Lexical Analysis : Part 1

Lecture 2

Token

Token Type

Examples: ID, NUM, IF, EQUALS, ...

Lexeme

The characters actually matched.

Example:

```
    ... if x == -12.30 then ...
```

Token

Token Type

Examples: ID, NUM, IF, EQUALS, ...

Lexeme

The characters actually matched.

Example:

```
    ... if x == -12.30 then ...
```

How to describe/specify tokens?

Formal:

Regular Expressions

```
                    Letter ( Letter | Digit )*
```

Informal:

"// through end of line"

Token

How to describe/specify tokens?

Formal:

Regular Expressions

`Letter ( Letter | Digit )*`

Informal:

"// through end of line"

Tokens will appear as TERMINALS in the grammar.

Stmt → **while Expr do StmtList end While**

→ ID **"=" Expr ";"**

→

Lexical Error

Most errors tend to be "typos"
Not noticed by the programmer

```
return 1.23;
retunn 1,23;
```

... Still results in sequence of legal tokens

```
<ID,"retunn"> <INT,1> <COMMA> <INT,23> <SEMICOLON>
```

No lexical error, but problems during parsing!

Lexical Error

Most errors tend to be "typos"
Not noticed by the programmer

```
return 1.23;
retunn 1,23;
```

... Still results in sequence of legal tokens

```
<ID,"retunn"> <INT,1> <COMMA> <INT,23> <SEMICOLON>
```

No lexical error, but problems during parsing!

Errors caught by lexer:

- EOF within a String / missing "
- Invalid ASCII character in file
- String / ID exceeds maximum length
- Numerical overflow

etc...

Lexical Error

Errors caught by lexer:

- EOF within a String / missing ”
- Invalid ASCII character in file
- String / ID exceeds maximum length
- Numerical overflow

etc...

Lexer must keep going!

- Always return a valid token.
- Skip characters, if necessary.
- May confuse the parser
- The parser will detect syntax errors and get straightened out (hopefully!)

Managing Input Buffer

Option 1: Read one char from OS at a time.
Option 2: Read N characters per system call

e.g., N = 4096

Manage input buffers in Lexer
More efficient

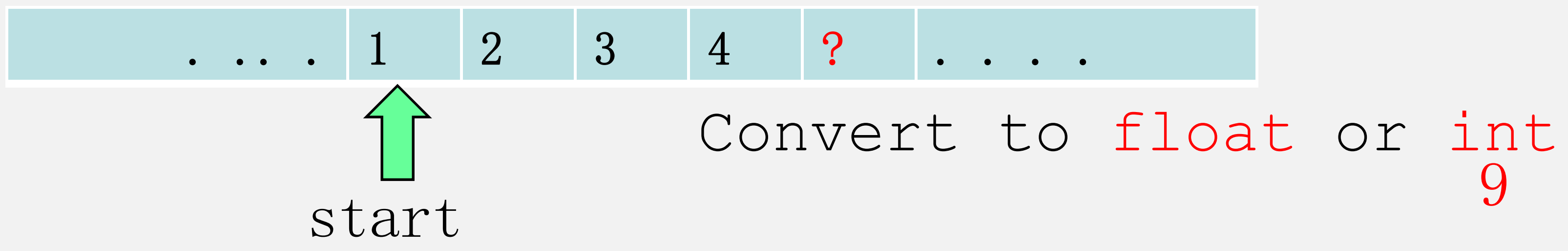
Managing Input Buffer

Option 1: Read one char from OS at a time.
Option 2: Read N characters per system call

e.g., N = 4096

Manage input buffers in Lexer
More efficient

Often, we need to look ahead



Managing Input Buffer

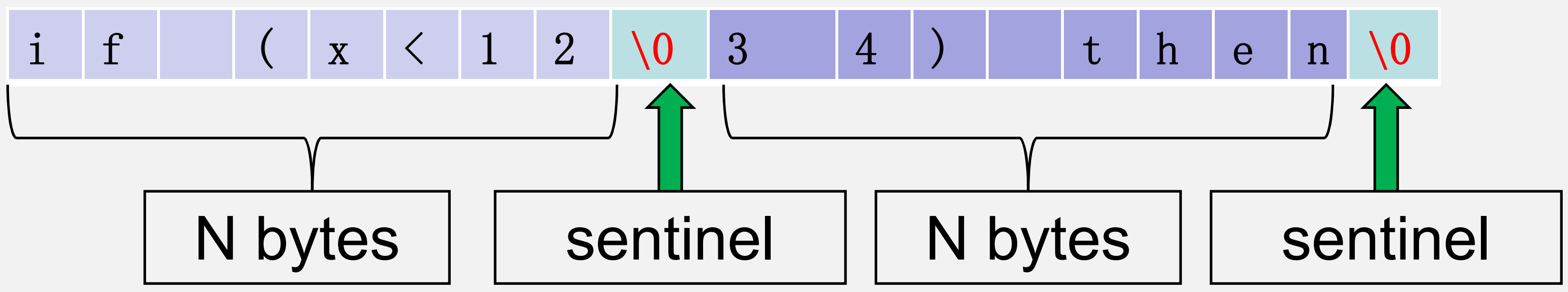
Token could overlap / span buffer boundaries.
- need 2 buffers

