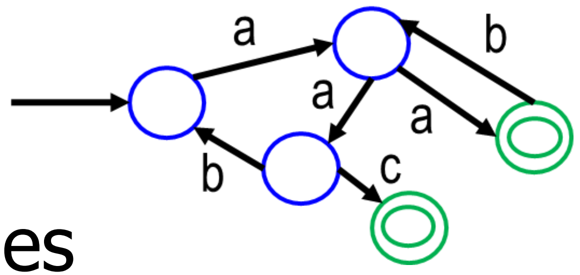
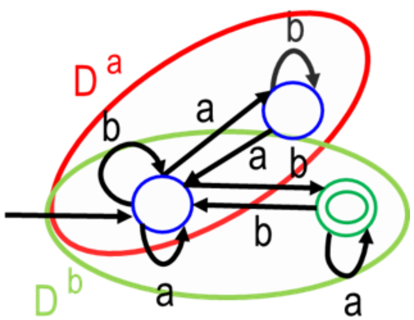
Topic 2: Lexing and Flexing

COS 320



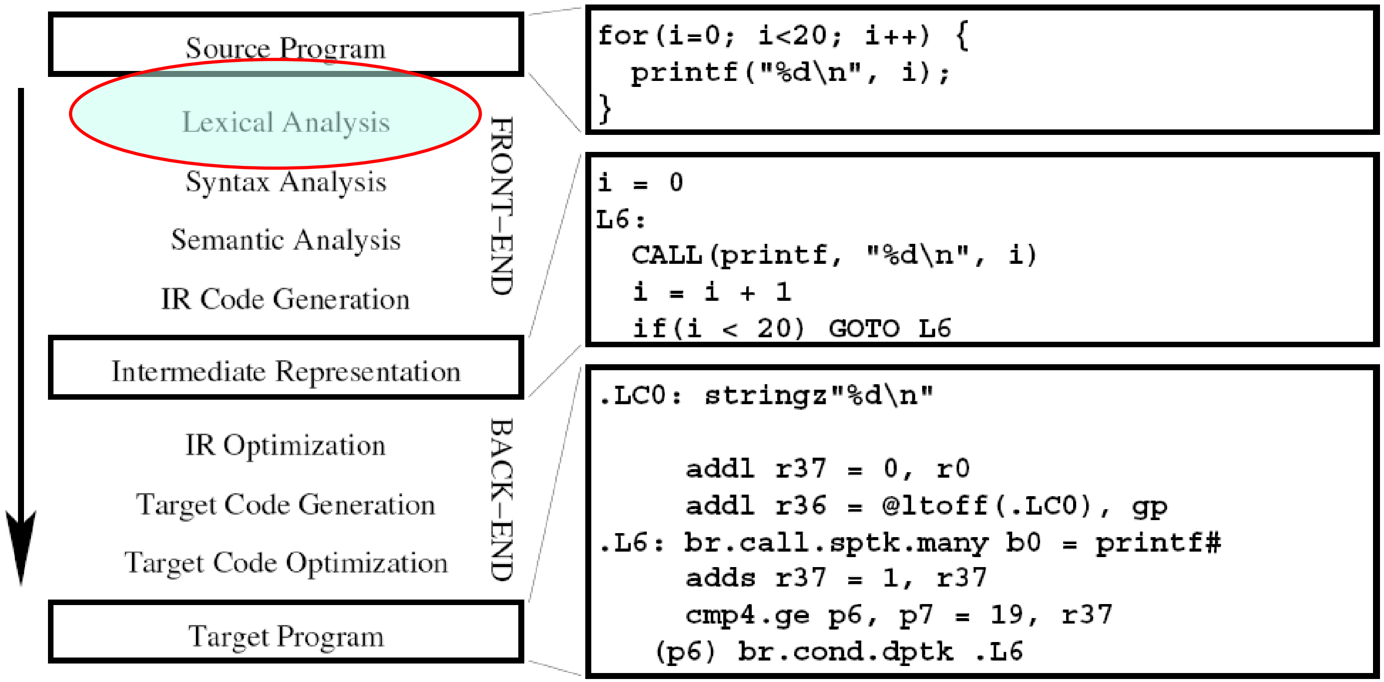
Compiling Techniques



Princeton University
Spring 2016

Lennart Beringer

The Compiler



- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

Lexical Analysis

- Goal: break stream of ASCII characters (source/input) into sequence of **tokens**
- **Token**: sequence of characters treated as a unit (cf. **word**)

Lexical Analysis

- Goal: break stream of ASCII characters (source/input) into sequence of **tokens**
- **Token**: sequence of characters treated as a unit (cf. **word**)
- Each token has a **token type** (cf. classification **verb** - **noun** - **punctuation symbol**):

Lexical Analysis

- Goal: break stream of ASCII characters (source/input) into sequence of **tokens**
- <u>**Token**</u>: sequence of characters treated as a unit (cf. **word**)
- Each token has a **token type** (cf. classification **verb** - **noun** - **punctuation symbol**):

IDENTIFIER	foo, x, quicksort, …	NUM	1, 50, -100
REAL	6.7, 3.9E-33, -4.9	IF	if
SEMI	;	THEN	then
LPAREN (	RPAREN)	EQ =	PLUS +

…

Lexical Analysis

- Goal: break stream of ASCII characters (source/input) into sequence of **tokens**
- **Token**: sequence of characters treated as a unit (cf. **word**)
- Each token has a **token type** (cf. classification **verb** - **noun** - **punctuation symbol**):

IDENTIFIER	foo, x, quicksort, …	NUM	1, 50, -100
REAL	6.7, 3.9E-33, -4.9	IF	if
SEMI	;	THEN	then
LPAREN (	RPAREN)	EQ =	PLUS +

…

- Many tokens have associated semantic information: NUM(1), NUM(50), IDENTIFIER(foo), IDENTIFIER(x), but typically not SEMI(;), LPAREN(()

Lexical Analysis

- Goal: break stream of ASCII characters (source/input) into sequence of **tokens**
- **Token**: sequence of characters treated as a unit (cf. **word**)
- Each token has a **token type** (cf. classification **verb** - **noun** - **punctuation symbol**):

IDENTIFIER	foo, x, quicksort, …	NUM	1, 50, -100
REAL	6.7, 3.9E-33, -4.9	IF	if
SEMI	;	THEN	then
LPAREN (	RPAREN)	EQ =	PLUS +

…

- Many tokens have associated semantic information: NUM(1), NUM(50), IDENTIFIER(foo), IDENTIFIER(x), but typically not SEMI(;), LPAREN(()
- Definition of tokens (mostly) part of **language definition**
- White space and comments often discarded. Pros/Cons?

Lexical Analysis Example

```
x = ( y + 4.0 );
```

Lexical Analysis Example

```
x = ( y + 4.0 );
```

IDENTIFIER(x) EQ LPAREN IDENTIFIER(y) PLUS

NUM(4.0) RPAREN SEMI

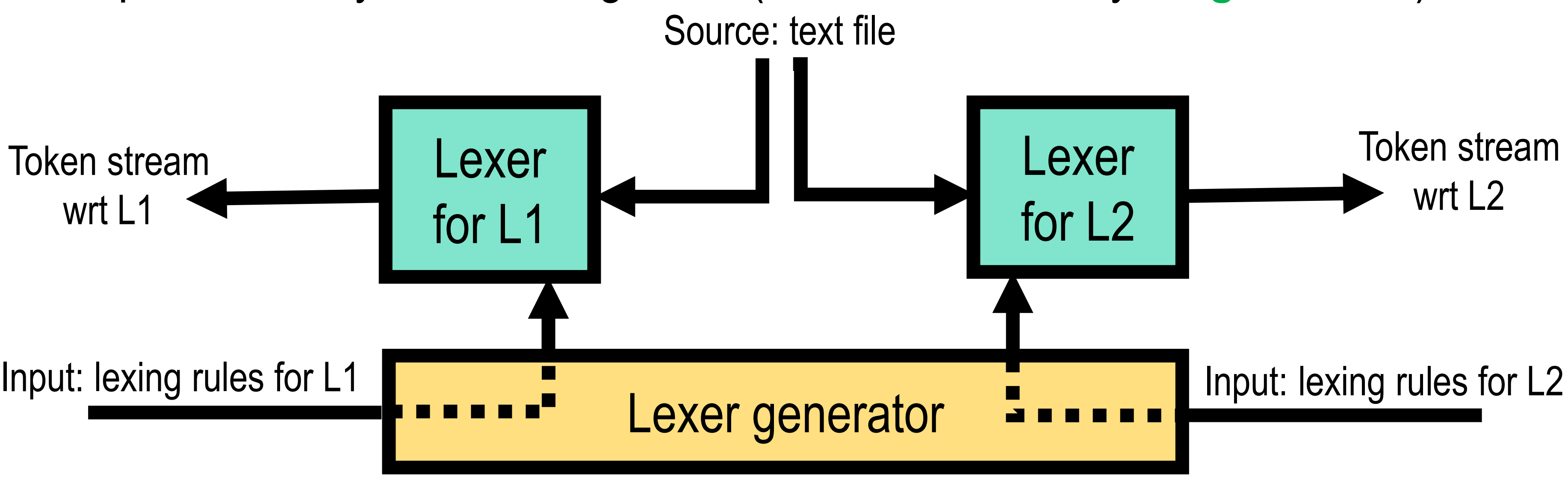
Implementing a Lexical Analyzer (Lexer)

