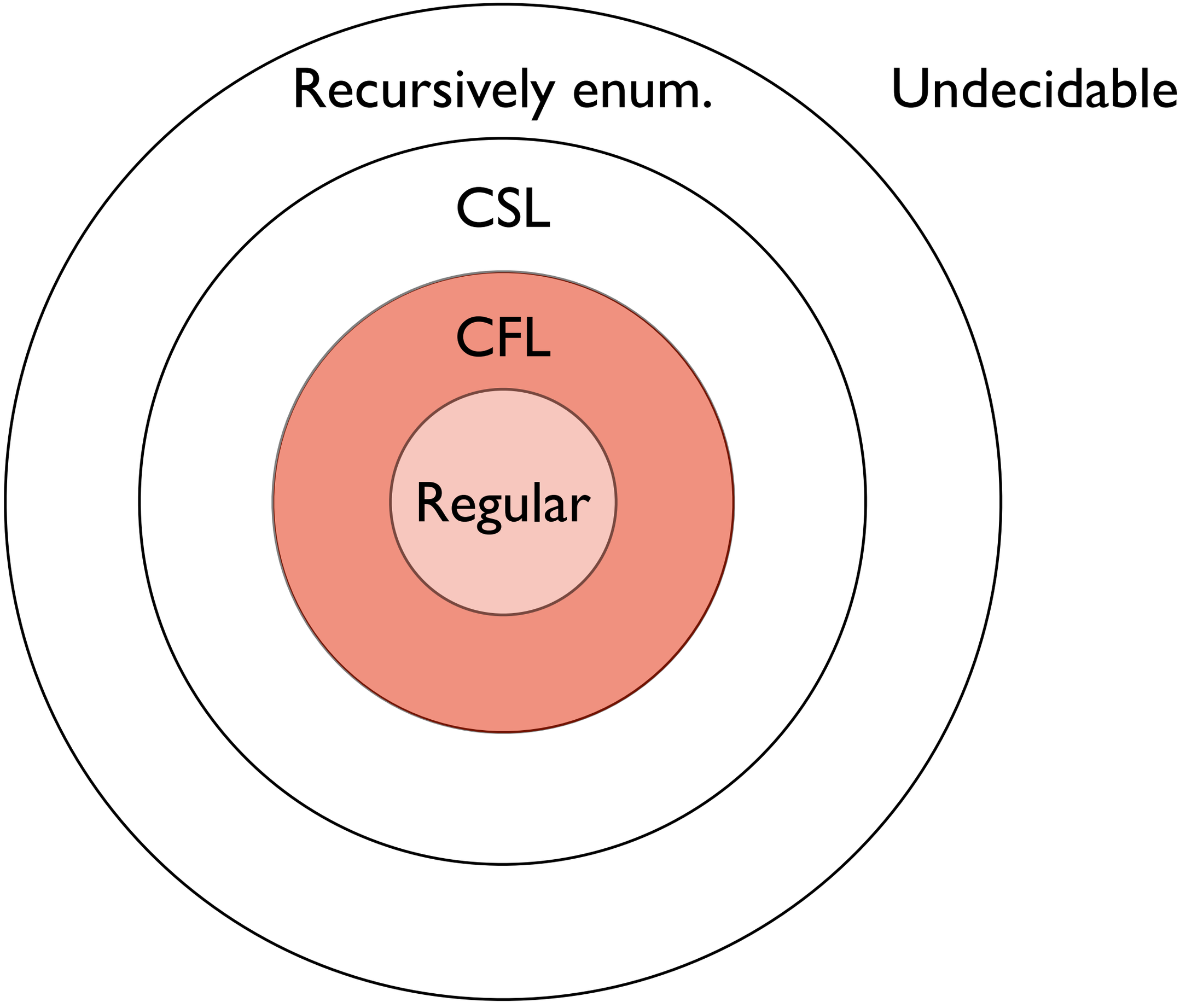
Examples

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

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

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


Recursively enum.
Undecidable
CSL
CFL
Regular


Recursively enum.
Undecidable
CSL
CFL
Regular

Atomic languages

$\emptyset$

ϵ

" "

$$\epsilon = \{\text{''}\,\text{''}\}$$

$$\epsilon = \text{''''}$$

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

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

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

Union

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

Exercise

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

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

Exponentiation

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

$$L^0 = \epsilon$$

Exercise

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

Kleene star

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

Exercise

$$\{a, b\}^\star$$

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

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

$$w \in L?$$

RegEx => NFA => DFA

or, derivatives.