Code:
```
if (ptr at end of buffer1) or (ptr at end of buffer2) then ...
```

Technique: Use "Sentinels" to reduce testing
Choose some character that occurs rarely in most inputs
`'\0'`

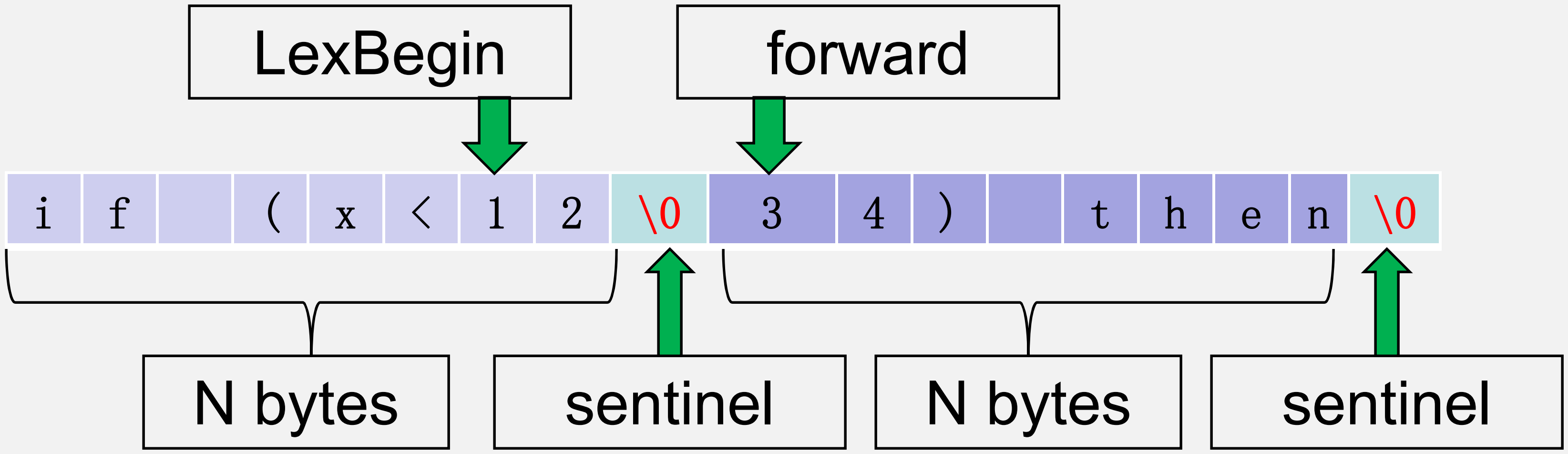
Managing Input Buffer

Technique: Use “Sentinels” to reduce testing
Choose some character that occurs rarely in most inputs
`'\0'`



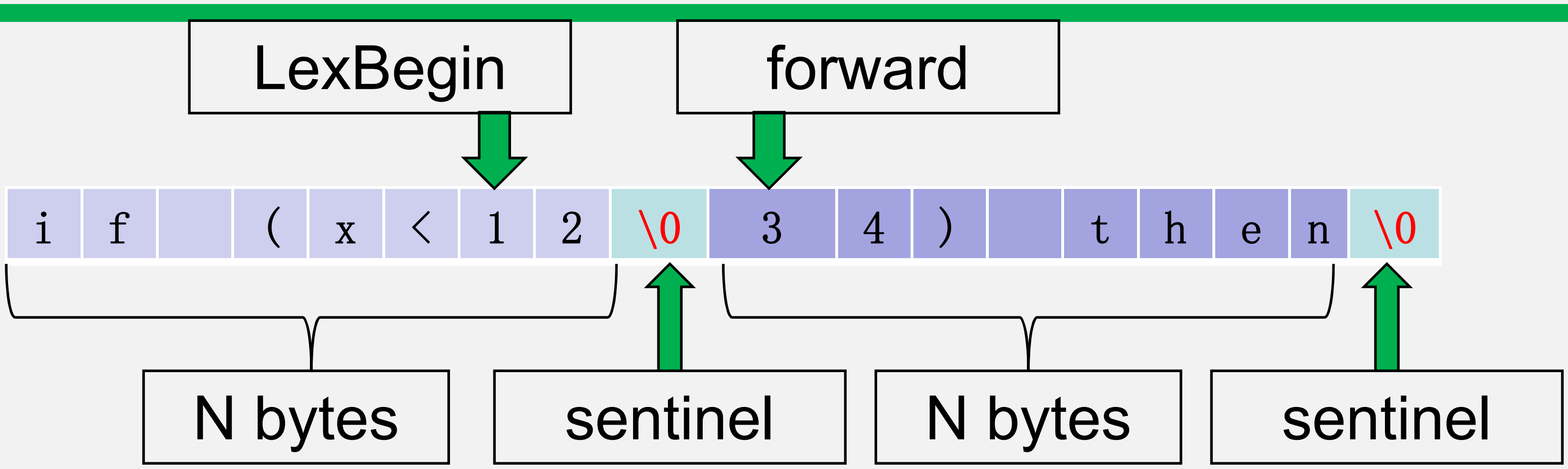
Managing Input Buffer

Technique: Use "Sentinels" to reduce testing
Choose some character that occurs rarely in most inputs
`'\0'`



Goal: Advance forward pointer to next character
...and reload buffer if necessary.

