Lexical Analysis

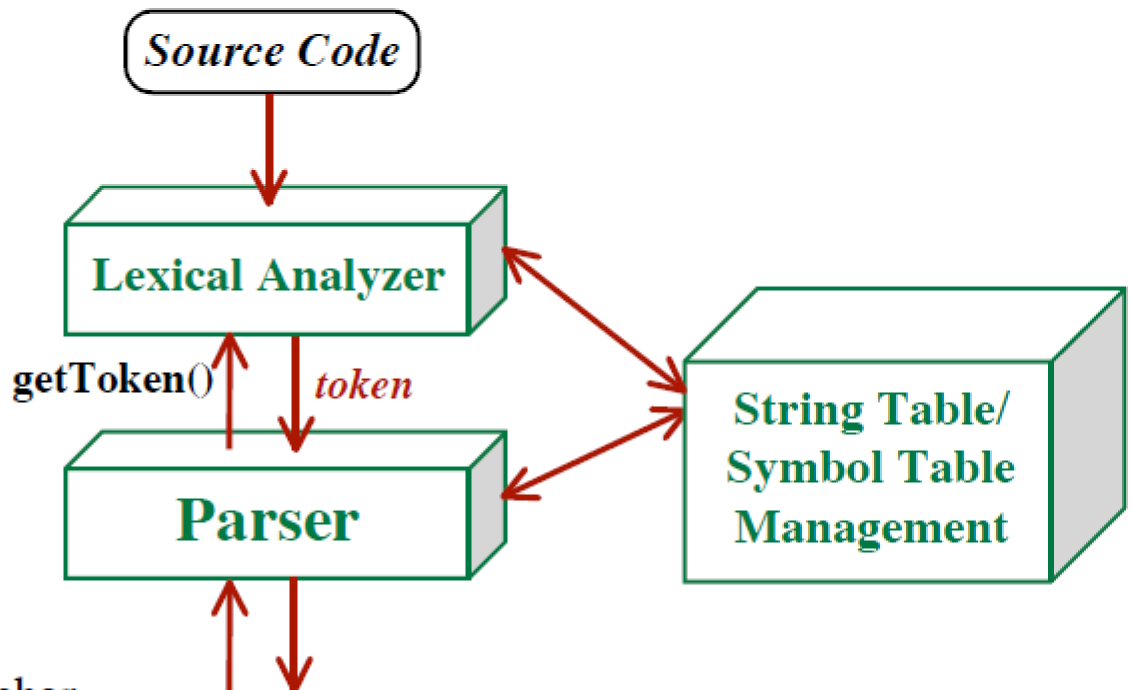
Chapter 3

Slide Credits

- Adapted from slides created by

  Harry H. Porter III, Ph.D., Portland State University
  - http://web.cecs.pdx.edu/~harry/compilers/syllabus.html

Lexical Analysis



also called
Lexer or scanner

- Must be efficient
- Looks at every input char
- Textbook, Chapter 3

Tokens

Token Type

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

Lexeme

The characters actually matched.

Example:

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

Tokens

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"

Tokens

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"

Tokens will appear as TERMINALS in the grammar.

Stmt → `while` Expr `do` StmtList `endWhile`

→ ID "=" Expr ";"

→ ...

Lexical Errors

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 Errors

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 Errors

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...

Lexer must keep going!

Always return a valid token.

Skip characters, if necessary. Other possibilities: insert, replace, transpose

May confuse the parser

The parser will detect syntax errors and get straightened out (hopefully!)

Managing Input Buffers

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 Buffers

Option 1: Read one char from OS at a time.

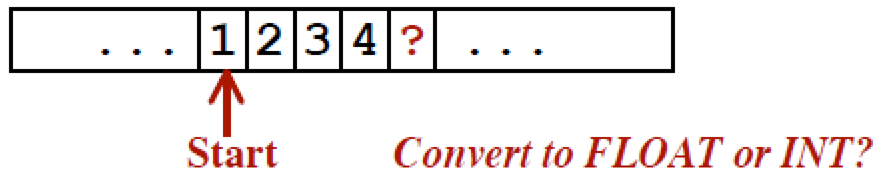
Option 2: Read N characters per system call

e.g., N = 4096

Manage input buffers in Lexer

More efficient

Often, we need to look ahead



Managing Input Buffers

Option 1: Read one char from OS at a time.