Option 1: write it from scratch:

Implementing a Lexical Analyzer (Lexer)

Option 1: write it from scratch:

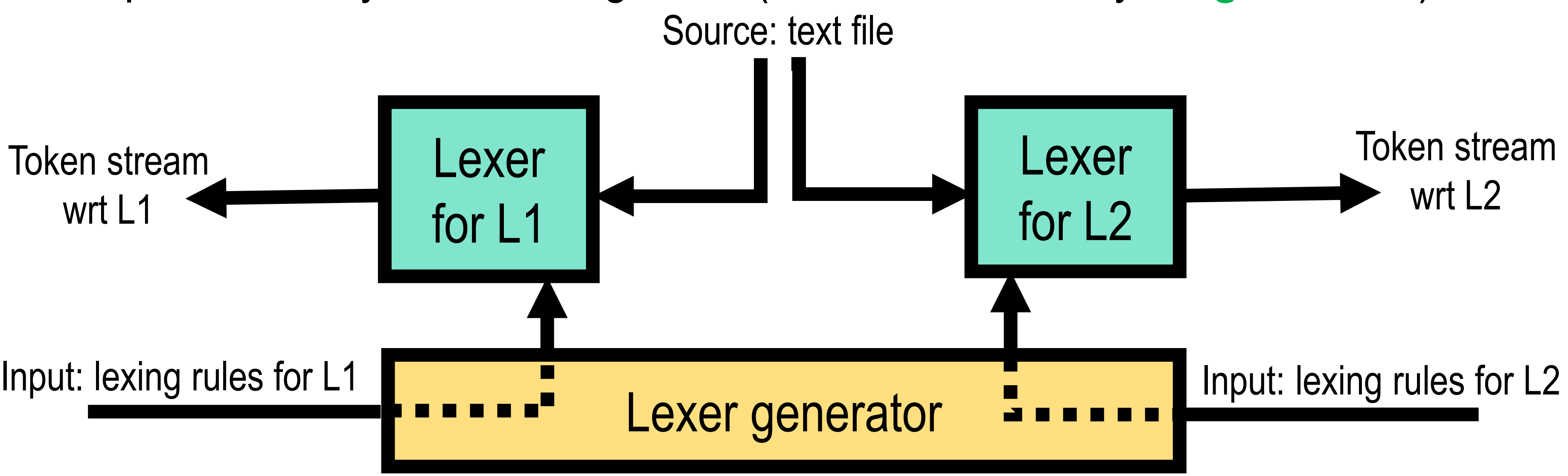
Option 2: eat your own dog food! (use a lexical analyzer **generator**):



Implementing a Lexical Analyzer (Lexer)

Option 1: write it from scratch

Option 2: eat your own dog food! (use a lexical analyzer **generator**)



Q: how do we describe the tokens for L1, L2, …?

A: using another language, of course!

Yeah, but how do we describe the tokens of that language???

Theory to the rescue: regular expressions

Some definitions

- An **alphabet** is a (finite) collection of symbols.
  Examples: **ASCII**, {0, 1}, {A, ..Z, a, .. Z}, {0, ..9}

Theory to the rescue: regular expressions

Some definitions

- An **alphabet** is a (finite) collection of symbols.
  Examples: **ASCII**, {0, 1}, {A, ..Z, a, .. Z}, {0, ..9}
- A **string/word** (over alphabet **A**) is a finite sequence of symbols from **A**.

Theory to the rescue: regular expressions

Some definitions

- An **alphabet** is a (finite) collection of symbols.
  Examples: **ASCII**, {0, 1}, {A, ..Z, a, .. Z}, {0, ..9}
- A **string/word** (over alphabet **A**) is a finite sequence of symbols from **A**.
- A **language** (over **A**) is a (finite or infinite) set of strings over **A**.

Theory to the rescue: regular expressions

Some definitions

- An **alphabet** is a (finite) collection of symbols.
  Examples: **ASCII**, {0, 1}, {A, ..Z, a, .. Z}, {0, ..9}
- A **string/word** (over alphabet **A**) is a finite sequence of symbols from **A**.
- A **language** (over **A**) is a (finite or infinite) set of strings over **A**.
  Examples:
  - the ML language: set of all strings representing correct ML programs (infinite)
  - the language of ML keywords: set of all strings that are ML keywords (finite)
  - the language of ML tokens: set of all strings that map to ML tokens (infinite)

Theory to the rescue: regular expressions

Some definitions

- An **alphabet** is a (finite) collection of symbols.
  Examples: **ASCII**, {0, 1}, {A, ..Z, a, .. Z}, {0, ..9}
- A **string/word** (over alphabet **A**) is a finite sequence of symbols from **A**.
- A **language** (over **A**) is a (finite or infinite) set of strings over **A**.
  Examples:
  - the ML language: set of all strings representing correct ML programs (infinite)
  - the language of ML keywords: set of all strings that are ML keywords (finite)
  - the language of ML tokens: set of all strings that map to ML tokens (infinite)

Q: How to describe languages? **A**(for lexing)**: regular expressions!**

REs are **finite descriptions**/representations of (certain) finite or infinite languages, including

- the language of a (programming) language's tokens (eg the language of ML tokens)
- the language describing the language of a (programming) language's tokens,
- the language describing …

Constructing regular expressions

Base cases

Inductive cases: given RE's M and N,

Constructing regular expressions

Base cases

- the RE **ε** (epsilon): the (finite) language containing only the empty string.
- for each symbol **a** from **A**, the RE **a** denotes the (finite) language containing only the string **a**.

Inductive cases: given RE's M and N,

Constructing regular expressions

<u>Base cases</u>

- the RE **ε** (epsilon): the (finite) language containing only the empty string.
- for each symbol **a** from **A**, the RE **a** denotes the (finite) language containing only the string **a**.

<u>Inductive cases: given RE's M and N,</u>

- the RE **M | N** (alternation, union) describes the language containing the strings in **M** or **N**.
  Example: a | b denotes the two-element language {a, b}

Constructing regular expressions

Base cases

- the RE **ε** (epsilon): the (finite) language containing only the empty string.
- for each symbol **a** from **A**, the RE **a** denotes the (finite) language containing only the string **a**.

Inductive cases: given RE's M and N,

- the RE **M | N** (alternation, union) describes the language containing the strings in **M** or **N**.
  Example: a | b denotes the two-element language {a, b}
- The RE **MN** (concatenation) denotes the strings that can be written as the concatenation **mn** where **m** in from **M** and **n** is from **N**.
  Example: (a|b)(a|c) denotes the language {aa, ac, ba, bc}

Constructing regular expressions

Base cases

- the RE **ε** (epsilon): the (finite) language containing only the empty string.
- for each symbol **a** from **A**, the RE **a** denotes the (finite) language containing only the string **a**.

Inductive cases: given RE's M and N,

- the RE **M | N** (alternation, union) describes the language containing the strings in **M** or **N**.

  Example: a | b denotes the two-element language {a, b}
- The RE **MN** (concatenation) denotes the strings that can be written as the concatenation **mn** where **m** in from **M** and **n** is from **N**.

  Example: (a|b)(a|c) denotes the language {aa, ac, ba, bc}
- The RE **M*** (Kleene closure/star) denotes the (**infinitely** many) strings obtained by concatenating **finitely** many elements from **M**.

  Example: (a|b)* denotes the language {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, …}

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Strings with an even number of a's:	$RE^{a} =$

Strings that with an odd number of b's:	$RE^{b} =$

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Solutions not unique!

Strings with an even number of a's:	$RE^a = b^* (a\ b^*\ a\ b^*)^*$

Strings that with an odd number of b's:	$RE^b =$

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Solutions not unique!