Managing Input Buffer



Goal: Advance forward pointer to next character

...and reload buffer if necessary.

```
Code :
     forward++;
        if *forward == '\0' then
        if forward at end of buffer #1 then
             Read next N bytes into buffer #2;
             forward = address of first char of buffer #2;
        elseIf forward at end of buffer #2 then
             Read next N bytes into buffer #1;
             forward = address of first char of buffer #1;
        else
            // do nothing; a real \0 occurs in the input
    endIf
endIf
```

Some definition

Alphabet **(Σ)**

A set of symbols (“characters”)

Examples: `Σ = { a, b, c, d }`

`Σ = ASCII character set`

String **(or “Sentence”)**

Sequence of symbols

Finite in length

Example: **`abbadc`** **`Length of s = |s|`**

Some definition

Empty String (ε, "epsilon")

It is a string

|**ε**| = 0

Language

A set of strings

Examples: *L1* ***= { a, baa, bccb }***

L**2 = { }**

L**3 = {ε }**

L**4 = {ε, ab, abab, ababab, abababab,... }**

L**5 = { s |** s can be interpreted as an English sentence making a true statement about mathematics}

Each string is finite in length, but the set may have an infinite number of elements.

Some definition

Prefix ...of string s

```
s = hello   Prefixes:   he
                        hello
                        ε
```

Suffix ...of string s

```
s = hello   Prefixes:   llo
                        ε
                        hello
```

Some definition

Suffix ...of string s

```
        s = hello  Suffixes:   llo
                               ε
                               hello
```

Substring ...of string s

```
        s = hello  Substring:  ell
                               hello
                               ε
```

Proper prefix / suffix / substring ... of s

```
        ≠s and ≠ε
```

Some definition

Substring ...of string s

```
      s = hello Substring: ell
                           hello
                           ε
```

Proper prefix / suffix / substring ... of s

```
      ≠s and ≠ε
```

Subsequenc ...of string s,

```
      s = compilers Subsequences:   opilr
                                    cors
                                    compi
                                    ε
```

Some definition

Concatenation

Strings: x, y
Concatenation: xy

Example:

$x = $ **abb**
$y = $ **cdc**
$xy = $ **abbcdc**
$yx = $ **cdcabb**

Other notations:

$x \,||\, y$
$x + y$
$x \mathbin{++} y$
$x \cdot y$

Some definition

Concatenation

Strings: x, y
Concatenation: xy

Example:

x = **abb**
y = **cdc**
xy = **abbcdc**
yx = **cdcabb**

What is the “identity” for concatenation?

$$\varepsilon x = x\varepsilon = x$$

Multiplication & Concatenation
Exponentiation & ?

Define $s^0 = \varepsilon$

$$s^N = s^{N-1}s$$

Example x = **ab**

$x^0 = \varepsilon$
$x^1 = \mathbf{x}$ = **ab**
$x^2 = \mathbf{xx}$ = **abab**
$x^3 = \mathbf{xxx}$ = **ababab**
...etc...
$x^\infty = \mathbf{xxx}$ = **ababab. . .**

Some definition

Language

A set of strings

```
L = { ... }
M = { ... }
```

Generally, these are infinite sets.

Some definition

Language

A set of strings

```
L = { ... }
M = { ... }
```

Generally, these are infinite sets.

Union of two languages

```
L U M = { s | s is in L or is in M }
```

Example:

```
L = { a, ab }
M = { c, dd }

L U M = { a, ab, c, dd }
```

Some definition

Union of two languages

`L U M = { s | s is in L or is in M }`

Example:

```
        L = { a, ab }
        M = { c, dd }
        L U M = { a, ab, c, dd }
```

Concatenation of two languages

`L M = { st | s ϵ L and t ϵ M }`

Example:

```
        L = { a, ab }
        M = { c, dd }
        L M = { ac, add, abc, abdd }
```

Repeated Concatenation

Let: L = **{ a, bc }**

Example:

$L^0 = \{\varepsilon\}$

$L^1 = L = \{ a, bc \}$

$L^2 = LL = \{ aa, abc, bca, bcbc \}$

$L^3 = LLL = \{ aaa, aabc, abca, abcbc,$
$\quad bcaa, bcabc, bcbca, bcbcbc \}$

...etc...