An **automaton** is a state machine that recognizes a formal language.

Formally, a **deterministic automaton** M is a 5-tuple (A, Q, q_0, δ, F) where:

- The set A is an alphabet.
- The set Q is a set of control states.
- The control state q_0 is the initial control state.
- The function $\delta : Q \times A \to Q$ determines the next state.
- The set F determines the final (accepting) states.

$$\mathcal{L}(A, Q, q_0, \delta, F) \subseteq A^*$$

If P_1 and $\ldots$ and P_n, then Q.

$$P_1 \quad \cdots \quad P_n$$

$$Q$$

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

If P_1 and $\ldots$ and P_n, then Q.

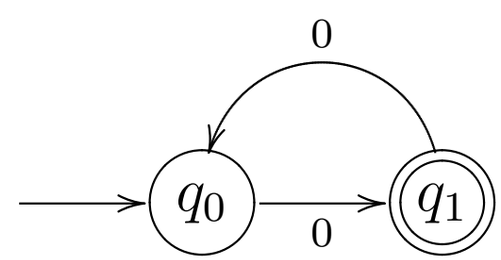
$$\frac{q_0 \in F}{\epsilon \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, \delta(q_0, c), \delta, F)}{cw \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

Example 4.2. The automaton (A, Q, q_0, δ, F), where:

$$A = \{0\} \qquad \delta(q_0, 0) = q_1$$
$$Q = \{q_0, q_1\} \qquad \delta(q_1, 0) = q_0$$
$$F = \{q_1\},$$

is depicted graphically as:



and it accepts exactly the set of strings with odd length—$\{0, 000, 00000, \ldots\}$.

Formally, a **nondeterministic automaton** M is also a 5-tuple (A, Q, q_0, δ, F) where:

- The set A is an alphabet.
- The set Q is a set of control states.
- The control state q_0 is the initial control state.
- The function $\delta : Q \times (A \cup \{\epsilon\}) \to \mathcal{P}(Q)$ determines the next state.
- The set F determines the final (accepting) states.

