Lexical Analysis

- Must be efficient
- Looks at every input char
- Textbook, Chapter 2

Tokens

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

Lexeme
The characters actually matched.
Example:

```plaintext
... 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 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

```c
return 1.23;
return 1,23;
```

... Still results in sequence of legal tokens

```c
<ID,"return"> <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;
return 1,23;
```

... Still results in sequence of legal tokens

```
<ID, "return"> <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.
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

```
... 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 ...
```
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

![Token could overlap / span buffer boundaries. ⇒ need 2 buffers](image)

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.
Lexical Analysis - Part 1

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

Code:

```c
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
```

One fast test...which usually fails

Alphabet (Σ)

A set of symbols (“characters”)

Examples: Σ = { a, b, c, d }

Σ = ASCII character set
“Alphabet” ($\Sigma$)
A set of symbols (“characters”)

*Examples:* $\Sigma = \{ \text{a, b, c, d} \}$
$\Sigma =$ ASCII character set

“String” (or “Sentence”)
Sequence of symbols
Finite in length

*Example:* $\text{abbadc}$ Length of $s = |s|$
“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|

“Empty String” (ε, “epsilon”)
It is a string
|ε| = 0

“Language”
A set of strings

Examples:

\[ L_1 = \{ a, baa, bccb \} \]
\[ L_2 = \{ \} \]
\[ L_3 = \{ ε \} \]
\[ L_4 = \{ ε, ab, abab, ababab, abababab, \ldots \} \]
\[ L_5 = \{ s \mid s \text{ can be interpreted as an English sentence making a true statement about mathematics} \} \]

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
Lexical Analysis - Part 1

“Prefix” ...of string s
s = hello
Prefixes:
he
hello
ε

“Suffix” ...of string s
s = hello
Suffixes:
ll0
ε
hello

“Substring” ...of string s
Remove a prefix and a suffix
s = hello
Substrings:
el1
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

“Subsequence” ...of string s,
S = compilers
Subsequences: opilr
cors
compilers
ε
“Concatenation”

Strings: x, y
Concatenation: xy
Example:

\[
\begin{align*}
x &= \text{abb} \\
y &= \text{cdc} \\
xy &= \text{abbcdc} \\
yx &= \text{cdcabb}
\end{align*}
\]

Other notations:

\[
\begin{align*}
x \parallel y \\
x + y \\
x ++ y \\
x \cdot y
\end{align*}
\]

What is the “identity” for concatenation?

\[
\begin{align*}
\varepsilon x &= x \\
\varepsilon x &= x
\end{align*}
\]

Multiplication $\Leftrightarrow$ Concatenation

Exponentiation $\Leftrightarrow$ ?
“Concatenation”
Strings: x, y
Concatenation: xy
Example:
  x = abb
  y = cdc
  xy = abbcdc
  yx = cdcabb
What is the “identity” for concatenation?
  ε x = xε = x
Multiplication ⇔ Concatenation
Exponentiation ⇔ ?

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...

Other notations:
  x \parallel y
  x + y
  x ++ y
  x \cdot y

Infinite sequence of symbols!
Technically, this is not a “string”
“Language”
A set of strings
L = { ... }
M = { ... }

Generally, these are infinite sets.

“Union” of two languages
L ∪ M = { s | s is in L or is in M }

Example:
L = { a, ab }
M = { c, dd }
L ∪ M = { a, ab, c, dd }

Generally, these are infinite sets.
“Language”
A set of strings
L = { ... }
M = { ... }

“Union” of two languages
L ∪ M = { s | s is in L or is in M }

Example:
L = { a, ab }
M = { c, dd }
L ∪ M = { a, ab, c, dd }

“Concatenation” of two languages
L M = { st | s ∈ L and t ∈ M }

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

Generally, these are infinite sets.