Strings with an even number of a's:	$RE^a = b^*(a\,b^*\,a\,b^*)^*$

Strings that with an odd number of b's:	$RE^b = a^*\,b\,a^*\,(b\,a^*\,b\,a^*)^*$

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Strings with an even number of a’s:	$RE^{a} = b^{*} (a\, b^{*}\, a\, b^{*})^{*}$
Strings that with an odd number of b’s:	$RE^{b} = a^{*}\, b\, a^{*} (b\, a^{*}\, b\, a^{*})^{*}$
Strings with an even number of a’s **OR** an odd number of b’s:	$RE^{a,b} =$
Strings that can be split into a string with an even number of a’s, **followed** by a string with an odd number of b’s:	

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Strings with an even number of a's:	$RE^{a} = b^* (a\, b^*\, a\, b^*)^*$
Strings that with an odd number of b's:	$RE^{b} = a^*\, b\, a^* (b\, a^*\, b\, a^*)^*$
Strings with an even number of a's **OR** an odd number of b's:	$RE^{a,b} = RE^{a} \mid RE^{b}$
Strings that can be split into a string with an even number of a's, **followed** by a string with an odd number of b's:	

Regular Expression Examples

For alphabet $\Sigma = \{a,b\}$:

Strings with an even number of a's:	$RE^a = b^* (a\, b^*\, a\, b^*)^*$
Strings that with an odd number of b's:	$RE^b = a^*\, b\, a^* (b\, a^*\, b\, a^*)^*$
Strings with an even number of a's **OR** an odd number of b's:	$RE^{a,b} = RE^a \mid RE^b$
Strings that can be split into a string with an even number of a's, **followed** by a string with an odd number of b's:	$RE_{a,b} = RE^a \; RE^b$

Regular Expression Examples

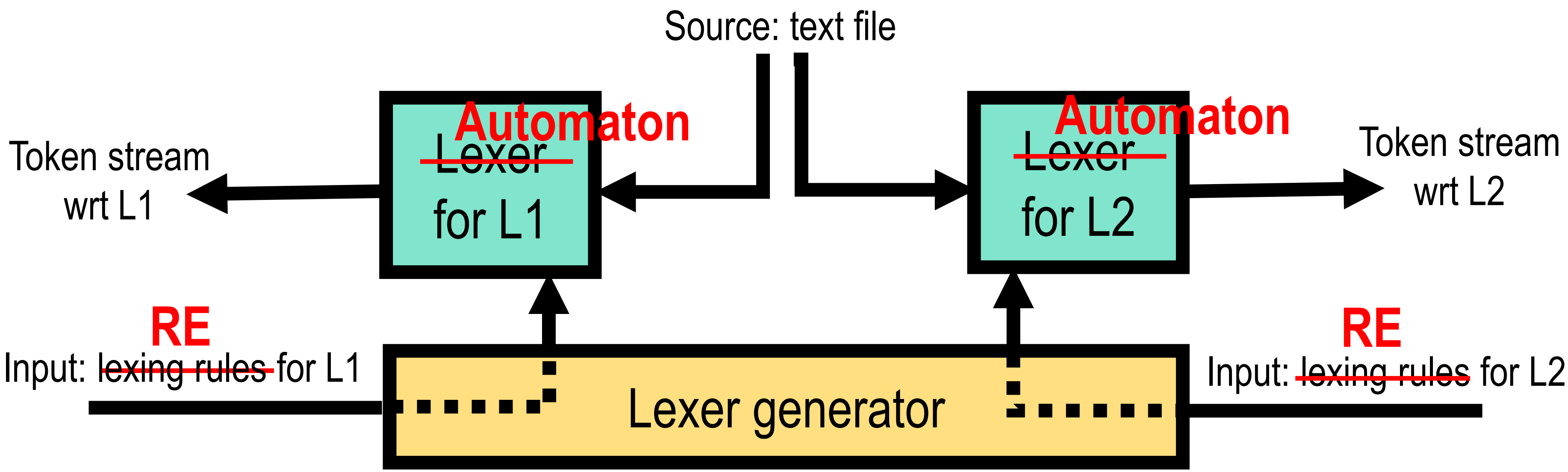
For alphabet $\Sigma = \{a,b\}$:

Strings with an even number of a's:	$RE^a = b^* (a b^* a b^*)^*$
Strings with an odd number of b's:	$RE^b = a^* b a^* (b a^* b a^*)^*$
Strings with an even number of a's **OR** an odd number of b's:	$RE^{a,b} = RE^a \mid RE^b$
Strings that can be split into a string with an even number of a's, **followed** by a string with an odd number of b's:	$RE_{a,b} = RE^a \ RE^b$

Optional Homework:

Strings with an even number of a's and an odd number of b's….

Implementing RE's: finite automata



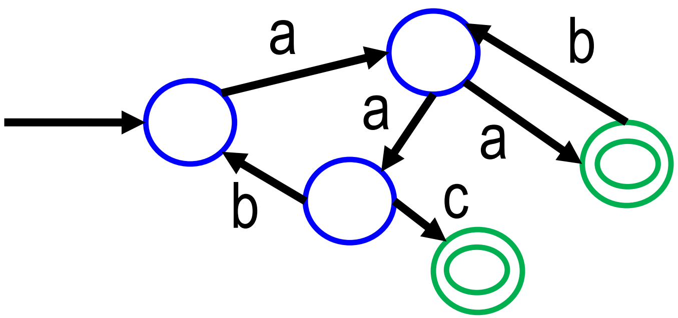
Finite automata (aka finite state machines, FSM's):
a computational model of machines with finite memory

Components of an automaton over alphabet **A**:

- a **finite** set **S** of nodes ("states")
- a set of directed edges ("transitions") $s \xrightarrow{a} t$, each linking two states and labeled with a symbol from **A**
- a designated start state **s0** from **S**, indicated by "arrow from nowhere"
- a nonempty set of **final** ("accepting") states (indicated by double circle)

Finite Automata **recognize** languages

Definition: the language recognized by an FA is the set of (finite) strings it **accepts**.



Q: how does the finite automaton **D** accept a string **A**?

A: follow the transitions:

1. start in the start state **s0** and inspect the first symbol**,** a1
2. when in state **s** and inspecting symbol **a**, traverse one edge labeled **a** to get to the next state. Look at the next symbol.
3. After reading in all **n** symbols: if the current state **s** is a **final** one, **accept**. Otherwise, **reject**.
4. whenever there's no edge whose label matches the next symbol: **reject**.

Classes of Finite Automata

Deterministic finite automaton (DFA)

- all edges leaving a node are uniquely labeled.

Nondeterministic finite automaton (NFA)

- two (or more) edges leaving a node may be identically uniquely labeled. Any choice that leads to acceptance is fine.
- edges may also be labeled with ε. So can “jump” to the next state without consuming an input symbol.

Classes of Finite Automata

Deterministic finite automaton (DFA)
- all edges leaving a node are uniquely labeled.

Nondeterministic finite automaton (NFA)
- two (or more) edges leaving a node may be identically uniquely labeled. Any choice that leads to acceptance is fine.
- edges may also be labeled with ε. So can "jump" to the next state without consuming an input symbol.

Strategy for obtaining a DFA that recognizes exactly the language described by an RE:
1. convert RE into an NFA that recognizes the language
2. transform the NFA into an equivalent DFA

Remember Tuesday's quiz?

NFA Examples

Over alphabet {a, b}:

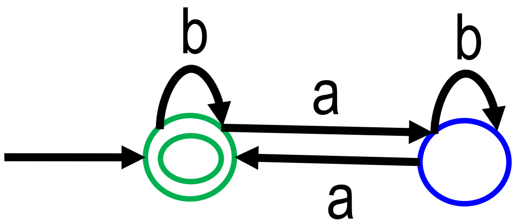
Strings with an even number of a’s:

Strings with an odd number of b’s:

NFA Examples

Over alphabet {a, b}:

Strings with an even number of a's: D^a:

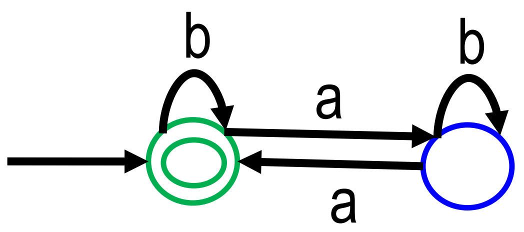


Strings with an odd number of b's:

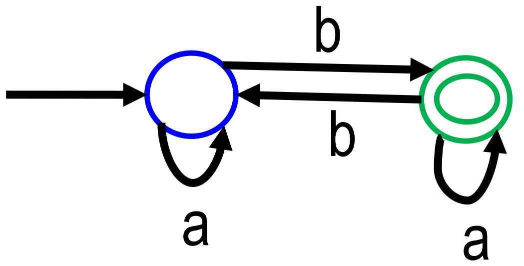
NFA Examples (adhoc)

Over alphabet {a, b}:

Strings with an even number of a's: D^a:

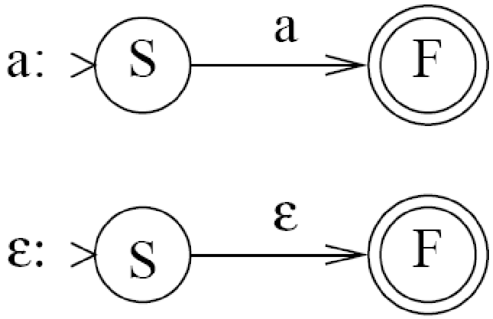


Strings with an odd number of b's: D^b:

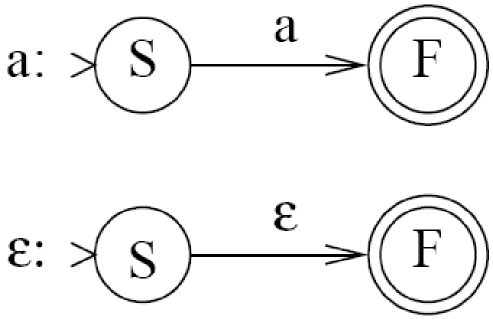


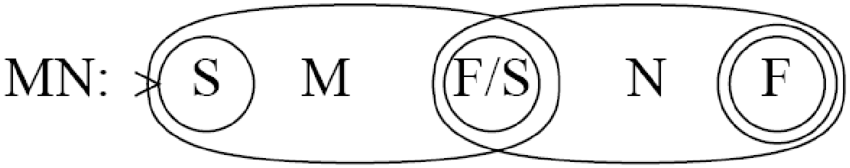
Can we systematically generate NFA's from RE's?