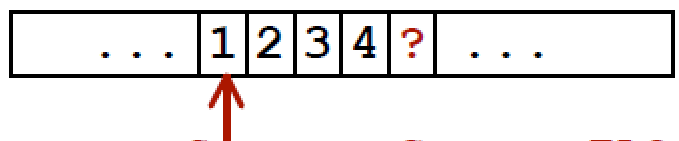
Option 2: Read N characters per system call

e.g., N = 4096

Manage input buffers in Lexer

More efficient

Often, we need to look ahead



Start *Convert to FLOAT or INT?*

Token could overlap / span buffer boundaries.

⇒ need 2 buffers

Code:

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

Managing Input Buffers

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

...	1	2	3	4	?	...

Start *Convert to FLOAT or INT?*

Token could overlap / span buffer boundaries.

⇒ need 2 buffers

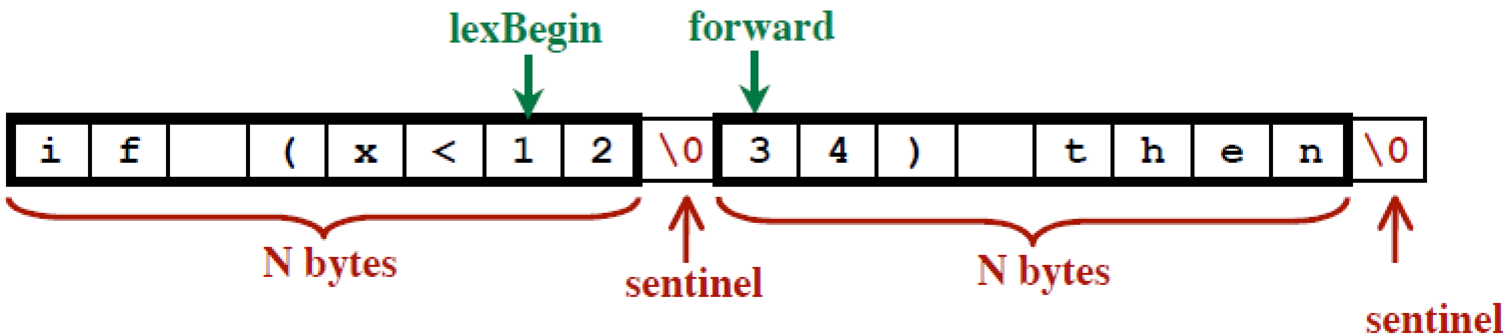
Code:

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

Technique: Use "Sentinels" to reduce testing

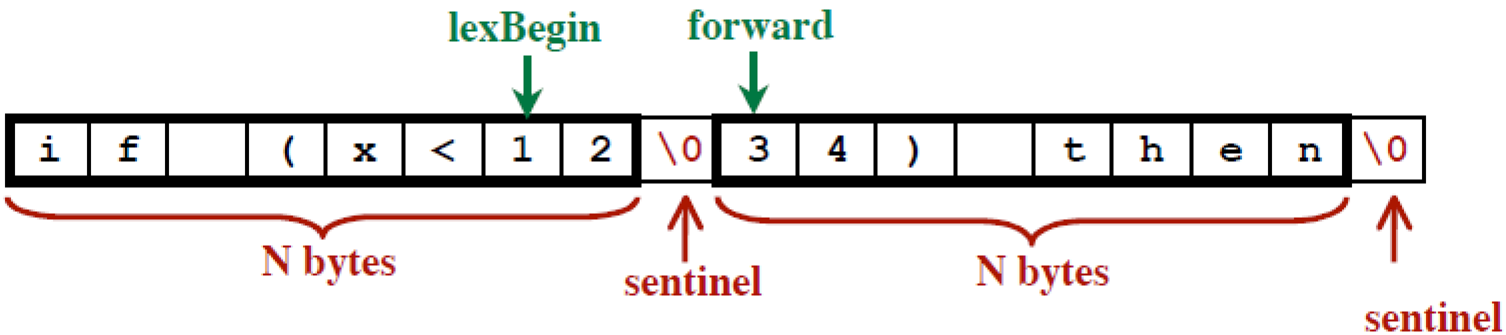
Choose some character that occurs rarely in most inputs

`'\0'`



Goal: Advance **`forward`** pointer to next character

...and reload buffer if necessary.