$L^N = L^{N-1}L = LL^{N-1}$

Kleene Closure

The “Kleene Closure” of a language:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \; U \; L^1 \; U \; L^2 \; U \; \ldots.$$

Example:

$$L^* = \{ \underbrace{\varepsilon}_{L^0}, \underbrace{a, bc}_{L^1}, \underbrace{aa, abc, bca, bcbc}_{L^2}, \underbrace{aaa, aabc, abca, abcbc}_{L^3}, \ldots \}$$

Positive Kleene Closure

The “ Kleene Closure” of a language:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \; U \; L^1 \; U \; L^2 \; U \; \ldots.$$

$$L^* = \{ \underbrace{\varepsilon}_{L^0}, \underbrace{a, bc}_{L^1}, \underbrace{aa, abc, bca, bcbc}_{L^2}, \underbrace{aaa, aabc, abca, abcbc}_{L^3}, \ldots \}$$

The “ Positive Kleene Closure” of a language:

$$L^+ = \bigcup_{i=0}^{\infty} L^i = L^1 \; U \; L^2 \; U \; L^3 \ldots.$$

$$L^+ = \{ \underbrace{a, bc}_{L^1}, \underbrace{aa, abc, bca, bcbc}_{L^2}, \underbrace{aaa, aabc, abca, abcbc}_{L^3}, \ldots \}$$

Examples

Let: L = { **a, b, c, ..., z** }

D = { **0, 1, 2, ..., 9** }

D^+ **=**

Examples

Let: L = { **a, b, c, ..., z** }
D = { **0, 1, 2, ..., 9** }

D^{+} **=** "The set of strings with one or more digits"

L U D =

Examples

Let: $L = \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots, \mathbf{z} \}$

$D = \{ \mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{9} \}$

D^+ **=** "The set of strings with one or more digits"

$L \cup D$ = "The set of alphanumeric characters"

= $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots, \mathbf{z}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{9}\}$

$(L \cup D)^* =$

Examples

Let: L = { **a, b, c, ..., z** }
D = { **0, 1, 2, ..., 9** }

D^+ **=** "The set of strings with one or more digits"

L U D = "The set of alphanumeric characters"
= {**a, b, c, ..., z, 0, 1, 2, ..., 9**}

(L U D)* = "Sequences of zero or more letters and digits"

L(L U D)* = "Set of strings that start with a letter, followed by zero or more letters and and digits. "

How to Parse Regular Expression

Assume the alphabet is given... *e.g., Σ =* ***{ a, b, c, ... z }***

`Example:` **`a ( b | c ) d* e`**

A regular expression describes a language.

Notation:

r = regular expression

L(r) = the corresponding language

Example:

r = a (b | c) d* e

L(r) = { abe, abde, abdde, abddde, . . . , ace, acde, acdde, acddde, . . . }

Regular Expression

- * has highest precedence.
- Concatenation as middle precedence.
- | has lowest precedence.
- Use parentheses to override these rules.

Examples:

`a b* = a(b*)`

If you want `(ab)*` you must use parentheses.

`a | b c = a | (b c)`

If you want (a | b) c you must use parentheses.

Concatenation and | are associative.

`(a b) c = a (b c) = a b c`

`(a | b) | c = a | (b | c) = a | b | c`

Example:

`bd | ef*| ga = b) | e(f*) | (ga)`

Regular Expression

- * has highest precedence.
- Concatenation as middle precedence.
- | has lowest precedence.
- Use parentheses to override these rules.

Examples:

`a b* = a(b*)`

If you want `(ab)*` you must use parentheses.

`a | b c = a | (b c)`

If you want (a | b) c you must use parentheses.

Concatenation and | are associative.

`(a b) c = a (b c) = a b c`

`(a | b) | c = a | (b | c) = a | b | c`

Example:

`bd | ef*| ga = (bd) | (e(f*)) | (ga)`

Regular Expression

- * has highest precedence.
- Concatenation as middle precedence.
- | has lowest precedence.
- Use parentheses to override these rules.

Examples:

$a\ b^* = a(b^*)$

If you want (ab)* you must use parentheses.

a | b c = a | (b c)

If you want (a | b) c you must use parentheses.

Concatenation and | are associative.

(a b) c = a (b c) = a b c

(a | b) | c = a | (b | c) = a | b | c

Example:

$bd\ |\ ef^*|\ ga = ((bd)\ |\ (e(f^*)))\ |\ (ga)$

Definition : Regular Expression

(Over alphabet Σ)

1. ε is a regular expression.
2. If a is a symbol (i.e., if a∈Σ), then a is a regular expression.
3. If R and S are regular expressions, then R|S is a regular expression.
4. If R and S are regular expressions, then RS is a regular expression.
5. If R is a regular expression, then R* is a regular expression.
6. If R is a regular expression, then (R) is a regular expression.

Definition : Regular Expression

And, given a regular expression R, what is L(R) ?

1. ε is a regular expression.

   L(R) = {ε}

1. If a is a symbol (i.e., if a∊Σ), then a is a regular expression.

   L(a) = { a }

1. If R and S are regular expressions, then R|S is a regular expression.

   L(R|S) = L(R) U L(S)

Definition : Regular Expression

(Over alphabet Σ)

1. If R and S are regular expressions, then RS is a regular expression.

   $L(RS) = L(R)\ L(S)$

1. If R is a regular expression, then R* is a regular expression.

   $L(R^*) = (L(R))^*$

1. If R is a regular expression, then (R) is a regular expression.

   $L((R)) = L(R)$

Regular Language

Definition: “Regular Language” (or “Regular Set”)
... A language that can be described by a regular expression.