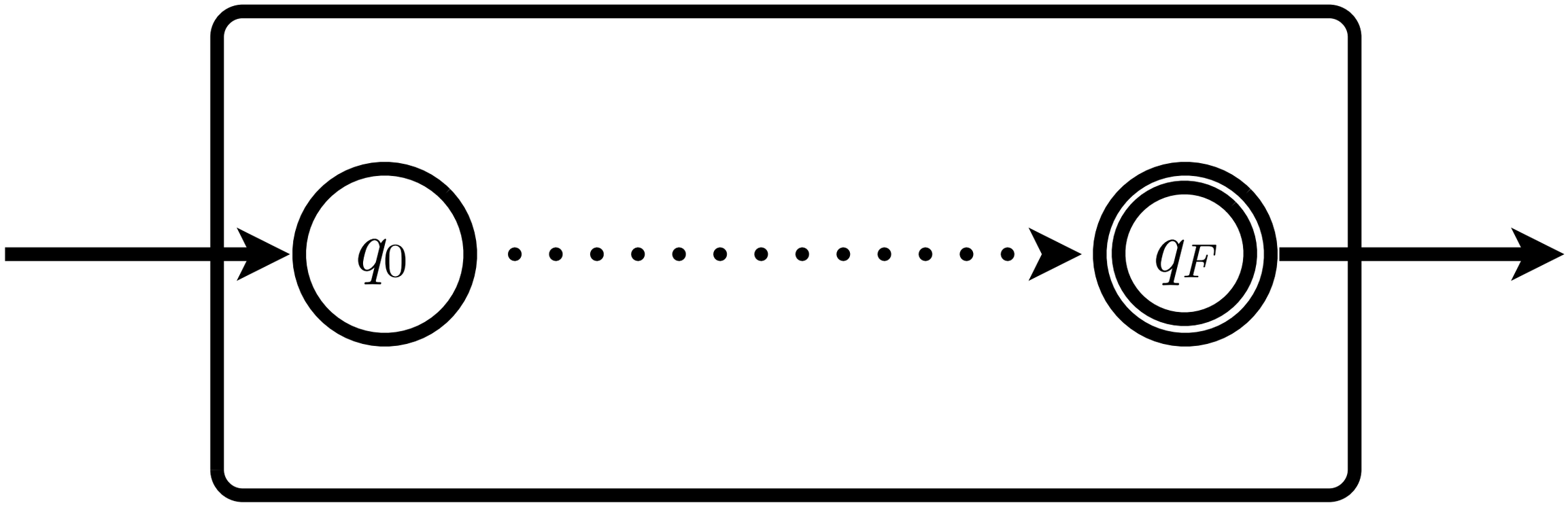
$$\delta : Q \times A \to Q$$

$$\delta : Q \times (A \cup \{\epsilon\}) \to \mathcal{P}(Q)$$

$$\frac{q_0 \in F}{\epsilon \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, q', \delta, F) \qquad q' \in \delta(q_0, c)}{cw \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, q', \delta, F) \qquad q' \in \delta(q_0, \epsilon)}{w \in \mathcal{L}(A, Q, q_0, \delta, F)}$$