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

One fast test
...which usually fails

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
```

"Alphabet" (Σ)

A set of symbols ("characters")

Examples: $\Sigma = \{$ `a, b, c, d` $\}$

Σ = ASCII character set

"Alphabet" (Σ)

A set of symbols ("characters")

Examples: $\Sigma = \{$ `a, b, c, d` $\}$

Σ = ASCII character set

"String" (or "Sentence")

Sequence of symbols

Finite in length

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

"Alphabet" (Σ)

A set of symbols ("characters")

Examples: $\Sigma = \{$ `a, b, c, d` $\}$

Σ = ASCII character set

"String" (or "Sentence")

Sequence of symbols

Finite in length

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

"Empty String" (ε, "epsilon")

It is a string

$|\varepsilon| = 0$

"Alphabet" (Σ)

A set of symbols ("characters")

Examples: $\Sigma = \{$ `a, b, c, d` $\}$

Σ = ASCII character set

"String" (or "Sentence")

Sequence of symbols

Finite in length

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

"Empty String" (ε, "epsilon")

It is a string

$|\varepsilon| = 0$

"Language"

A set of strings

Examples:

$L_1 = \{$ `a, baa, bccb` $\}$

$L_2 = \{ \}$

$L_3 = \{ \varepsilon \}$

$L_4 = \{$ 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.

"Prefix" ...of string s

s = **`hello`**

Prefixes: **`he`**
`hello`
ε

"Prefix" ...of string s

s = `hello`

Prefixes: `he`
`hello`
ε

"Suffix" ...of string s

s = `hello`

Suffixes: `llo`
ε
`hello`

"Prefix" ...of string s

s = `hello`

Prefixes: `he`
`hello`
ε

"Suffix" ...of string s

s = `hello`

Suffixes: `llo`
ε
`hello`

"Substring" ...of string s

Remove a prefix and a suffix

s = `hello`

Substrings: `ell`
`hello`
ε

"Prefix" ...of string s

s = `hello`

Prefixes: `he`
`hello`
ε

"Suffix" ...of string s

s = `hello`

Suffixes: `llo`
ε
`hello`

"Substring" ...of string s

Remove a prefix and a suffix

s = `hello`

Substrings: `ell`
`hello`
ε

"Proper" prefix / suffix / substring ... of s

≠ s and ≠ ε

"Prefix" ...of string s

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

"Suffix" ...of string s

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

"Substring" ...of string s
Remove a prefix and a suffix

s = `hello` *Substrings:* `ell`
`hello`
ε

"Proper" prefix / suffix / substring ... of s
≠ s and ≠ ε

"Subsequence" ...of string s,

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

"Concatenation"

Strings: x, y

Concatenation: xy

Example:

x = `abb`

y = `cdc`

xy = `abbcdc`

yx = `cdcabb`

Other notations: **x || y**

x + y

x ++ y

x · y

"Concatenation"

Strings: x, y

Concatenation: xy

Example:

x = abb

y = cdc

xy = abbcdc

yx = cdcabb

Other notations: **x || y**

x + y

x ++ y

x · y

What is the "identity" for concatenation?

$\varepsilon x = x\varepsilon = x$

Multiplication ⇔ Concatenation

Exponentiation ⇔ ?

"Concatenation"

Strings: x, y

Concatenation: xy

Example:

$x =$ abb

$y =$ cdc

$xy =$ abbcdc

$yx =$ cdcabb

Other notations: **x || y**

x + y

x ++ y

x · y

What is the "identity" for concatenation?

$\varepsilon x = x\varepsilon = x$

Multiplication ⇔ Concatenation

Exponentiation ⇔ ?

Define $s^{\mathbf{0}} = \varepsilon$

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

Example $x =$ ab

$x^{\mathbf{0}} = \varepsilon$

$x^{\mathbf{1}} = x =$ ab

$x^{\mathbf{2}} = xx =$ abab

$x^{\mathbf{3}} = xxx =$ ababab

...etc...

"Concatenation"

Strings: x, y

Concatenation: xy

Example:

$x = abb$

$y = cdc$

$xy = abbcdc$

$yx = cdcabb$

Other notations: $x \| y$

$x + y$

$x ++ y$

$x \cdot y$

What is the "identity" for concatenation?

$\varepsilon x = x\varepsilon = x$

Multiplication $\Leftrightarrow$ Concatenation

Exponentiation $\Leftrightarrow$?

Define $s^0 = \varepsilon$

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

Example $x = ab$

$x^0 = \varepsilon$

$x^1 = x = ab$

$x^2 = xx = abab$

$x^3 = xxx = ababab$

...etc...

$x^* = x^\infty = abababababab...$

Infinite sequence of symbols!

Technically, this is not a "string"

"Language"

A set of strings

$L = \{ \dots \}$

$M = \{ \dots \}$

Generally, these are infinite sets.

"Language"

A set of strings

$L = \{ \ldots \}$

$M = \{ \ldots \}$

Generally, these are infinite sets.

"Union" of two languages

$L \cup M = \{ s \mid s \text{ is in } L \text{ or is in } M \}$

Example:

$L = \{$ `a`, `ab` $\}$

$M = \{$ `c`, `dd` $\}$

$L \cup M = \{$ `a`, `ab`, `c`, `dd` $\}$

"Language"

A set of strings

$L = \{ \dots \}$

$M = \{ \dots \}$

Generally, these are infinite sets.

"Union" of two languages

$L \cup M = \{ s \mid s \text{ is in } L \text{ or is in } M \}$

Example:

$L = \{$ `a`, `ab` $\}$

$M = \{$ `c`, `dd` $\}$

$L \cup M = \{$ `a`, `ab`, `c`, `dd` $\}$

"Concatenation" of two languages

$L\,M = \{ st \mid s \in L \text{ and } t \in M \}$

Example:

$L = \{$ `a`, `ab` $\}$

$M = \{$ `c`, `dd` $\}$

$L\,M = \{$ `ac`, `add`, `abc`, `abdd` $\}$

Repeated Concatenation

Let: $L = \{ \texttt{a}, \texttt{bc} \}$

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

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

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

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

...etc...

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

Kleene Closure

Let: $L = \{ \texttt{a}, \texttt{bc} \}$

Example:

$$L^0 = \{ \varepsilon \}$$
$$L^1 = L = \{ \texttt{a}, \texttt{bc} \}$$
$$L^2 = LL = \{ \texttt{aa}, \texttt{abc}, \texttt{bca}, \texttt{bcbc} \}$$
$$L^3 = LLL = \{ \texttt{aaa}, \texttt{aabc}, \texttt{abca}, \texttt{abcbc}, \texttt{bcaa}, \texttt{bcabc}, \texttt{bcbca}, \texttt{bcbcbc} \}$$

...etc...

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

The "Kleene Closure" of a language:

$$\sum_{i=0}^{\infty} a^i = a^0 \cup a^1 \cup a^2 \cup \ldots$$

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$

Example:

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

Positive Closure

Let: $L = \{ \texttt{a}, \texttt{bc} \}$

Example:
$L^0 = \{ \varepsilon \}$
$L^1 = L = \{ \texttt{a}, \texttt{bc} \}$
$L^2 = LL = \{ \texttt{aa}, \texttt{abc}, \texttt{bca}, \texttt{bcbc} \}$
$L^3 = LLL = \{ \texttt{aaa}, \texttt{aabc}, \texttt{abca}, \texttt{abcbc}, \texttt{bcaa}, \texttt{bcabc}, \texttt{bcbca}, \texttt{bcbcbc} \}$
...etc...
$L^N = L^{N-1}L = LL^{N-1}$

$$\sum_{i=0}^{\infty} a^i = a^0 \cup a^1 \cup a^2 \cup \ldots$$

The "Positive Closure" of a language:

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

Example:

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

Positive Closure

Let: $L = \{ \texttt{a}, \texttt{bc} \}$

Example:
$L^0 = \{ \varepsilon \}$
$L^1 = L = \{ \texttt{a}, \texttt{bc} \}$
$L^2 = LL = \{ \texttt{aa}, \texttt{abc}, \texttt{bca}, \texttt{bcbc} \}$
$L^3 = LLL = \{ \texttt{aaa}, \texttt{aabc}, \texttt{abca}, \texttt{abcbc}, \texttt{bcaa}, \texttt{bcabc}, \texttt{bcbca}, \texttt{bcbcbc} \}$
...etc...
$L^N = L^{N-1}L = LL^{N-1}$

$$\sum_{i=0}^{\infty} a^i = a^0 \cup a^1 \cup a^2 \cup \ldots$$

The "Positive Closure" of a language:

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

Example:

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

Note that ε is not included UNLESS it is in L to start with

Examples

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

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

$D^{+} =$

Examples

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

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

$L \cup D =$

Examples

Let: $L = \{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z} \}$