- Any finite language (i.e., finite set of strings) is a regular language.
- Regular languages are (usually) infinite.
- Regular languages are, in some sense, simple languages.
- Regular Languages Context-Free Languages

Examples:

a|b|cab	{a, b, cab}
b*	{**ε**, b, bb, bbb, ...}
a|b*	{a, **ε**, b, bb, bbb, ...}
a|b)*	{**ε**, a, b, aa, ab, ba, bb, aaa, ...}
Set of all strings of a’s and b’s, including **ε**.	

Regular Language

Definition: “Regular Language” (or “Regular Set”)
... A language that can be described by a regular expression.

- Any finite language (i.e., finite set of strings) is a regular language.
- Regular languages are (usually) infinite.
- Regular languages are, in some sense, simple languages.
- Regular Languages Context-Free Languages

Examples:

`a|b|cab`	`{a, b, cab}`
`b*`	`{`**ε**`, b, bb, bbb, ...}`
`a|b*`	`{a, `**ε**`, b, bb, bbb, ...}`
`a|b)*`	`{`**ε**`, a, b, aa, ab, ba, bb, aaa, ...}`

`Set of all strings of a’s and b’s, including `**ε**`.`

Algebraic Laws of RE

Let R, S, T be regular expressions...

$\mid$ is commutative	$\mid$ is associative
$R \mid S = S \mid R$	$R \mid (S \mid T) = (R \mid S) \mid T = R \mid S \mid T$
Concatenation is associative	Concatenation distributes over $\mid$
$R\ (S\ T) = (R\ S)\ T = R\ S\ T$	$R\ (S \mid T) = RS \mid RT$
	$(R \mid S)\ T = RT \mid ST$
* is idempotent	
$(R^*)^* = R^*$	ε is the identity for concatenation
	$\varepsilon\ R = R\ \varepsilon = R$
Relation between * and ε	
$R^* = (R \mid \varepsilon)^*$	

Regular Definition

```
Letter = a | b | c | ... | z
Digit = 0 | 1 | 2 | ... | 9

ID = Letter ( Letter | Digit )*
```

Names (e.g., Letter) are underlined to distinguish from a sequence of symbols.

```
Letter ( Letter | Digit )*
= {"Letter", "LetterLetter", "LetterDigit",
... }
```

Regular Definition

```
Letter = a | b | c | ... | z
Digit = 0 | 1 | 2 | ... | 9

ID = Letter ( Letter | Digit )*
```

Each definition may only use names *previously* defined.
- No recursion

Regular Sets = no recursion
CFG = recursion

Addition Notation/Shorthand

One-or-more: **+**

```
X+ = X(X*)
Digit+ = Digit Digit* = Digits
```

Optional (zero-or-one): ***?***

```
x? = (x |ε)
Num = Digit+ (.Digit+)?
```

Character Classes: **[FirstChar-LastChar]**

Assumption: The underlying alphabet is known ...and is ordered.

```
Digit = [0-9]
Letter = [a-zA-Z] = [A-Za-z]
```

Addition Notation/Shorthand

Character Classes: **[FirstChar-LastChar]**
Assumption: The underlying alphabet is known ...and is ordered.

```
Digit = [0-9]
Letter = [a-zA-Z] = [A-Za-z]
```

Variations:

Zero-or-more	: $ab^{*}c$ = a{b}c = a{b}*c
One-or-more	: $ab^{+}c$ = $a\{b\}^{+}c$
Optional	: ab?c = a[b]c

Addition Notation/Shorthand

Many sets of strings are not regular.

...no regular expression for them!

The set of all strings in which parentheses are balanced.

`(()(()))`

Must use a CFG!

Strings with repeated substrings

`{ XcX | X is a string of a's and b's }`

`a b b b a b c a b b b a b`

CFG is not even powerful enough.

Describe/Recognize Token

Problem: How to describe tokens?
Solution: Regular Expressions

Problem: How to recognize tokens?

Approaches:

- Hand-coded routines
  Examples: E-Language, PCAT-Lexer
- Finite State Automata
- Scanner Generators (Java: JLex, C: Lex)

Scanner Generators

Scanner Generators

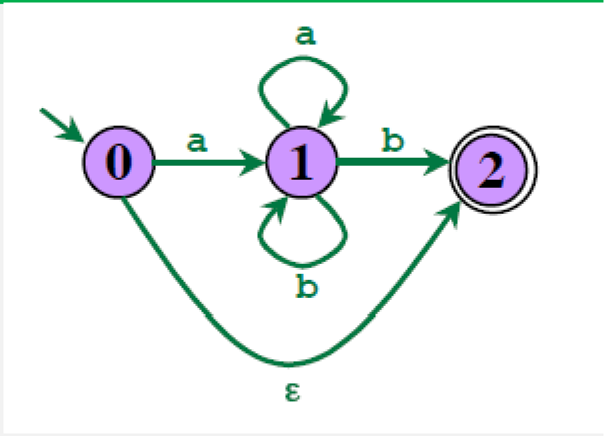
Input: **Sequence of regular definitions**
Output: **A lexer (e.g., a program in Java or "C")**

Approach:

- Read in regular expressions
- Convert into a Finite State Automaton (FSA)
- Optimize the FSA
- Represent the FSA with tables / arrays
- Generate a table-driven lexer (Combine "canned" code with tables.)