Repeated Concatenation

Let:
L = { a, bc }

Example:
L⁰ = { ε }
L¹ = L = { a, bc }
L² = LL = { aa, abc, bca, bcbbc }
L³ = LLL = { aaa, aabc, abca, abcbbc, bcaa, bcabc, bcbca, bcbbcbc }
...etc...
Lᴺ = Lᴺ⁻¹L = LLᴺ⁻¹
**Kleene Closure**

Let: \( L = \{ a, bc \} \)

Example: \( L^0 = \{ \varepsilon \} \)

\[
\begin{align*}
L^1 &= L = \{ a, bc \} \\
L^2 &= LL = \{ aa, abc, bca, bcbc \} \\
L^3 &= LLL = \{ aaa, aabc, abca, abc, bca, bcb, bcba, bcbbc \} \\
& \quad \text{...etc...} \\
L^N &= L^{N-1}L = LL^{N-1}
\end{align*}
\]

The “Kleene Closure” of a language:

\[
L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup ...
\]

Example:

\[
L^* = \{ \varepsilon, a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcbc, ... \}
\]

---

**Positive Closure**

Let: \( L = \{ a, bc \} \)

Example: \( L^0 = \{ \varepsilon \} \)

\[
\begin{align*}
L^1 &= L = \{ a, bc \} \\
L^2 &= LL = \{ aa, abc, bca, bcbc \} \\
L^3 &= LLL = \{ aaa, aabc, abca, abc, bca, bcb, bcba, bcbbc \} \\
& \quad \text{...etc...} \\
L^N &= L^{N-1}L = LL^{N-1}
\end{align*}
\]

The “Positive Closure” of a language:

\[
L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup ...
\]

Example:

\[
L^+ = \{ a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcbc, ... \}
\]
Positive Closure

Let: \[ L = \{ a, bc \} \]

Example: \[ L^0 = \{ \epsilon \} \]
- \[ L^1 = L = \{ a, bc \} \]
- \[ L^2 = LL = \{ aa, abc, bca, bcbc \} \]
- \[ L^3 = LLL = \{ aaa, aabc, abca, abcba, bcabc, bcbca, bcbcbc \} \]
...etc...
- \[ L^N = L^{N-1}L = LL^{N-1} \]

The “Positive Closure” of a language:
\[ L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup ... \]

Example:
\[ L^+ = \{ a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcba, bcabc, bcbca, bcbcbc, ... \} \]

Note that \( \epsilon \) is not included UNLESS it is in \( L \) to start with

Examples

Let: \[ L = \{ a, b, c, ..., z \} \]
- \[ D = \{ 0, 1, 2, ..., 9 \} \]
- \[ D^+ = \]
Examples

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

\( D^+ = \)

"The set of strings with one or more digits"

\( L \cup D = \)

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

---

Examples

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

\( D^+ = \)

"The set of strings with one or more digits"

\( L \cup D = \)

"The set of alphanumeric characters"
\( \{ a, b, c, \ldots, z, 0, 1, 2, \ldots, 9 \} \)

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

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

\[ D^+ = \]
“The set of strings with one or more digits”

\[ L \cup D = \]
“The set of alphanumeric characters”
\[ \{ a, b, c, ..., z, 0, 1, 2, ..., 9 \} \]

\[ (L \cup D)^* = \]
“Sequences of zero or more letters and digits”

\[ L \ (L \cup D)^* = \]
Examples

Let:

\[ L = \{ a, b, c, \ldots, z \} \]

\[ D = \{ 0, 1, 2, \ldots, 9 \} \]

\[ D^+ = \]

“\( \text{The set of strings with one or more digits} \)”

\[ L \cup D = \]

“\( \text{The set of alphanumeric characters} \)”

\[ \{ a, b, c, \ldots, z, 0, 1, 2, \ldots, 9 \} \]

\[ (L \cup D)^* = \]

“\( \text{Sequences of zero or more letters and digits} \)”

\[ L((L \cup D)^*) = \]

“\( \text{Set of strings that start with a letter, followed by zero or more letters and digits.} \)”

Regular Expressions

Assume the alphabet is given... e.g., \( \Sigma = \{ a, b, c, \ldots, z \} \)

Example:

\[ a (b \mid c) d^* e \]

A regular expression describes a language.