$D = \{ \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

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

$L \cup D =$
"The set of alphanumeric characters"
$\{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

$(L \cup D)^{*} =$

Examples

Let: $L = \{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z} \}$
$D = \{ \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

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

$L \cup D =$
"The set of alphanumeric characters"
$\{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

$(L \cup D)^* =$
"Sequences of zero or more letters and digits"

$L \ (L \cup D)^* =$

Examples

Let: $L = \{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z} \}$
$D = \{ \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

$D^+ =$

"The set of strings with one or more digits"

$L \cup D =$

"The set of alphanumeric characters"

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

$(L \cup D)^* =$

"Sequences of zero or more letters and digits"

$L \, ((L \cup D)^*) =$

Examples

Let: $L = \{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z} \}$
$D = \{ \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

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

$L \cup D =$
"The set of alphanumeric characters"
$\{ \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9} \}$

$(L \cup D)^{*} =$
"Sequences of zero or more letters and digits"

$L((L \cup D)^{*}) =$
"Set of strings that start with a letter, followed by zero or more letters and and digits."

Regular Expressions

Assume the alphabet is given... *e.g.,* $\Sigma = \{$ `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`,
`. . .`}

Meta Symbols:

`( )`

`|`

`*`

ε

How to "Parse" Regular Expressions

* has highest precedence.

Concatenation as middle precedence.

| has lowest precedence.

Use parentheses to override these rules.

How to "Parse" Regular Expressions

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

Examples:

$a\ b^* =$

How to "Parse" Regular Expressions

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

Examples:

`a b*  =  a (b*)`

If you want `(a b)*` you must use parentheses.

How to "Parse" Regular Expressions

* has highest precedence.

Concatenation as middle precedence.

| has lowest precedence.

Use parentheses to override these rules.

Examples:

`a b*  = a (b*)`

If you want `(a b)*` you must use parentheses.

`a | b c =`

How to "Parse" Regular Expressions

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

Examples:

`a b*  =  a (b*)`

If you want `(a b)*` you must use parentheses.

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

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

How to "Parse" Regular Expressions

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

Examples:

`a b* = a (b*)`

If you want `(a b)*` 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`

How to "Parse" Regular Expressions

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

Examples:

`a b* = a (b*)`

If you want `(a b)*` 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:

`b d | e f * | g a =`

How to "Parse" Regular Expressions

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

Examples:

`a b*  = a (b*)`

If you want `(a b)*` 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:

`b d | e f* | g a  =  b d | e (f*) | g a`

How to "Parse" Regular Expressions

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

Examples:

`a b*  = a (b*)`

If you want `(a b)*` 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:

`b d | e f * | g a  =   (b d) | (e (f *))  | (g a)`

How to "Parse" Regular Expressions

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

Examples:

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

If you want $(a\ b)^*$ you must use parentheses.

$a \mid b\ c = a \mid (b\ c)$

If you want $(a \mid b)\ c$ you must use parentheses.

Concatenation and | are associative.

$(a\ b)\ c = a\ (b\ c) = a\ b\ c$

$(a \mid b) \mid c = a \mid (b \mid c) = a \mid b \mid c$

Example:

$b\ d \mid e\ f^* \mid g\ a = ((b\ d) \mid (e\ (f^*))) \mid (g\ a)$

Fully parenthesized

Definition: Regular Expressions

(Over alphabet Σ)

- ε is a regular expression.
- If `a` is a symbol (i.e., if $a \in \Sigma$), then `a` is a regular expression.
- If `R` and `S` are regular expressions, then `R|S` is a regular expression.
- If `R` and `S` are regular expressions, then `RS` is a regular expression.
- If `R` is a regular expression, then `R*` is a regular expression.
- If `R` is a regular expression, then `(R)` is a regular expression.

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

- If `a` is a symbol (i.e., if `a`$\in\Sigma$), then `a` is a regular expression.

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

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

- If `R` is a regular expression, then `R*` is a regular expression.

- If `R` is a regular expression, then `(R)` is a regular expression.

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

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

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

- If R is a regular expression, then R^* is a regular expression.

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

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

  $L(a) = \{ a \}$

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

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

- If R is a regular expression, then R^* is a regular expression.

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

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

  $L(a) = \{ a \}$

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

  $L(R|S) = L(R) \cup L(S)$

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

- If R is a regular expression, then R^* is a regular expression.

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

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

  $L(a) = \{ a \}$

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

  $L(R|S) = L(R) \cup L(S)$

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

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

- If R is a regular expression, then R^* is a regular expression.

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

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

  $L(a) = \{ a \}$

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

  $L(R|S) = L(R) \cup L(S)$

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

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

- If R is a regular expression, then R^* is a regular expression.

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

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

Definition: Regular Expressions

(Over alphabet Σ)

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

- ε is a regular expression.

  $L(\varepsilon) = \{ \varepsilon \}$

- If a is a symbol (i.e., if $a \in \Sigma$), then a is a regular expression.

  $L(a) = \{ a \}$

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

  $L(R|S) = L(R) \cup L(S)$

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

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

- If R is a regular expression, then R^* is a regular expression.

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

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

  $L(\,(R)\,) = L(R)$

Regular Languages

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 Langauges $\subset$ 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."

Equality v. Equivalence

Are these regular expressions equal?