Finite State Automata (FSA)

- One start state
- Many final states
- Each state is labeled with a state name
- Directed edges, labeled with symbols



- Deterministic (DFA)

  No **ε-edges**

  Each outgoing edge has different symbol

- Non-deterministic (NFA)

Finite State Automata (FSA)

Formalism: **< S, Σ, δ, S_0, S_F >**

S = Set of states

$S = \{s_0, s_1, \ldots, s_N\}$

Σ = Input Alphabet

Σ = ASCII Characters

δ= Transition Function

S X Σ → States (deterministic)

S X Σ → Sets of States (non-deterministic)

s_0 = Start State

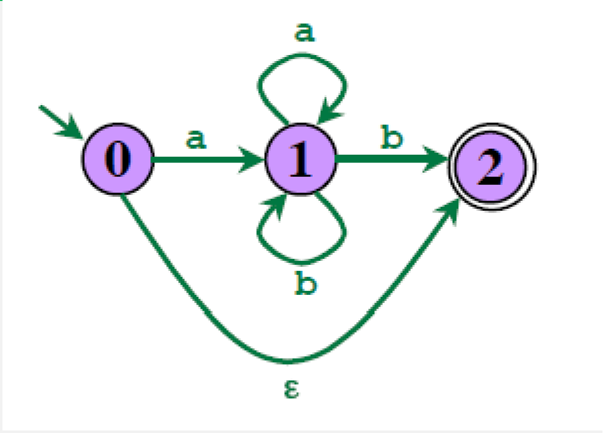
“Initial state”

$s_0 \in S$

S_F = Set of final states

“accepting states”

$S_F \subseteq S$



Finite State Automata (FSA)

Formalism: **< S, Σ, δ, S_0, S_F >**

S = Set of states

$S = \{s_0, \mathbf{s_1}, \ldots, \mathbf{s_N}\}$

Σ = Input Alphabet

Σ = ASCII Characters

δ= Transition Function

S X Σ → States (deterministic)

S X Σ → Sets of States (non-deter

s_0 = Start State

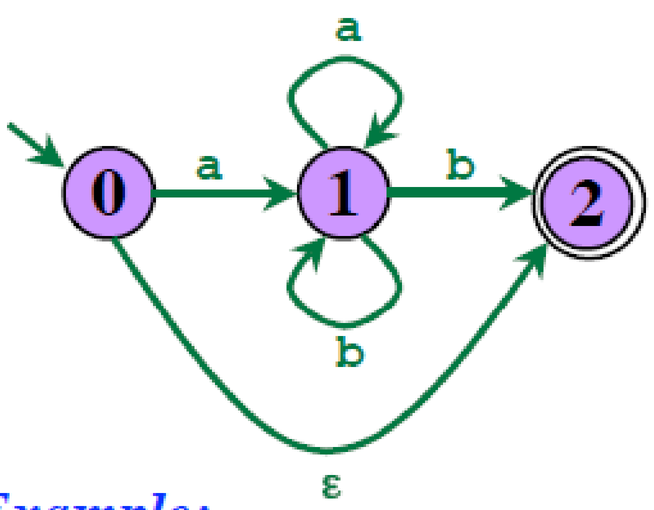
"Initial state"

$s_0 \in \mathbf{S}$

S_F = Set of final states

"accepting states"

$S_F \subseteq S$



Example:

$S = \{0, 1, 2\}$

$\Sigma = \{a, b\}$

$s_0 = 0$

$S_F = \{ 2 \}$

$\delta =$

Input Symbols

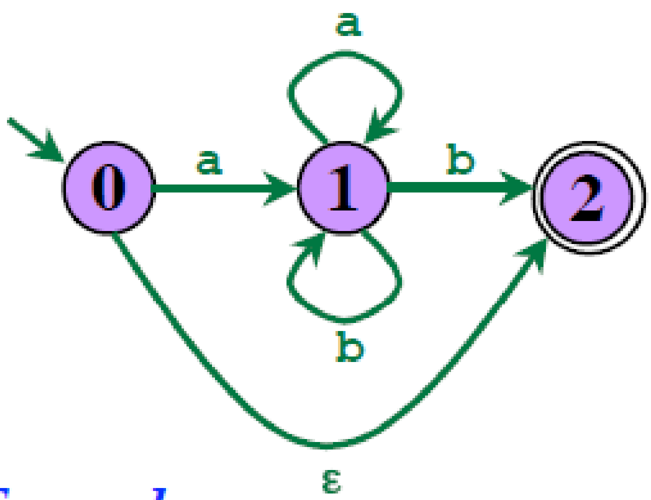
States	a	b	ε
0	{1}	{}	{2}
1	{1}	{1,2}	{}
2	{}	{}	{}