q_0
q_F

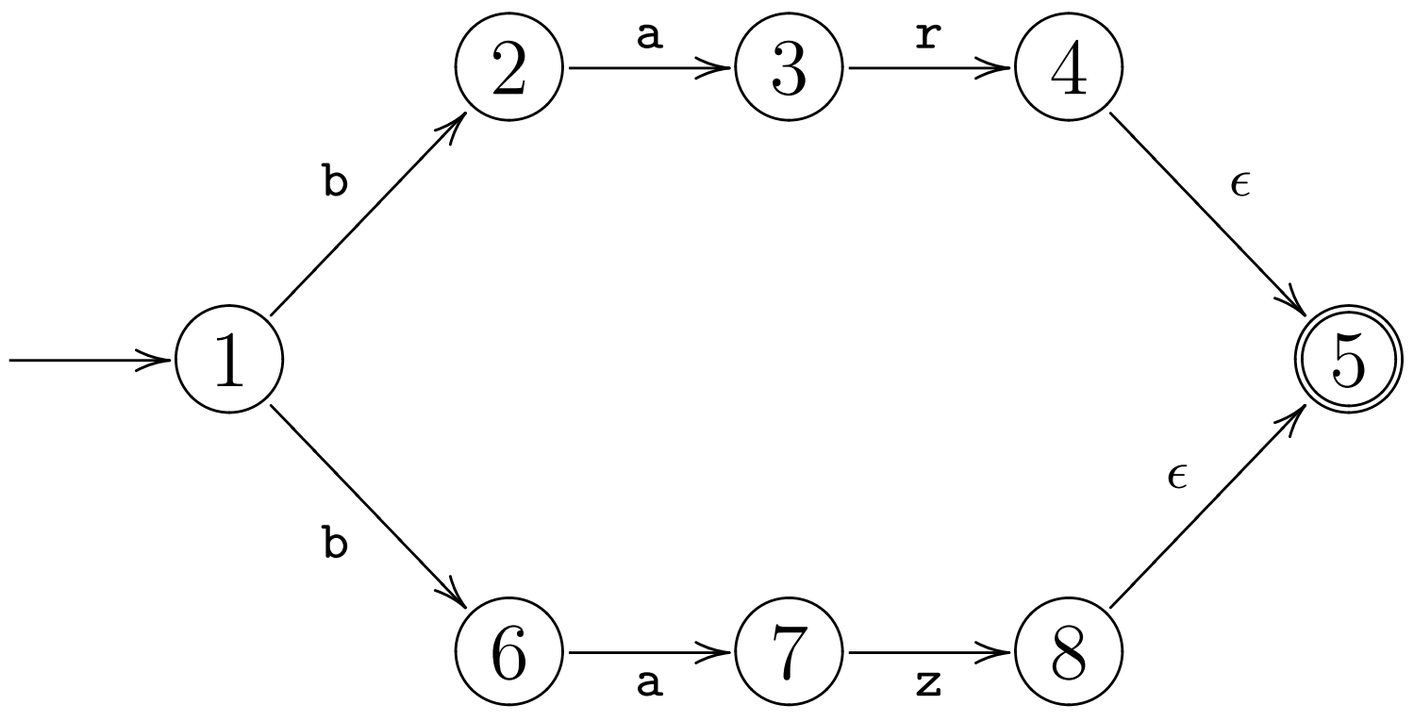
RegEx => NFA

- Empty language
- Empty-string singleton
- One-character singleton
- Concatenation
- Union
- Repetition