```
R = a a* (b | c)
S = a* a (c | b)
```

... No!

Yet, they describe the same language.

L(R) = L(S)

"Equivalence" of regular expressions

If L(R) = L(S) then we say R ≈ S

"R is equivalent to S"

"Syntactic" equality versus "deeper" equality...

Algebra:

Does... $x(3+b) = 3x+bx$?

From now on, we'll just say R = S to mean R ≈ S

Notation:

Equality	Equivalence
=	=
	==
	≈
	≡
	⇔

Algebraic Laws of Regular Expressions

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

| is commutative

$$R \mid S = S \mid R$$

| is associative

$$R \mid (S \mid T) = (R \mid S) \mid T = \overbrace{R \mid S \mid T}^{\textit{Preferred}}$$

Concatenation is associative

$$R\,(S\,T) = (R\,S)\,T = \underbrace{R\,S\,T}_{\textit{Preferred}}$$

Concatenation distributes over |

$$R\,(S \mid T) = RS \mid RT$$
$$(R \mid S)\,T = RT \mid ST$$

ε is the identity for concatenation

$$\varepsilon R = R \varepsilon = R$$

* is idempotent

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

Relation between * and ε

$$R^* = (R \mid \varepsilon)^*$$

Regular Definitions

Letter	=	`a	b	c	...	z`
Digit	=	`0	1	2	...	9`
ID	=	Letter (Letter | Digit)*

Regular Definitions