RE to NFA Rules

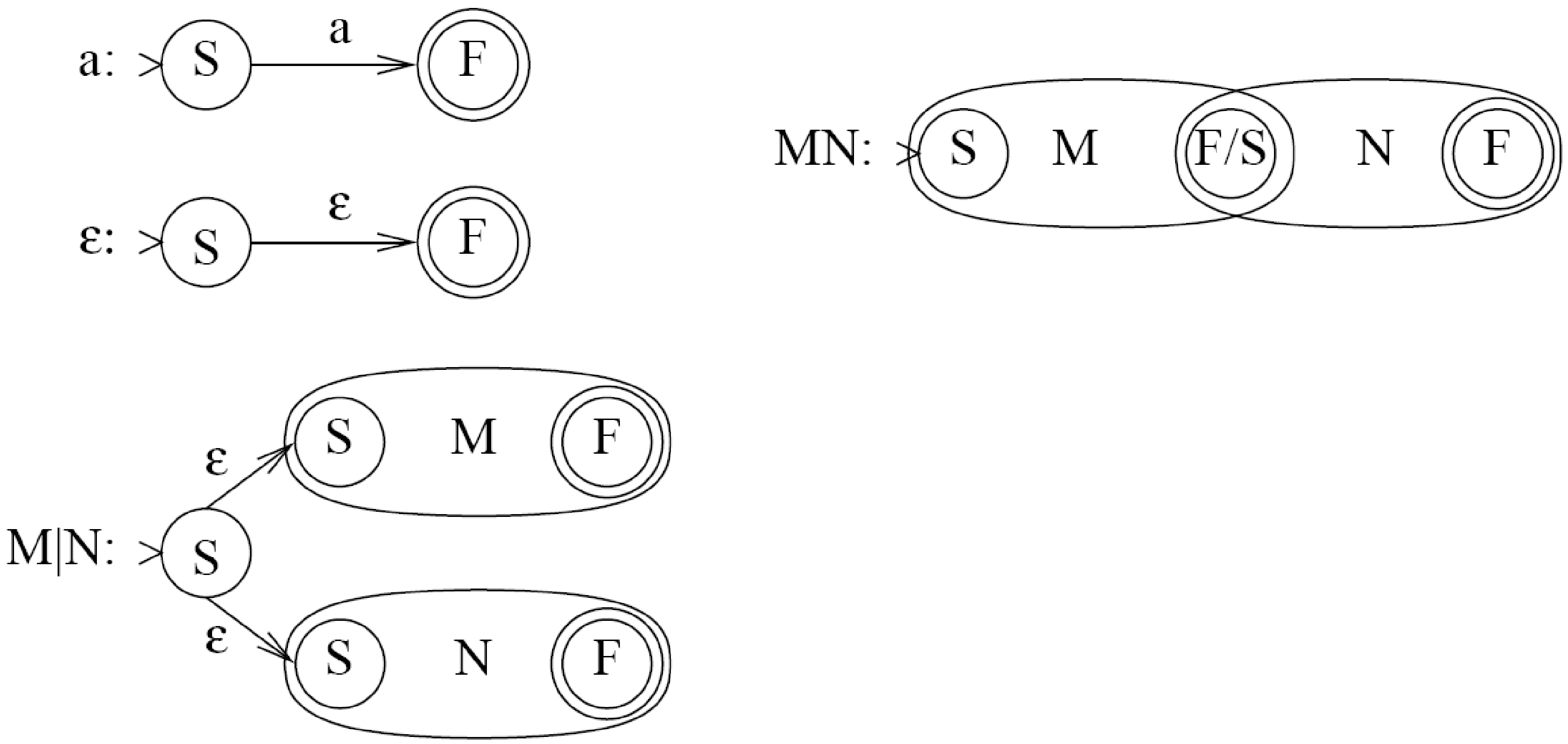


RE to NFA Rules

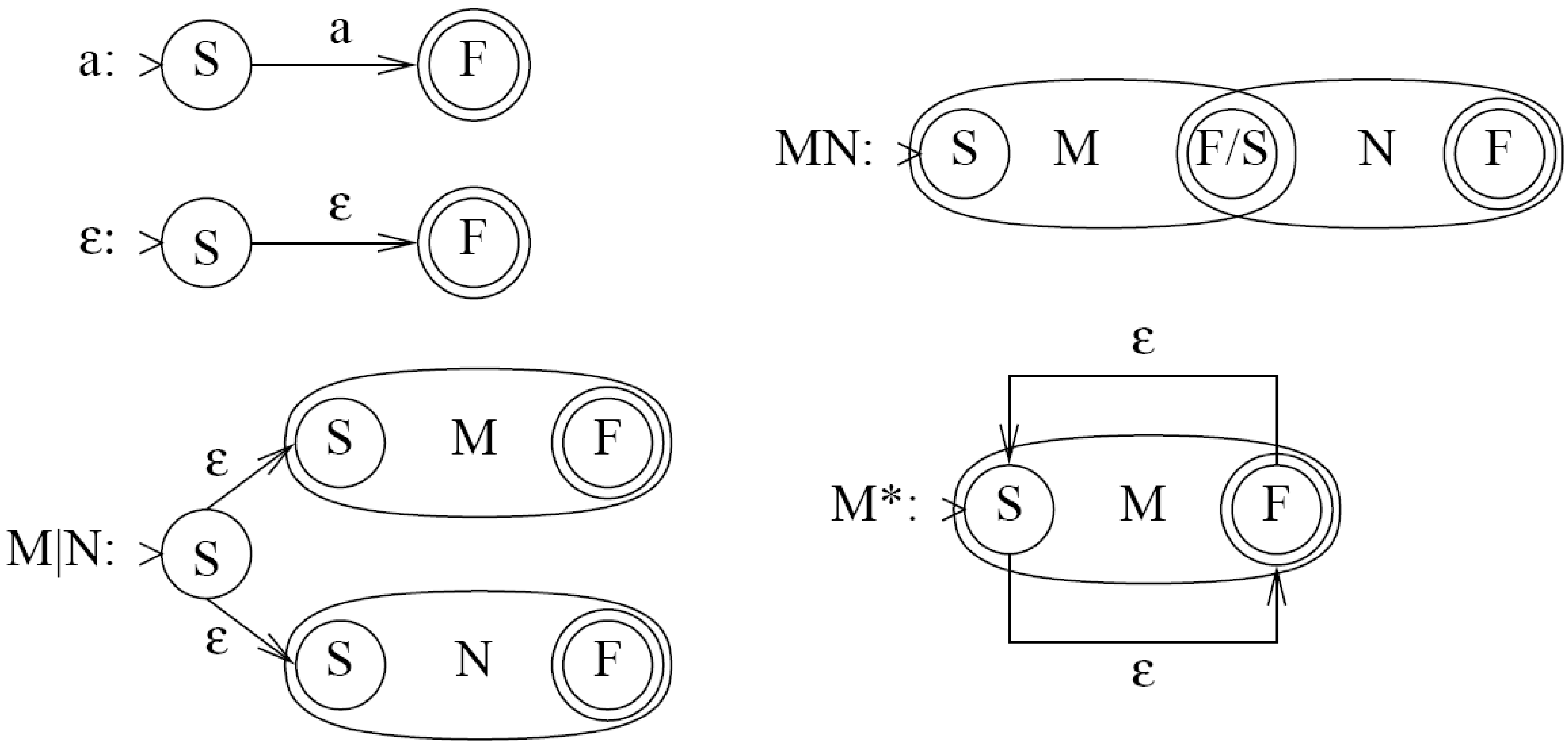




RE to NFA Rules

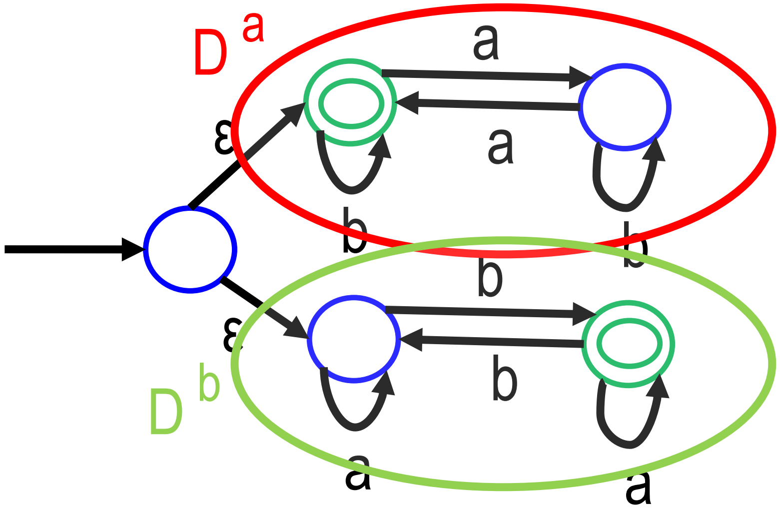


RE to NFA Rules



RE to NFA Conversion: examples for | and concat

Strings with an even number of a's **OR** an odd number of b's: $RE^a \mid RE^b$

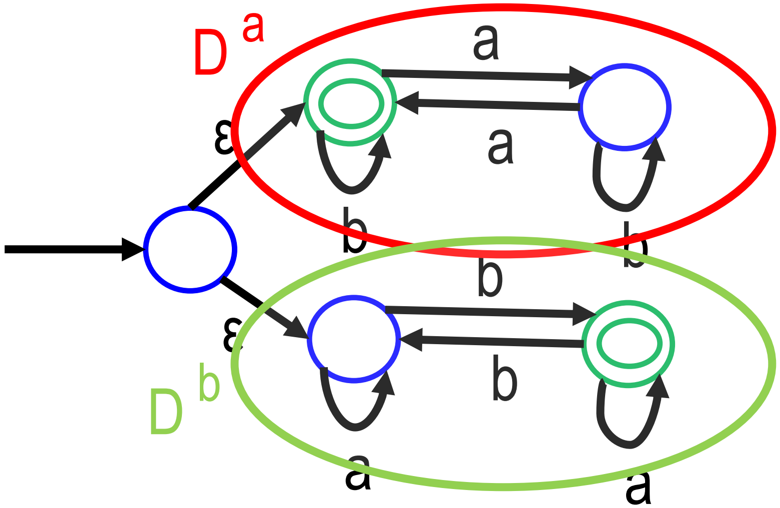


Strings that can be split into a string with an even number of a's, **followed** by a string with an odd number of b's:

$RE^a \quad RE^b$

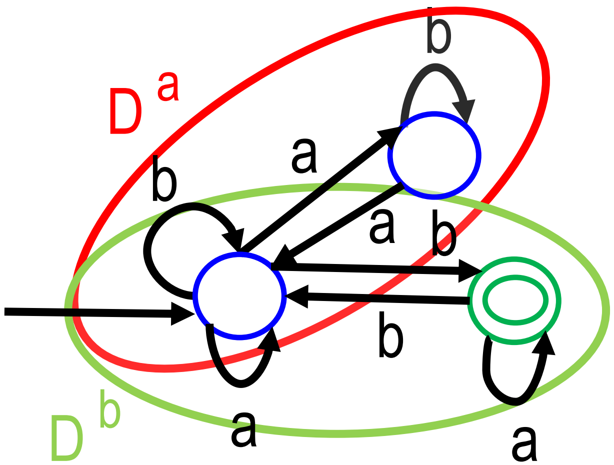
RE to NFA Conversion: examples for | and concat

Strings with an even number of a's
OR an odd number of b's: RE [a] | RE [b]

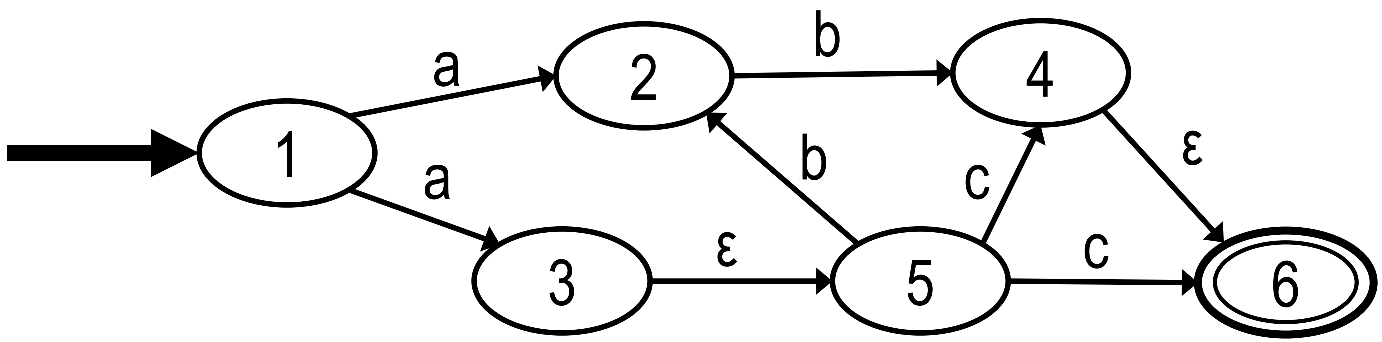


Strings that can be split into a string with an even number of a's **followed** by a string with an odd number of b's:

RE [a] RE [b]



NFA to DFA Conversion



Idea:

- combine identically labeled NFA transitions
- DFA states represent **sets** of “equivalent” NFA states

NFA to DFA conversion

Auxiliary definitions:

$$\text{edge}(s, a) = \{\, t \mid s \xrightarrow{a} t \,\}$$