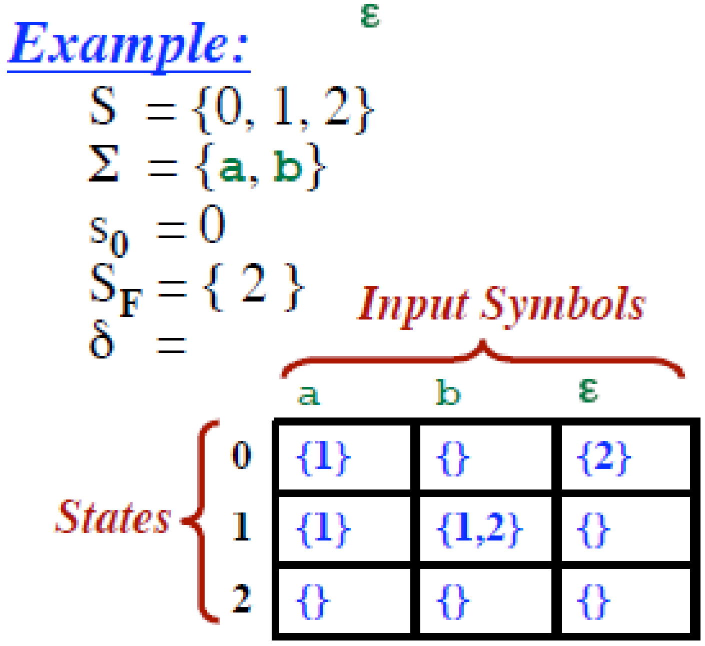
Finite State Automata (FSA)

A string is "accepted"...
(a string is "recognized"...)
by a FSA if there is a path
from Start to any accepting state
where edge labels match the string.

Example:
This FSA accepts:

ε
`aaab`
`abbb`





Example:

$S = \{0, 1, 2\}$
$\Sigma = \{a, b\}$
$s_0 = 0$
$S_F = \{ 2 \}$
$\delta =$

Input Symbols

States	a	b	ε
0	{1}	{}	{2}
1	{1}	{1,2}	{}
2	{}	{}	{}

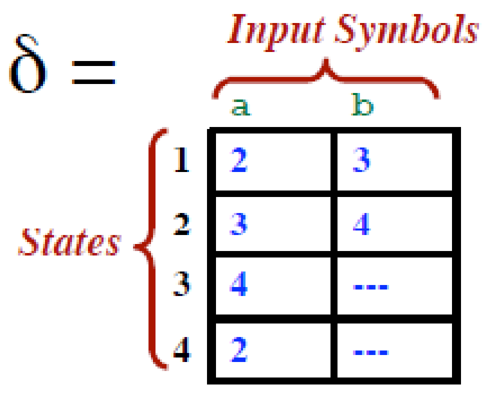
Deterministic Finite State Automata

No ε-**moves**

The transition function returns a single state

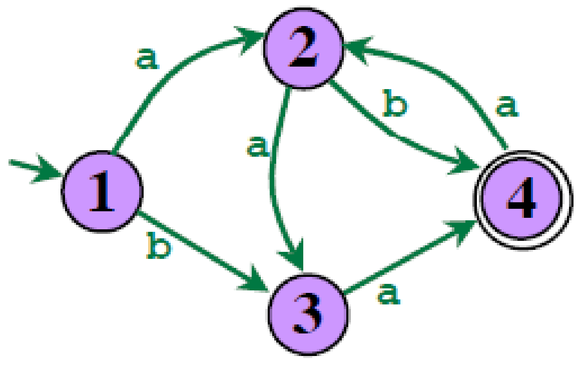
$\delta = S \times \Sigma$

```
function Move (s:State, a:Symbol) returns State
```