```
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 Definitions

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`", ... }

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

Addition Notation / Shorthand

One-or-more: $+$

$X^+ = X\,(X^*)$

Digit$^+$ = Digit Digit* = Digits

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

$X? = (X \mid \varepsilon)$

Num = Digit$^+$ (. Digit$^+$) ?

Addition Notation / Shorthand

One-or-more: $^{+}$

$X^{+} = X\,(X^{*})$

Digit^{+} = Digit Digit* = Digits

Optional (zero-or-one): ?

$X? = (X \mid \varepsilon)$

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]`

Many sets of strings are not regular.

...no regular expression for them!

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!

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

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

$\underbrace{\texttt{abbbab}}\texttt{c}\underbrace{\texttt{abbbab}}$

CFG is not even powerful enough.

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

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

`abbbabcabbbab`

CFG is not even powerful enough.

The Problem?

In order to recognize a string,

these languages require *memory*!

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

Input: Sequence of regular definitions

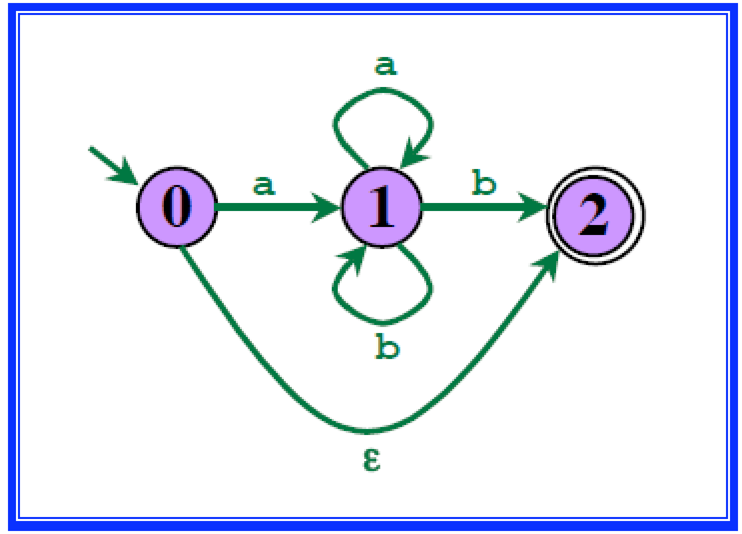
Output: A lexer (e.g., a program in Java or "C")

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 (FSAs)

("Finite State Machines","Finite Automata", "FA")