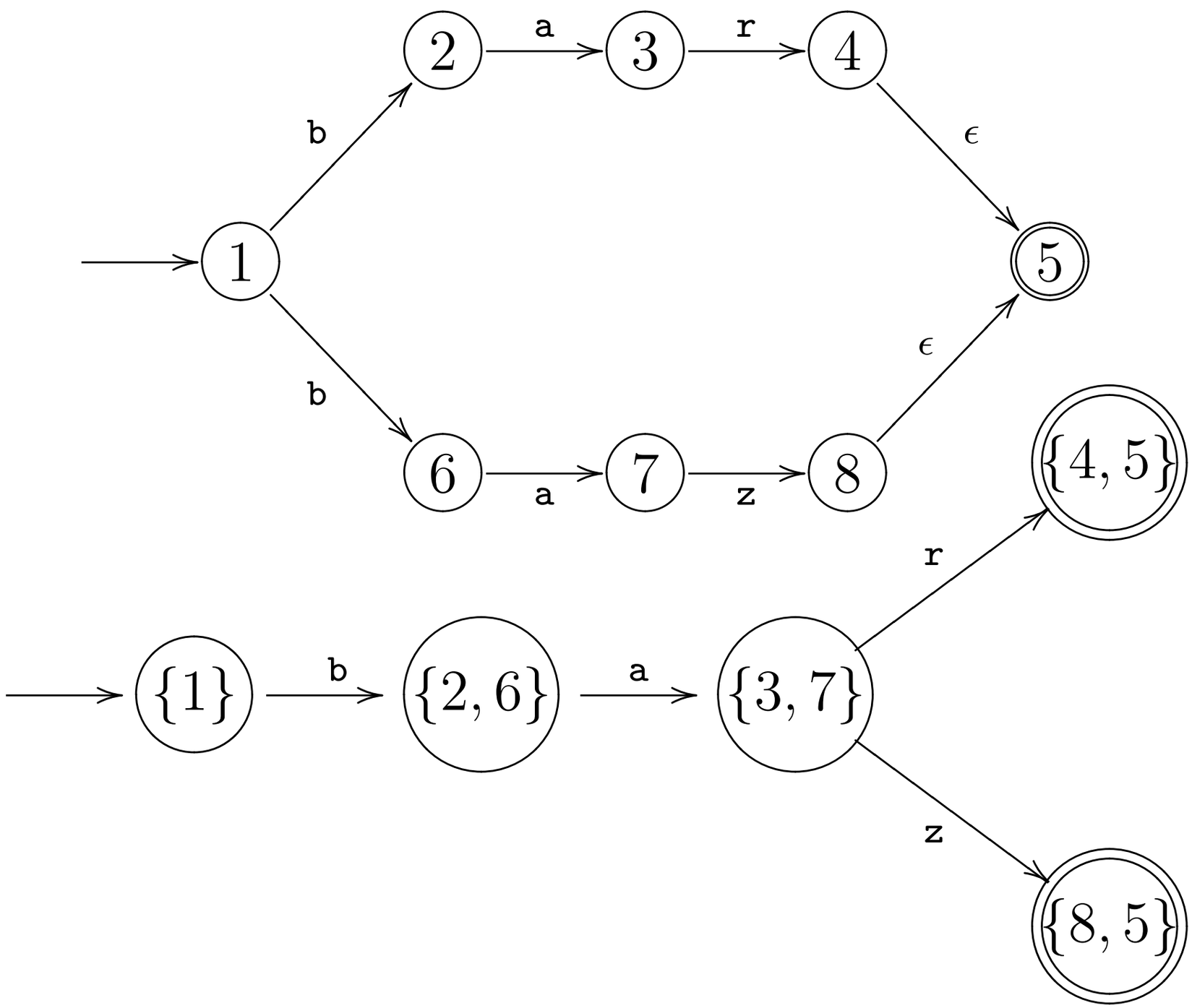
NFA => DFA

Thompson subset construction.

Example: NFA => DFA



Example: NFA => DFA



RegEx => NFA => DFA

$$w \in L?$$

Lexical Analysis

Matthew Might
University of Utah
matt.might.net

Matthew Might
University of Utah
matt.might.net

flex

Matthew Might
University of Utah
matt.might.net

flex

Derivatives

Matthew Might
University of Utah
matt.might.net

Midterm: Feb 23

Derivatives of Regular Expressions

JANUSZ A. BRZOZOWSKI

Princeton University, Princeton, New Jersey†

Abstract. Kleene's regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.

1964

Derivatives of Regular Expressions

JANUSZ A. BRZOZOWSKI

Princeton University, Princeton, New Jersey†

Abstract. Kleene's regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.

D_cL

1. **Filter**:
   Keep every string starting with c.

2. **Chop**:
   Remove c from the start of each.

$D_{\mathfrak{f}}$

foo frak bar

$\mathfrak{D}$

foo frak

$$D_{\mathbf{f}}$$

oo rak

Recognition algorithm

- Derive with respect to each character.
- Does the derived language contain ε?

$$\texttt{foo} \in (\texttt{foo})^*$$

$$\texttt{oo} \in \texttt{f}^{(\texttt{foo})*}$$

$$\texttt{oo} \in D_{\texttt{f}}(\texttt{foo})*$$

$$\texttt{oo} \in \texttt{oo(foo)*}$$

$$\texttt{oo} \in \texttt{oo(foo)*}$$

$$o \in o(foo)^*$$

$$\varepsilon \in \texttt{(foo)*}$$

$$\varepsilon \in (\texttt{foo})^*$$

```
abstract class RegEx {

  public abstract RegEx derive(char c) ;
  public abstract boolean isNullable() ;

  public boolean matches(String s) {
    if (s.length() == 0)
      return this.isNullable() ;
    else
      return this.derive(s.charAt(0)).matches(s.substring(1)) ;
  }
}
```

Deriving atomic languages

$$\epsilon \equiv \{\text{''}\,\text{''}\}$$

$$c \equiv \{c\}$$

$$\emptyset \equiv \{\}$$

```
class Primitive extends RegEx { public Primitive(char c) ... }

class Blank extends RegEx { public Blank() ... }

class Empty extends RegEx { public Empty() ... }
```