set of NFA states reachable from NFA state s by an a step

$$\text{closure}(S) = S \cup \left(\bigcup_{s \in S} \text{edge}(s, \varepsilon) \right)$$

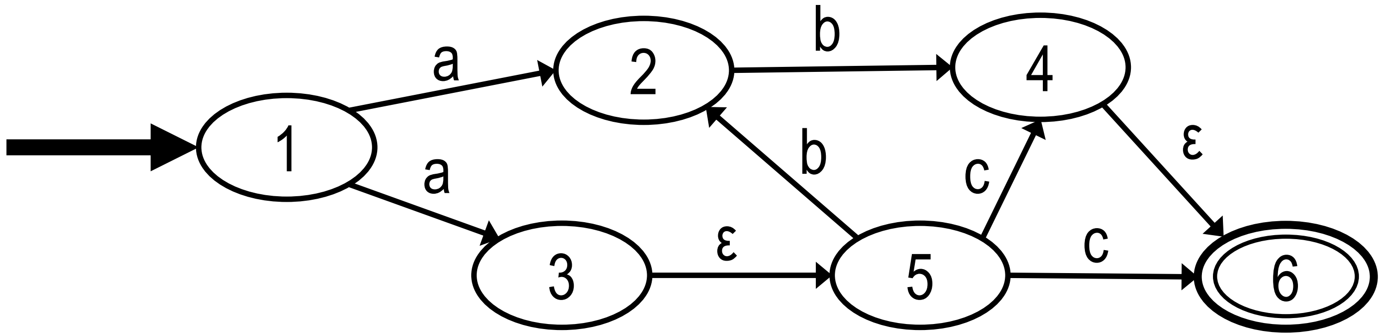
set of NFA states reachable from any $s \in S$ by an ε step

Main calculation:

$$\text{DFA-edge}(D,a) = \text{closure}\left(\bigcup_{s \in D} \text{edge}(s,a) \right)$$

set of NFA states reachable from D by making one a step and (then) any number of ε steps

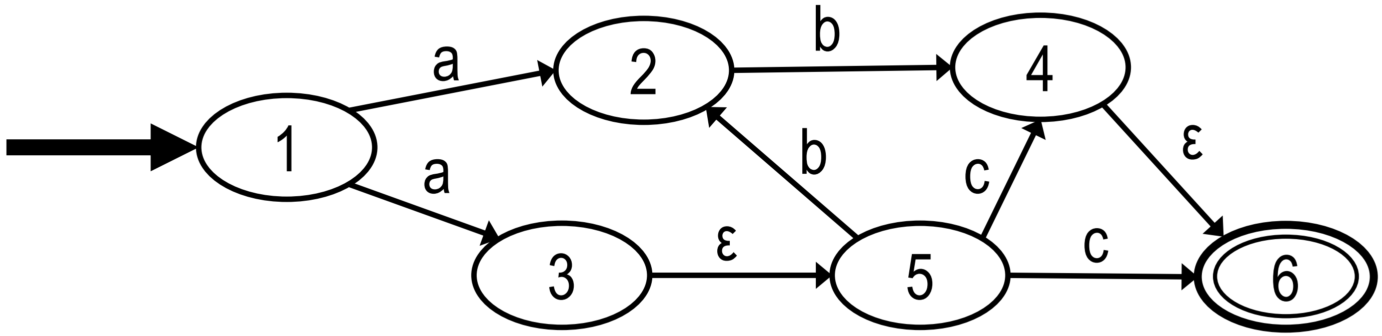
NFA to DFA Example



$\boxed{\text{closure}(S)}$

$$S \cup \left(\bigcup_{s \in S} \text{edge}(s, \varepsilon) \right)$$

NFA to DFA Example



Step 1: closure sets

1 : {1}
2:{2}
3:{3,5}
4:{4,6}
5:{5}
6:{6}

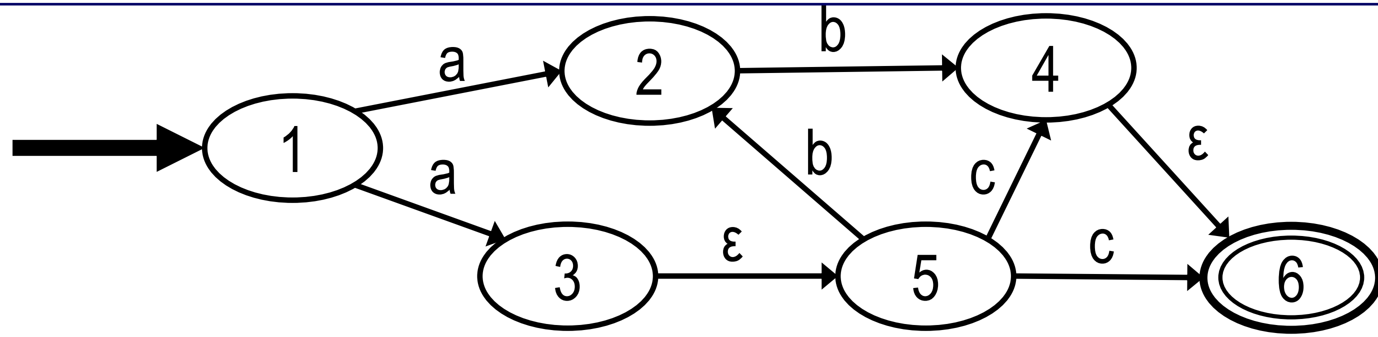
closure(S)

$$S \cup \left(\bigcup_{s \in S} \text{edge}(s, \varepsilon) \right)$$

edge(s, a)

$$\{ t \mid s \xrightarrow{a} t \}$$

NFA to DFA Example



Step 1: closure sets

1	**{1}**
2	{2}
3	{3,5}
4	{4,6}
5	{5}
6	{6}

Step 2: edge sets

	a	b	c
1	2,3	-	-
2	-	4	-
3	-	-	-
4	-	-	-
5	-	2	4,6
6	-	-	-

closure(S)