$\delta =$ Input Symbols

States	a	b
1	2	3
2	3	4
3	4	---
4	2	---



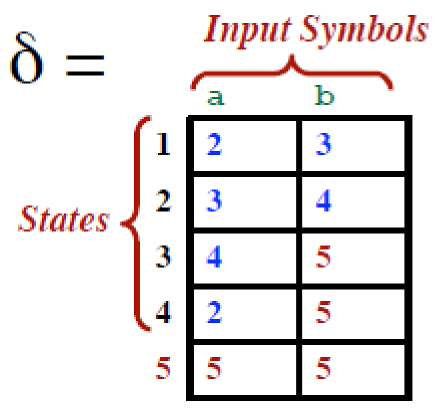
Deterministic Finite State Automata

No ε**-moves**

The transition function returns a single state

$\delta = S \times \Sigma$

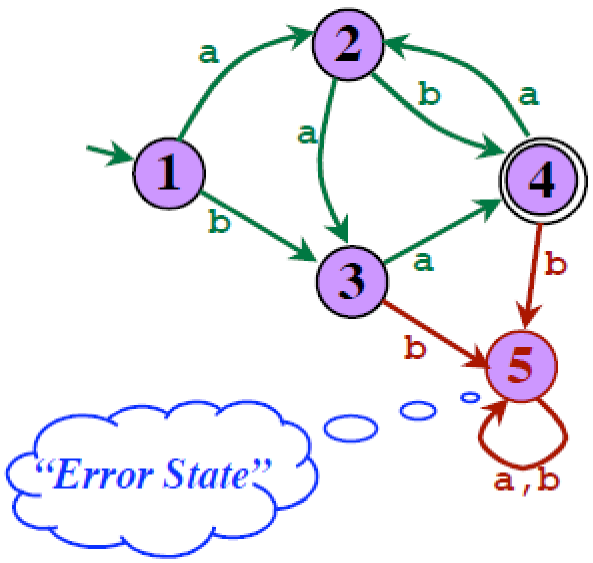
```
function Move (s:State, a:Symbol) returns State
```



$\delta =$

Input Symbols

States	a	b
1	2	3
2	3	4
3	4	5
4	2	5
5	5	5



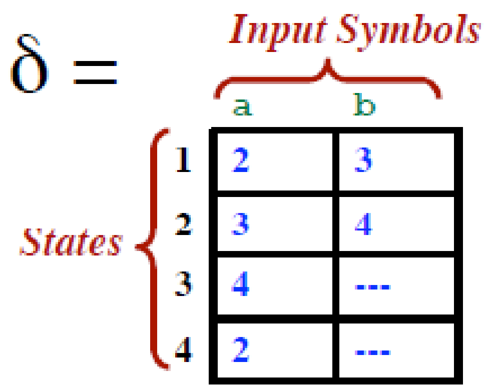
Deterministic Finite State Automata

No ε**-moves**

The transition function returns a single state

$\delta = S \times \Sigma$

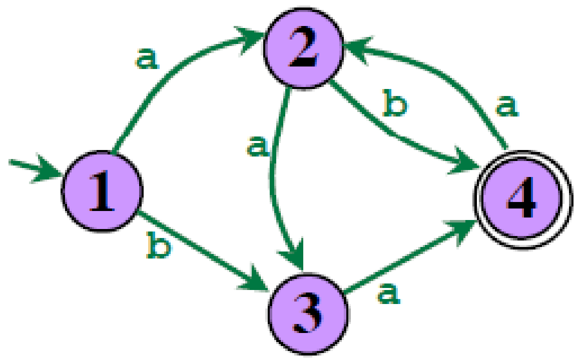
```
function Move (s:State, a:Symbol) returns State
```



$\delta =$

Input Symbols

States	a	b
1	2	3
2	3	4
3	4	---
4	2	---



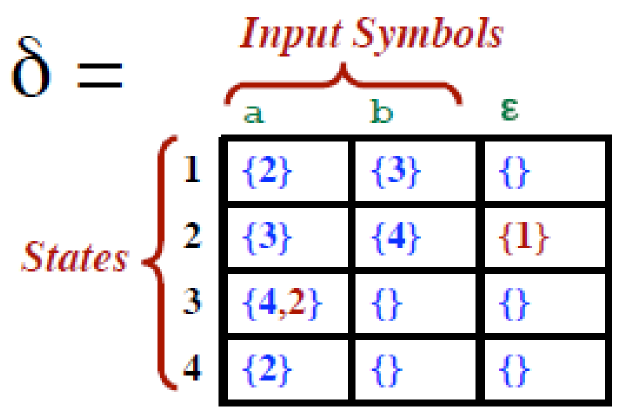
Non Deterministic Finite State Automata

No ε-**moves**

The transition function returns a set of states

$\delta = S \times \Sigma$

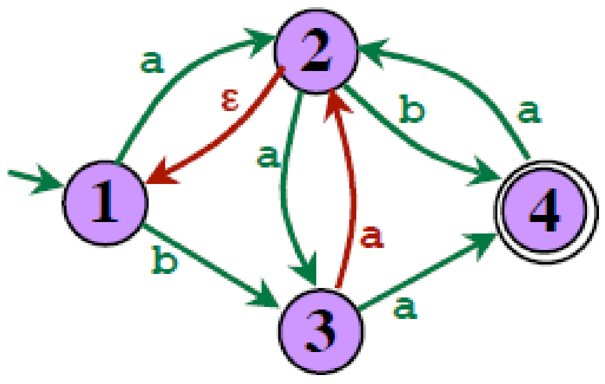
```
function Move (s:State, a:Symbol) returns set of State
```



δ =

Input Symbols

States	a	b	ε
1	{2}	{3}	{}
2	{3}	{4}	{1}
3	{4,2}	{}	{}
4	{2}	{}	{}



Theoretical Results

The set of strings recognized by an NFA
can be described by a Regular Expression.

The set of strings described by a Regular Expression
can be recognized by an NFA.

The set of strings recognized by an DFA
can be described by a Regular Expression.

The set of strings described by a Regular Expression
can be recognized by an DFA.

DFAs, NFAs, and Regular Expressions all have the same “power”.
They describe “Regular Sets” (“Regular Languages”)

The DFA may have a lot more states than the NFA.
(May have exponentially as many states, but...)

Thank You