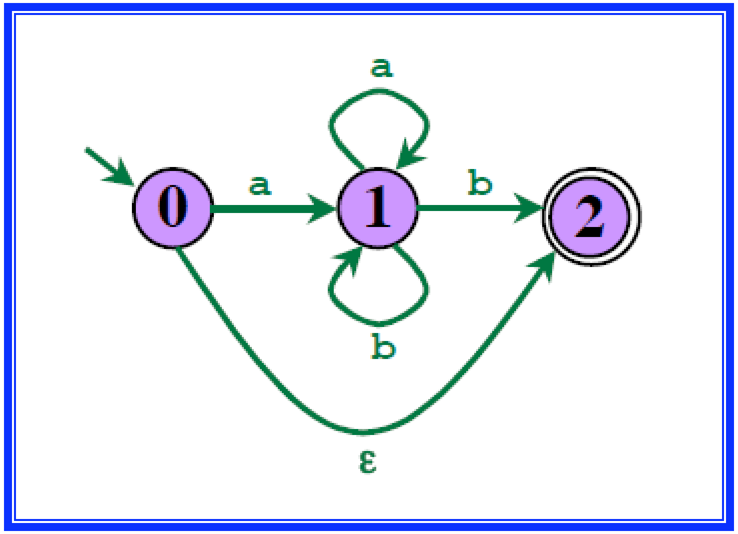
- One start state
- Many final states
- Each state is labeled with a state name
- Directed edges, labeled with symbols
  - Deterministic (DFA)

    No **ε**-edges

    Each outgoing edge has different symbol
  - Non-deterministic (NFA)

Finite State Automata (FSAs)

Formalism: $\langle S, \Sigma, \delta, S_0, S_F \rangle$



S = Set of states

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

Σ = Input Alphabet

Σ = ASCII Characters

δ = Transition Function

$S \times \Sigma \rightarrow$ States (deterministic)

$S \times \Sigma \rightarrow$ 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 (FSAs)

Formalism: $< S, \Sigma, \delta, S_0, S_F >$

S = Set of states

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

Σ = Input Alphabet

Σ = ASCII Characters

δ = Transition Function

$S \times \Sigma \rightarrow$ States (deterministic)

$S \times \Sigma \rightarrow$ Sets of States (non-deterministic)

s_0 = Start State

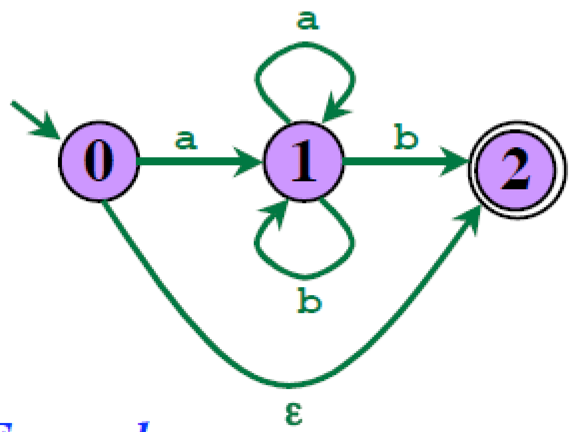
"Initial state"

$s_0 \in S$

S_F = Set of final states

"accepting states"

$S_F \subseteq S$



Example:

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

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

$s_0 = 0$

$S_F = \{ 2 \}$

δ =

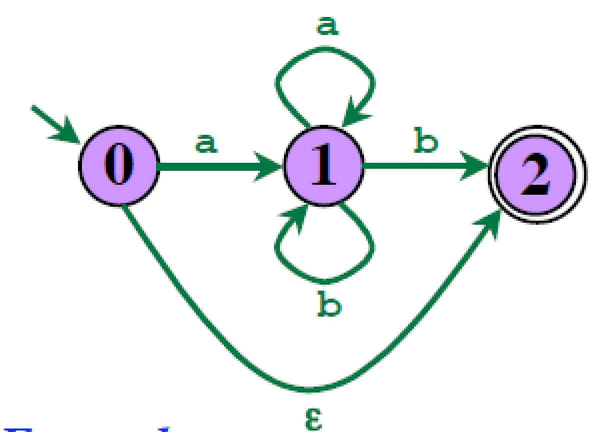
States \ Input Symbols	a	b	ε
0	{1}	{}	{2}
1	{1}	{1,2}	{}
2	{}	{}	{}

Finite State Automata (FSAs)

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:

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

Input Symbols

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

Deterministic Finite Automata (DFAs)