$S \cup (\bigcup_{s \in S} \text{edge}(s, \varepsilon))$

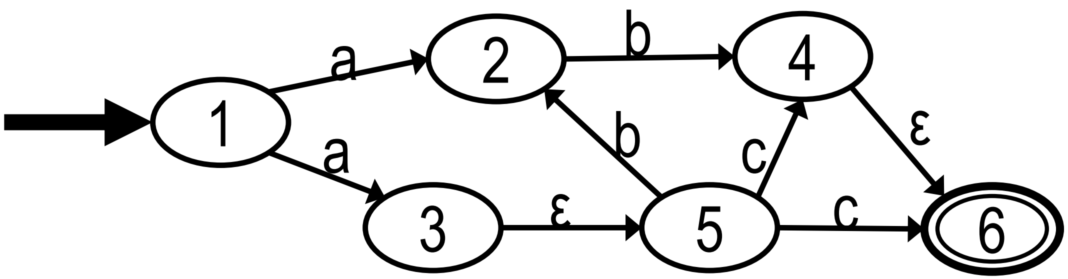
edge(s, a)

$\{ t \mid s \xrightarrow{a} t \}$

DFA-edge(D,a)

$\text{closure}(\bigcup_{s \in D} \text{edge}(s,a))$

NFA to DFA Example



	a	b	c
1	2,3	-	-
2	-	4	-
3	-	-	-
4	-	-	-
5	-	2	4,6
6	-	-	-

Step 3: DFA-sets

D	**a**	**b**	**c**
{1}	Cl(2) + Cl(3) = {2,3,5}	{}	{}
{2,3,5}	{}	Cl(2)+Cl(4) = {2,4,6}	Cl(4) + Cl(6) = {4,6}
{2,4,6}	{}	Cl(4) = {4,6}	{}
{4,6}	{}	{}	{}

1	**{1}**
2	{2}
3	{3,5}
4	{4,6}
5	{5}
6	{6}

closure(S)

$$S \cup \left(\bigcup_{s \in S} \text{edge}(s, \varepsilon) \right)$$

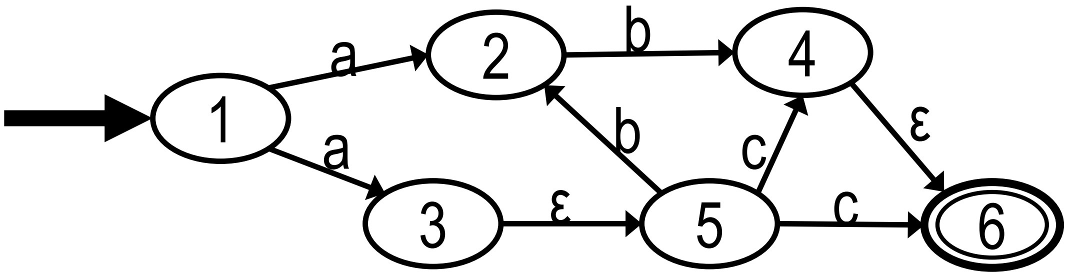
edge(s, a)

$$\{ t \mid s \xrightarrow{a} t \}$$

DFA-edge(D,a)

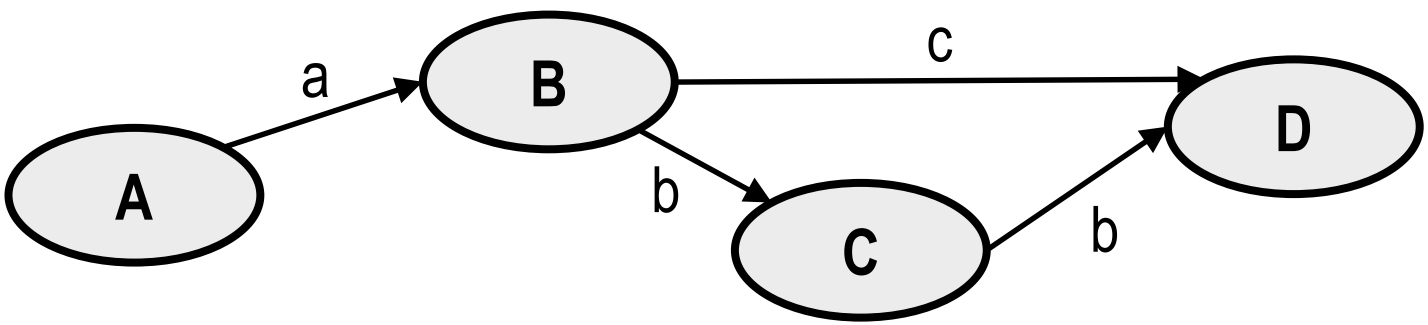
$$\text{closure}\left(\bigcup_{s \in D} \text{edge}(s,a) \right)$$

NFA to DFA Example

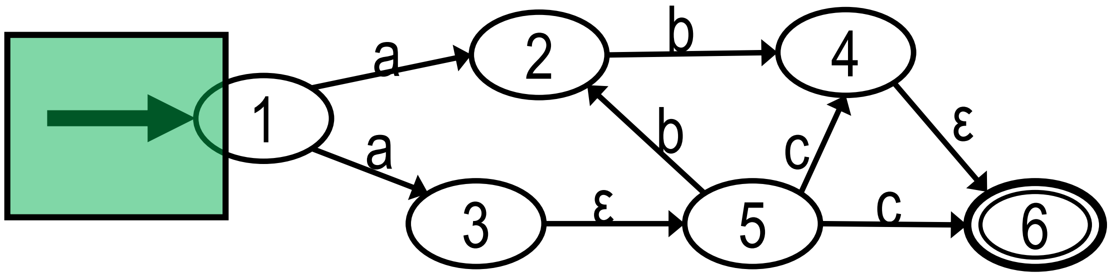


Step 4: Transition matrix

	D	a	b	c
A	{1}	Cl(2) + Cl(3) = {2,3,5}	{}	{}
B	{2,3,5}	{}	Cl(2)+Cl(4) = {2,4,6}	Cl(4) +Cl(6) = {4,6}
C	{2,4,6}	{}	Cl(4) = {4,6}	{}
D	{4,6}	{}	{}	{}



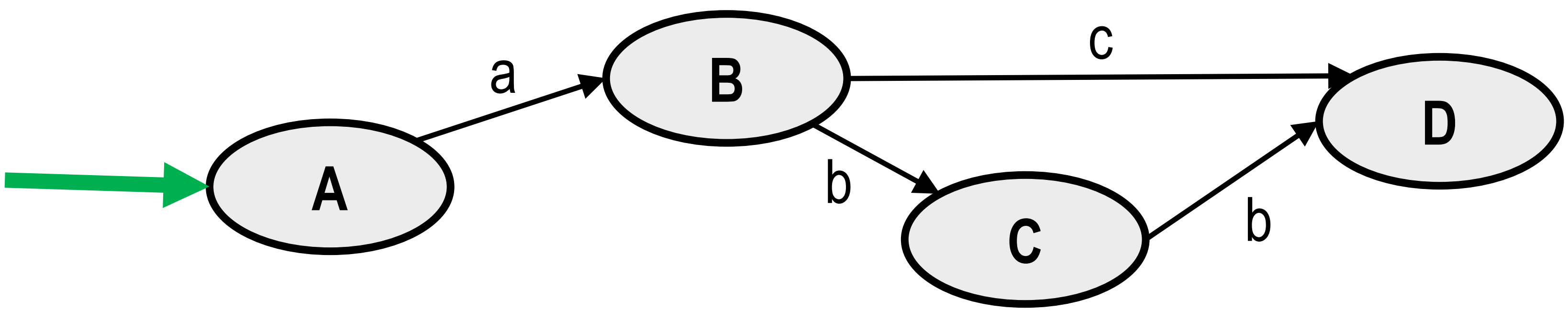
NFA to DFA Example



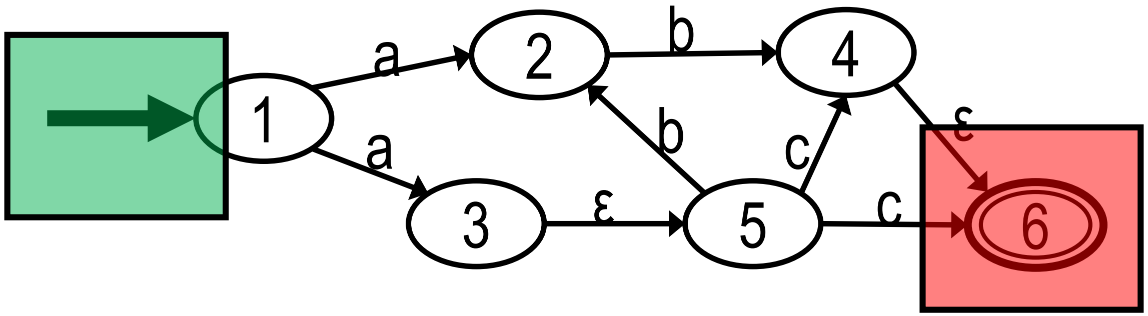
Step 5:

Initial state: closure of initial NFA state

	D	a	b	c
A	{1}	Cl(2) + Cl(3) = {2,3,5}	{}	{}
B	{2,3,5}	{}	Cl(2)+Cl(4) = {2,4,6}	Cl(4) + Cl(6) = {4,6}
C	{2,4,6}	{}	Cl(4) = {4,6}	{}
D	{4,6}	{}	{}	{}

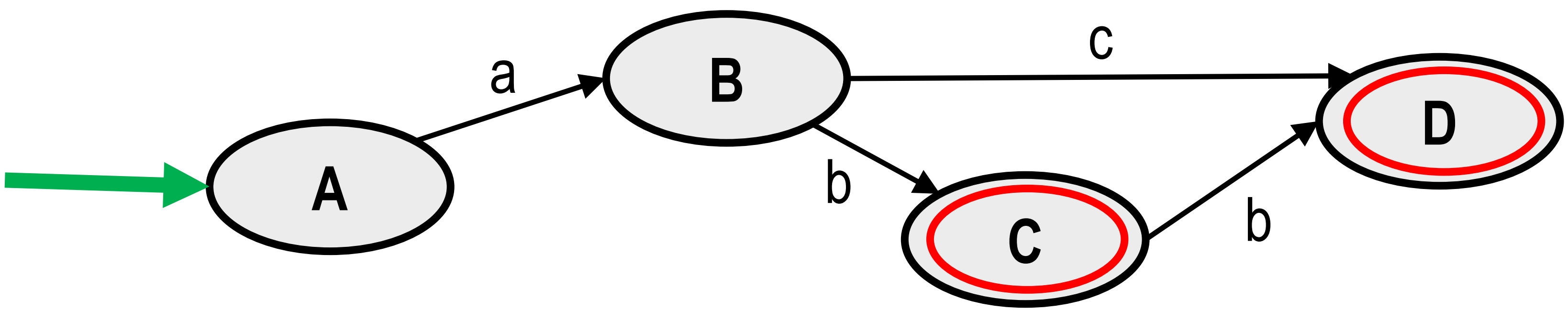


NFA to DFA Example



Step 6:
Final state(s): DFA states "containing" a final NFA state

	D	a	b	c
A	{1}	Cl(2) + Cl(3) = {2,3,5}	{}	{}
B	{2,3,5}	{}	Cl(2)+Cl(4) = {2,4,6}	Cl(4) + Cl(6) = {4,6}
C	{2,4,**6**}	{}	Cl(4) = {4,6}	{}
D	{4,**6**}	{}	{}	{}



 Algorithm in pseudo-code: Appel, page 27

The Longest Token

Lexer should identify the **longest matching** token:

ifz8 should be lexed as IDENTIFIER, not as two tokens IF, IDENTIFIER

Hence, the implementation

- saves the most recently encountered accepting state of the DFA (and the corresponding stream position) and
- updates this info when passing through another accepting state
- Uses the order of rules as tie-breaker in case several tokens (of equal length) match

Other Useful Techniques

Read Chapters 1 and 2.

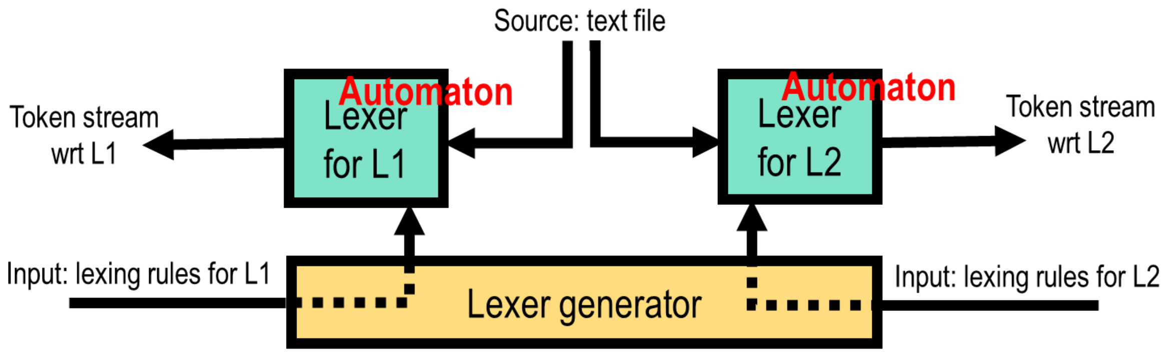
Equivalent states:

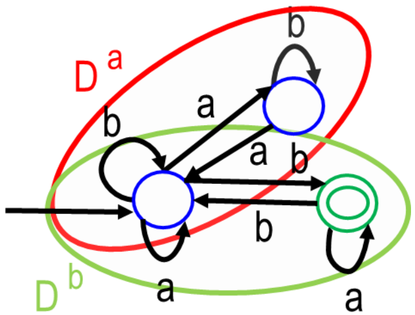
- Eliminate redundant states, smaller FA.
- Do Exercise 2.6 (hand in optional).

FA $\rightarrow$ RE:

- Useful to confirm correct RE $\rightarrow$ FA. (see exercise 2.7)
- GNFAs! (generalized NFA's: transitions may be labeled with RE's)
- See: *Introduction to the Theory of Computation* by Michael Sipser

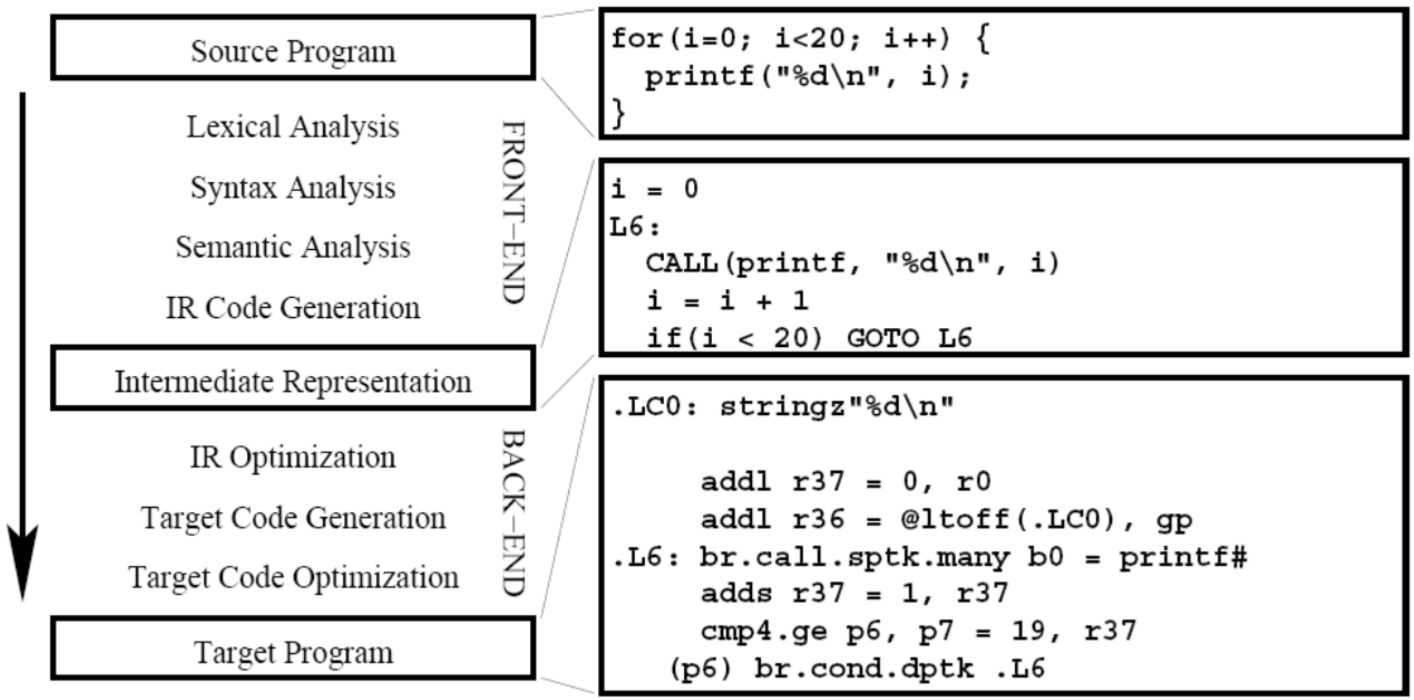
Summary





- Motivated use of lexer generators for partitioning input stream into tokens
- Three formalisms for describing and implementing lexers:
  - Regular expressions
  - NFA's
  - DFA's
  - Conversions RE -> NFA -> DFA
- Next lecture: practicalities of lexing (ML-LEX)