Notation:

- \( r \) = regular expression
- \( L(r) \) = the corresponding language

Example:

\[ r = a (b \mid c) d^* e \]

\[ L(r) = \{ \text{abe, abde, abdde, abddde, \ldots, ace, acde, acdde, acddde, \ldots} \} \]
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.
\[ a \mid b\,c = a \mid (b\,c) \]
If you want \((a \mid b)\,c\) you must use parentheses.

Concatenation and | are associative.
\[ (a\,b)\,c = a\,(b\,c) = a\,b\,c \]
\[ (a \mid b) \mid c = a \mid (b \mid c) = a \mid b \mid c \]
**How to “Parse” Regular Expressions**

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

**Examples:**

\[
\begin{align*}
    a \ b^* &= a \ (b^*) \\
    \text{If you want} \ (a \ b)^* \text{ you must use parentheses.}
    \\
    a \ | \ b \ c &= a \ | \ (b \ c) \\
    \text{If you want} \ (a \ | \ b) \ c \text{ you must use parentheses.}
\end{align*}
\]

Concatenation and | are associative.

\[
\begin{align*}
    (a \ b) \ c &= a \ (b \ c) = a \ b \ c \\
    (a \ | \ b) \ | \ c &= a \ | \ (b \ | \ c) = a \ | \ b \ | \ c
\end{align*}
\]

**Example:**

\[
\begin{align*}
    b \ d \ | \ e \ f^* \ | \ g \ a &= b \ d \ | \ e \ (f^*) \ | \ g \ a
\end{align*}
\]
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:**
  \[
  b \, d \mid e \, f^* \mid g \, a = (b \, d) \mid (e \, (f^*)) \mid (g \, a)
  \]
Definition: Regular Expressions
(Over alphabet \( \Sigma \))

- \( \varepsilon \) 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.

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

- \( \varepsilon \) 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 $\Sigma$)

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

• $\varepsilon$ 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 \( \Sigma \))

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

- \( \varepsilon \) 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 \( R \mid S \) is a regular expression.
  \[ L(R \mid S) = L(R) \cup L(S) \]

- 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 $\Sigma$)

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

• $\varepsilon$ 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 Languages \( \subset \) Context-Free Languages

**Examples:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a</td>
<td>b</td>
</tr>
<tr>
<td>( b^* )</td>
<td>{ε, b, bb, bbb, \ldots}</td>
</tr>
<tr>
<td>( a</td>
<td>b^* )</td>
</tr>
<tr>
<td>((a</td>
<td>b)^*)</td>
</tr>
</tbody>
</table>

“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 \( R \sim S \)

“R is equivalent to S”

“Syntactic” equality versus “deeper” equality...

Algebra:

\[ x(3+b) = 3x+bx \ ? \]

From now on, we’ll just say \( R = S \) to mean \( R \sim S \)
Algebraic Laws of Regular Expressions

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

**1 is commutative**

$$R I S = S I R$$

**1 is associative**

$$R I (S I T) = (R I S) I T = R I S I T$$

**Concatenation is associative**

$$R (S T) = (R S) T = R S T$$

**Concatenation distributes over $I$**

$$R (S T) = R S I R T$$

$$R S T = R S T$$

$\epsilon$ is the identity for concatenation

$$\epsilon R = R \epsilon = R$$

* is idempotent

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

**Relation between $*$ and $\epsilon$**

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

Regular Definitions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Letter</strong></td>
<td>$\equiv a \mid b \mid c \mid \ldots \mid z$</td>
</tr>
<tr>
<td><strong>Digit</strong></td>
<td>$\equiv 0 \mid 1 \mid 2 \mid \ldots \mid 9$</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>$\equiv \text{Letter} (\mid \text{Letter} \mid \text{Digit} )^*$</td>
</tr>
</tbody>
</table>
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: $+$
\[ X^+ = X (X^*) \]
\[ \text{Digit}^+ = \text{Digit} \quad \text{Digit}^* = \text{Digits} \]

Optional (zero-or-one): $?$
\[ X? = (X \mid \epsilon) \]
\[ \text{Num} = \text{Digit}^+ (\ .\ \text{Digit}^+ )? \]
Addition Notation / Shorthand

One-or-more:  
\[ X^+ = X (X^+) \]
\[ \text{Digit}^+ = \text{Digit} \quad \text{Digit}^* = \text{Digits} \]