No ε-moves

The transition function returns a single state

$\delta: S \times \Sigma \rightarrow S$

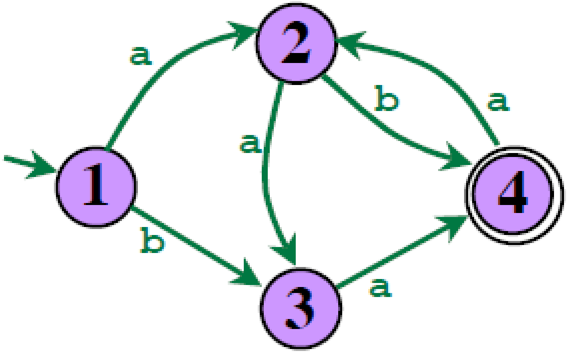
```
function Move (s:State, a:Symbol) returns State
```

$\delta =$

Input Symbols

States

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



Deterministic Finite Automata (DFAs)

No ε-moves

The transition function returns a single state

$\delta: S \times \Sigma \rightarrow S$

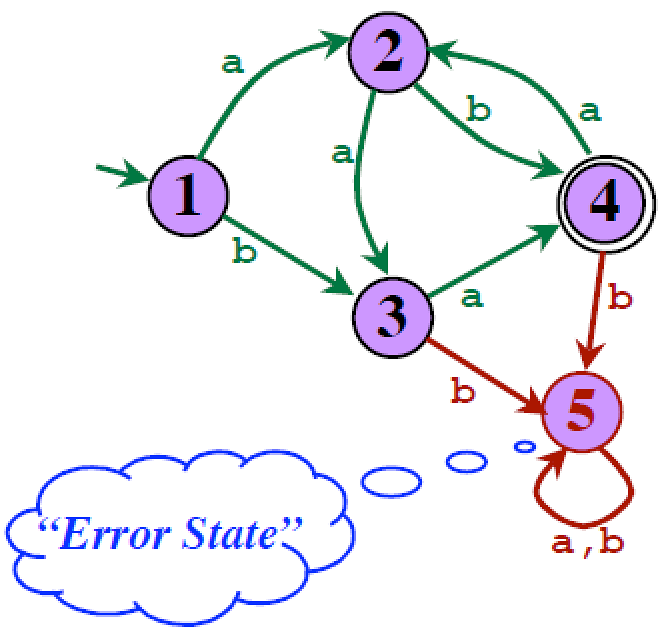
```
function Move (s:State, a:Symbol) returns State
```

$\delta =$

Input Symbols

States

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



Deterministic Finite Automata (DFAs)

No **ε**-moves

The transition function returns a single state

$\delta: S \times \Sigma \rightarrow S$

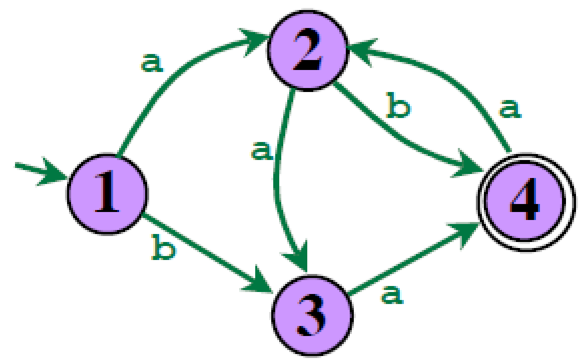
```
function Move (s:State, a:Symbol) returns State
```

$\delta =$

Input Symbols

States

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



Non-Deterministic Finite Automata (NFAs)

Allow **ε**-moves

The transition function returns a *set of* states

$\delta: S \times \Sigma \rightarrow \text{Powerset}(S)$

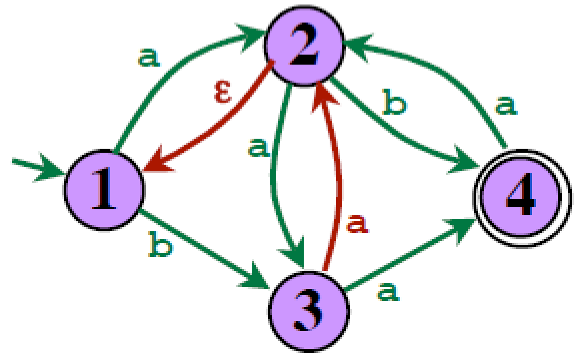
$\delta: S \times \Sigma \rightarrow P\,(S)$

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

$\delta =$

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.

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.

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.

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.

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")

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...)