$$D_c\emptyset =$$

$$D_c \emptyset = \emptyset$$

```
class Empty extends RegEx {
  ...
  public RegEx derive (char c) {
    return RegEx.empty ;
  }
  ...
}
```

```
abstract class RegEx {

  public abstract RegEx derive(char c) ;
  public abstract boolean isNullable() ;

  public boolean matches(String s) {
    if (s.length() == 0)
      return this.isNullable() ;
    else
      return this.derive(s.charAt(0)).matches(s.substring(1)) ;
  }

  public static final RegEx empty = new Empty() ;
  public static final RegEx blank = new Blank() ;
}
```

$$D_c(\epsilon) =$$

$$D_c(\epsilon) = \emptyset$$

```
class Blank extends RegEx {
  ...
  public RegEx derive (char c) {
    return RegEx.empty ;
  }
  ...
}
```

$$D_c\{c\} = \epsilon$$

$$D_c\{c\} = \epsilon$$

$$D_c\{c'\} = \emptyset \text{ if } c \neq c'$$

```
class Primitive extends RegEx {
  ...
  public RegEx derive (char c) {
    if (c == this.c)
      return RegEx.blank ;
    else
      return RegEx.empty ;
  }
  ...
}
```

Deriving regular languages

$$L_1 \cup L_2$$

$$L_1 \cdot L_2$$

$$L_1^\star$$

```
class Choice extends RegEx
{ public Choice(RegEx thisOne, RegEx thatOne) ... }

class Sequence extends RegEx
{ public Sequence(RegEx first, RegEx second) ... }

class Repetition extends RegEx
{ public Repetition(RegEx internal) ... }
```

$$D_c(L_1 \cup L_2)$$

$$
\begin{aligned}
D_c(L_1 \cup L_2) &= \{w : cw \in L_1 \cup L_2\} \\
&= \{w : cw \in L_1 \text{ or } cw \in L_2\} \\
&= \{w : w \in D_c L_1 \text{ or } w \in D_c L_2\} \\
&= \{w : w \in D_c L_1\} \cup \{\, w : w \in D_c L_2\} \\
&= D_c L_1 \cup D_c L_2.
\end{aligned}
$$

```
class Choice extends RegEx {
  ...
  public RegEx derive (char c) {
    return new Choice(thisOne.derive(c),
                      thatOne.derive(c)) ;
  }
  ...
}
```

$$D_c(L^\star) =$$

$$D_c(L^\star) = (D_c L) \cdot L^\star$$

```
class Repetition extends RegEx {
  ...
  public RegEx derive (char c) {
    return new Sequence(internal.derive(c),this) ;
  }
  ...
}
```

Concatenation?

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2)$$

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2)$$

Needs nullability

$$\delta(L) = \epsilon \text{ if } \epsilon \in L$$

$$\delta(L) = \emptyset \text{ if } \epsilon \notin L$$

$$D_c(L_1 \cdot L_2) =$$

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2)$$

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \cup (\delta(L_1) \cdot D_c L_2)$$

```
class Sequence extends RegEx {
  ...
  public RegEx derive (char c) {
    if (first.isNullable())
      return new Choice(new Sequence(first.derive(c),
                                     second),
                        second.derive(c)) ;
    else
      return new Sequence(first.derive(c),second) ;
  }
  ...
}
```

To recognize?

Need nullability

Need nullability

Need to *compute* nullability

$$\delta(\epsilon) = \epsilon$$

$$\delta(c) = \emptyset$$

$$\delta(\emptyset) = \emptyset$$

```
class Primitive extends RegEx {
  ...
  public boolean isNullable () { return false ; }
  ...
}

class Blank extends RegEx {
  ...
  public boolean isNullable () { return true ; }
  ...
}

class Empty extends RegEx {
  ...
  public boolean isNullable () { return false ; }
  ...
}
```

$$\delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2)$$

$$\delta(L_1 \cdot L_2) = \delta(L_1) \cdot \delta(L_2)$$

$$\delta(L_1^\star) = \epsilon$$

```
class Choice extends RegEx {
  ...
  public boolean isNullable ()
  { return thisOne.isNullable() || thatOne.isNullable ; }
  ...
}

class Sequence extends RegEx {
  ...
  public boolean isNullable ()
  { return first.isNullable() && second.isNullable ; }
  ...
}

class Repetition extends RegEx {
  ...
  public boolean isNullable () { return true ; }
  ...
}
```