Optional (zero-or-one):  
\[ X? = (X \mid \epsilon) \]
\[ \text{Num} = \text{Digit}^+ ( \cdot \text{Digit}^+ )? \]

Character Classes:  
\([\text{FirstChar} \text{−} \text{LastChar}]\)
Assumption: The underlying alphabet is known ...and is ordered.
\[ \text{Digit} = [0-9] \]
\[ \text{Letter} = [a-zA-Z] = [A-Za-z] \]

Variations:

Zero-or-more:  
\[ ab^*c = a(b)c = a(b)^*c \]

One-or-more:  
\[ ab^+c = a(b)^c \]

Optional:  
\[ ab?c = a[b)c \]

What does \( ab\ldots bc \) mean?
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  
\[
\{ XcX \mid X \text{ is a string of } a\text{'s and } b\text{'s } \}
\]
\[
abbbabcabbbab
\]
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 $\varepsilon$-edges
    Each outgoing edge has different symbol
  • Non-deterministic (NFA)
**Finite State Automata (FSAs)**

Formalism: \(< S, \Sigma, \delta, S_0, S_F >\)

- \(S = \text{Set of states}\)
  - \(S = \{s_0, s_1, ..., s_N\}\)

- \(\Sigma = \text{Input Alphabet}\)
  - \(\Sigma = \text{ASCII Characters}\)

- \(\delta = \text{Transition Function}\)
  - \(S \times \Sigma \rightarrow \text{States (deterministic)}\)
  - \(S \times \Sigma \rightarrow \text{Sets of States (non-deterministic)}\)

- \(s_0 = \text{Start State}\)
  - “Initial state”
  - \(s_0 \in S\)

- \(S_F = \text{Set of final states}\)
  - “accepting states”
  - \(S_F \subseteq S\)

---

**Example:**

- \(S = \{0, 1, 2\}\)
- \(\Sigma = \{a, b\}\)
- \(s_0 = 0\)
- \(S_F = \{2\}\)
- \(\delta = \begin{array}{c|cc}
0 & a & b \\
1 & (1) & (2) \\
2 & (1, 2) & (1) \\
\end{array}\)

**Input Symbols**

- \(a\)
- \(b\)
- \(\varepsilon\)

**States**

- 0
  - (1) (2)
- 1
  - (1, 2)
- 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:
- $\varepsilon$
- $aaab$
- $abbb$

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

Deterministic Finite Automata (DFAs)

No $\varepsilon$-moves
The transition function returns a single state
$\delta: S \times \Sigma \rightarrow S$

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

$\delta =$
Deterministic Finite Automata (DFAs)

No $\varepsilon$-moves
The transition function returns a single state
$\delta: S \times \Sigma \rightarrow S$

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

$\delta = \begin{array}{c|c|c|c|c|}
\text{Input Symbols} & a & b \\
\hline
\text{States} & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 2 & 3 & 4 & 5 & 5 \\
2 & 3 & 4 & 5 & 5 & 5 \\
3 & 4 & 5 & 5 & 5 & 5 \\
4 & 2 & 5 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 \\
\end{array}$

© Harry H. Porter, 2005
Non-Deterministic Finite Automata (NFAs)

Allow $\varepsilon$-moves

The transition function returns a *set of* states

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

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

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

$\delta =
\begin{array}{c|c|c|c}
\text{States} & a & b & \varepsilon \\
\hline
1 & (2) & (3) & () \\
2 & (3) & (4) & (1) \\
3 & (4,2) & () & () \\
4 & (2) & () & () \\
\end{array}$

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.

• 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.

• 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...)

What is the regular expression?
What is the regular expression?
\[ \varepsilon \mid a \ (a|b)^* \ b \]

What is an equivalent DFA?
What is the regular expression?
\[ \epsilon \mid a (a|b)^* b \]

What is an equivalent DFA?

What is the regular expression?
\[ \epsilon \mid a (a|b)^* b \]

What is an equivalent DFA?
What is the regular expression?
\[ \varepsilon \mid a (a|b)^* b \]

What is an equivalent DFA?