The Compiler



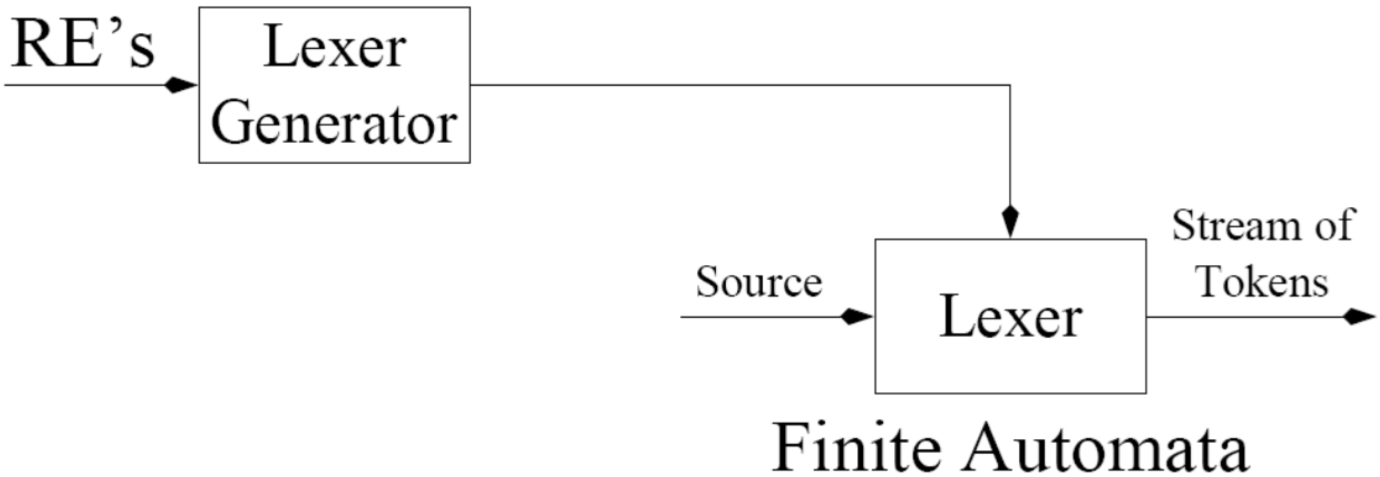
- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

Practicalities of lexing: ML Lex, Lex, Flex, …

The first phase of a compiler is called the **Lexical Analyzer** or **Lexer**.

Implementation Options:

1. Write Lexer from scratch.
2. Use Lexical Analyzer Generator.



- **ml-lex** is a lexical analyzer generator for ML.
- **lex** and **flex** are lexical analyzer generators for C.

ML Lex: lexer generator for ML (similar tools for C: lex, flex)

- Input to **ml-lex** is a set of *rules* specifying a lexical analyzer.
- Output from **ml-lex** is a lexical analyzer in ML.
- A *rule* consists of a pattern and an *action*:
  - *Pattern* is a *regular expression*.
  - *Action* is a fragment of ordinary *ML code*. (Typically returns a token type to calling function.)
- Examples:

```
if => (print("Found token IF"));
[0-9]+ => (print("Found token NUM"));
```

- General Idea: When prefix of input matches a pattern, the action is executed.

Lexical Specification

Specification of a lexer has three parts:

User Declarations
%%
ML-LEX Definitions
%%
Rules

User declarations:

- definitions of values to be used in the action fragments of rules
- Two values **must** be defined in the section:
  - type lexresult: type of the value returned by the rule actions
  - fun eof(): function to be called by the generated lexer when end of input stream is reached (eg call parser, print “done”)

Lexical Specification

Specification of a lexer has three parts:

User Declarations
%%
ML-LEX Definitions
%%
Rules

ML-LEX Definitions:

- definitions of regular expressions abbreviations:

  DIGITS=[0..9]+;
  LETTER = [a-zA-Z];

- definitions of start states to permit multiple lexers to run together:

  %s STATE1 STATE2 STATE3;

  Example: entering "comment" mode, e.g. for supporting nested comments

Lexical Specification

Specification of a lexer has three parts:

User Declarations
%%
ML-LEX Definitions
%%
Rules

optional, states must be defined in ML-LEX section

reg.expr

ML expression (eg construct a token and return it to the invoking function)

Rules:

- format: <start-state-list> pattern => (action_code);
- Intuitive reading: if you're in state **mode**, lex strings matching the **pattern** as described by the **action**.

Rule Patterns

<table>
<tr><th>symbol</th><th>matches</th></tr>
<tr><td>a</td><td>individual character “a” (not for reserved chars ?,*,+,[,{)</td></tr>
<tr><td>\{</td><td>reserved character {</td></tr>
<tr><td>[abc]</td><td>a | b | c</td></tr>
<tr><td>[a-zA-Z]</td><td>lowercase and capital letters</td></tr>
<tr><td>.</td><td>any character except new line</td></tr>
<tr><td>\n</td><td>newline</td></tr>
<tr><td>\t</td><td>tab</td></tr>
<tr><td>“abc?”</td><td>abc? taken literally (reserved chars as well)</td></tr>
<tr><td>{LETTER}</td><td>Use abbreviation LETTER defined in ML-LEX Definitions</td></tr>
<tr><td>a*</td><td>0 or more a’s</td></tr>
<tr><td>a+</td><td>1 or more a’s</td></tr>
<tr><td>a?</td><td>0 or 1 a</td></tr>
<tr><td>a|b</td><td>a or b</td></tr>
</table>

```
if|iff => (print("Found token IF or IFF"));
[0-9]+ => (print("Found token NUM"));
```

Rule Actions

- Actions can use various values defined in User Declarations section.
- Two values always available:

  `type lexresult`
  - type of the value returned by each rule action.

  `fun eof()`
  - called by lexer when end of input stream reached.

- Several special variables also available to action fragments.
  - `yytext` - input substring matched by regular expression.
  - `yypos` - file position of beginning of matched string.
  - `continue()` - recursively calls lexing engine.

Start States

- *Start states* permit multiple lexical analyzers to run together.
- Rules prefixed with a start state is matched only when lexer is in that state.
- States are entered with `YYBEGIN`.
- Example:

```
%%
%s COMMENT
%%
<INITIAL> if => (print("Token IF"));
<INITIAL> [a-z]+ => (print("Token ID"));
<INITIAL> "(*" => (YYBEGIN COMMENT; continue());
<COMMENT> "*)" => (YYBEGIN INITIAL; continue());
<COMMENT> "\n"|.  => (continue());
```

Rule Matching and Start States

```
<start_state_list> regular_expression => (action_code);
```

- Regular expression matched only if lexer is in one of the start states in start state list.
- If no start state list specified, the rule matches in all states.
- Lexer begins in predefined start state: `INITIAL`

If multiple rules match in current start state, use Rule Disambiguation.

Rule Disambiguation

- *Longest match* - longest initial substring of input that matches regular expression is taken as next token.

  `if8` matches `ID(''if8'')`, not `IF()` and `NUM(8)`.

- *Rule priority* - for a particular substring which matches more than one regular expression with equal length, choose first regular expression in rules section.

  If we want `if` to match `IF()`, not `ID(''if'')`, put keyword regular expression before identifier regular expression.

Example

```
(* -*- ml -*- *)
type lexresult = string
fun eof() = (print("End-of-file\n"); "EOF")

%%

INT=[1-9][0-9]*;

%s COMMENT;

%%

<INITIAL>"/*"          => (YYBEGIN COMMENT; continue());
<COMMENT>"*/"          => (YYBEGIN INITIAL; continue());
<COMMENT>"\n"|.        => (continue());

<INITIAL>if            => (print("Token IF\n");"IF");
<INITIAL>then          => (print("Token THEN\n");"THEN");
<INITIAL>{INT}         => (print("Token INT(" ^ yytext ^ ")\n");"INT");
<INITIAL>" "|"\n"|"\t" => (continue());
<INITIAL>.             => (print("ERR: '" ^ yytext ^ "'.\n");"ERR");
```

Example in Action

```
% cat x.txt

if 999 then 0999
/* This is a comment 099 if */
if 12 then 12

% sml
Standard ML of New Jersey, Version 109.33, November 21, 1997 [CM; ...]
- CM.make();
[.....]
val it = () : unit
- MyLexer.tokenize("x.txt");

Token IF
Token INT(999)
Token THEN
ERR: '0'.
Token INT(999)
Token IF
Token INT(12)
Token THEN
Token INT(12)
End-of-file
val it = () : unit
```