But, there’s more!

$$D_c(L_1 \cap L_2) =$$

$$D_c(L_1 \cap L_2) = D_c(L_1) \cap D_c(L_2)$$

$$D_c(L_1 - L_2) =$$

$$D_c(L_1 - L_2) = D_c(L_1) - D_c(L_2)$$

$$D_c(\overline{L}) =$$

$$D_c(\overline{L}) = \overline{D_c(L)}$$

What about nullability?

$$\delta(L_1 \cap L_2) =$$

$$\delta(L_1 \cap L_2) = \delta(L_1) \cap \delta(L_2)$$

$$\delta(L_1 - L_2) =$$

$$\delta(L_1 - L_2) = \delta(L_1) - \delta(L_2)$$

$$\delta(\overline{L}) =$$

$$\delta(\overline{L}) = \overline{\delta(L)} \cap \{\epsilon\}$$

lex

flex

lexical spec => lexer

- Identifiers match `[A-Za-z_][A-Za-z0-9_]*`
- Delimiters match `[();]`
- Operators match `[-*+/^=]`
- Integers match `-?([1-9][0-9]*|0)`
- Whitespace is ignored
- Comments match `^#[^\n]*`

How lex works

foo.l

```
%%
[0-9]+    return 1;
[A-Z]+    return 2;
[ \t\n]   {}
%%
```

foo.l

```
%%
[0-9]+    return 1;
[A-Z]+    return 2;
[ \t\n]   {}
%%
```

lex.yy.c

```
int yylex() {
  ...
}
```

```
defs, opts, decs
%%
rules
%%
C code
```

Definitions

name regex

{*name*}

```
digit [0-9]
int   {digit}+

alpha  [A-Za-z_]
alnum  {alpha}|{digit}
id     {alpha}{alnum}*
```

Options

```
%option case-insensitive
```

```
%option yylineno
```

```
%option noywrap
```

Declarations

```
%{
C code
%}
```

%s $state_1$ $state_2$ ...

%x $state_1$ $state_2$ ...

Rules

```
pattern { action }
```

<state> pattern { action }

If in state *state*,
and *pattern* matches the longest prefix,
then perform *action*.

```
%s state
%%
pat action
<state> pat action
```

```
%x state
%%
pat action
<state> pat action
```

Actions

{ }

```
return token_type ;
```

BEGIN(*state*);

ECHO;

REJECT;

Example

- `<MAIN> ‘‘(‘‘ { return(LPAR) ; }`
- `<MAIN> ‘‘)’’ { return(RPAR) ; }`
- `<MAIN> ‘‘/*’’ { BEGIN(COMMENT) ; }`
- `<COMMENT> ‘‘*/’’ { BEGIN(MAIN) ; }`
- `<COMMENT> . { }`

Matching length

- Suppose the pattern: `f(oo)*`
- How many ways can match:

  `foooooooooooooooooo`

Options

- Shortest match
- Longest match

Longest match

Given a string *w* and a set of regular expressions *R*, which regex can match the longest prefix of *w*?

Algorithm

```
NaïveRemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
  suffix ← w
  while (w ≠ ε)
   if ∃L ∈ R : ε ∈ L
      suffix ← w
   c:w ← w
   R ← D_c.R
  return suffix
```

Algorithm

```
RemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
  suffix ← w
  while (R ≠ ∅ or w ≠ ε)
   if ∃L ∈ R : ε ∈ L
      suffix ← w
   c:w ← w
   R ← D_c.R - {∅}
  return suffix
```

Example

- RegEx: fo*
- RegEx: foobar
- RegEx: foob
- Input string: foobarbaz

Examples

```
%{
 int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
int main(int argc, char* argv[]) {
   yylex();
   printf("# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```