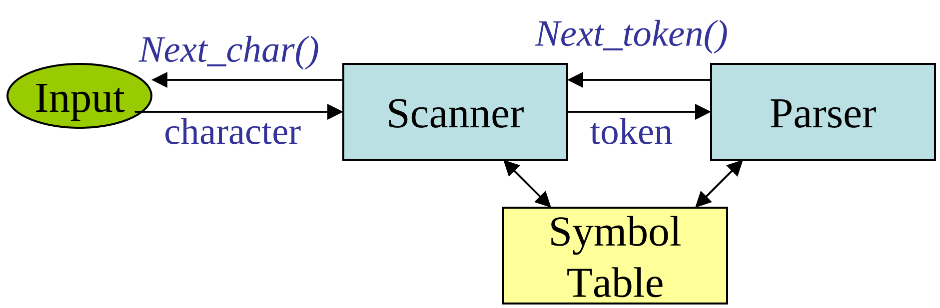
Lexical Analysis

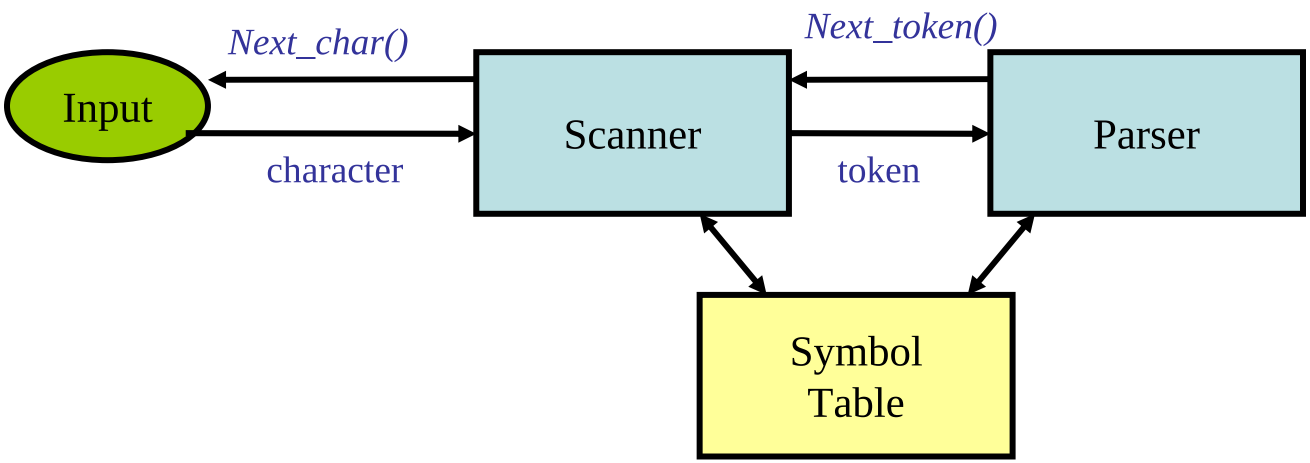
Lecture 02

Role of the Lexical Analyzer

- Identify the words: Lexical Analysis
  - Converts a stream of characters (input program) into a stream of tokens.
    - Also called Scanning or Tokenizing
- Identify the sentences: Parsing.
  - Derive the structure of sentences: construct parse trees from a stream of tokens.



Interaction of Lexical Analyzer with Parser



- Often a subroutine of the parser
- Secondary tasks of Lexical Analyzer
  - Strip out comments and white spaces from the source
  - Correlate error messages with the source program
  - Preprocessing may be implemented as lexical analysis takes place

Issues in lexical analysis

- **Simplicity/Modularity:** Conventions about “words" are often different from conventions about “sentences".
- **Efficiency:** Word identification problem has a much more efficient solution than sentence identification problem.
- **Portability:** Character set, special characters, device features.

Terminology

- **Token:** Name given to a family of words.
  - e.g., tok_integer_constant
- **Lexeme:** Actual sequence of characters representing a word.
  - e.g., 32894
- **Pattern:** Notation used to identify the set of lexemes represented by a token.
  - e.g., digit followed by zero or more digits

Token Sample	Lexemes	Pattern
tok_while	while	while
tok_integer_constant	32894, -1093, 0	digit followed by zero or more digits
tok_relation	<, <=, =, !=, >, >=	< or <= or = or != or >= or >
tok_identifier	buffer_size, D2	letter followed by letters or digits

Token Stream

- Tokens are terminal symbol in the grammar for the source language
- keywords, operators, identifiers, constants, literal strings, punctuation symbols etc. are treated as tokens

- Source:

  if (x == -3.1415) /* test x */ then ...

- Token Stream:

  < IF >

  < LPAREN >

  < ID, “x” >

  < EQUALS >

  < NUM, -3.14150000 >

  < RPAREN >

  < THEN >

  ...

Token Attributes

- More than one lexeme matches a pattern
  - We need attribute to distinguish them
    - e.g. “tok_relation” matches “< or <= or = or != or >= or >”
    - < tok_integer_constant, 1415 >

token type token attribute (if available)

  - Lexical analyzer collects information about tokens and as well as their attributes
- Attributes influence the translation of tokens
- A token usually has only a single attribute
  - A pointer to the symbol-table entry
  - Other attributes (e.g. line number, lexeme) can be stored in symbol table
- Example:
  - E = M * C ** 2

<tok_identifier, pointer to symbol table entry for E>
<tok_assign, >
<tok_identifier, pointer to symbol table entry for M>
......

Lexical Error

- Few errors can be caught by the lexical analyzer
  - 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!
  - Another example: fi (a == f(x))
- Errors caught by lexer:
  - EOF within a String / missing ”
  - Invalid ASCII character in file
  - String / ID exceeds maximum length
  - etc...

Recovery from lexical errors

- Panic mode recovery
  - Delete successive characters from the input until the lexical analyzer can find a well-formed token
  - "…….day = 30 ^^^ month; ……."
  - May confuse the parser
    - The parser will detect syntax errors and get straightened out (hopefully!)
- Other possible error-recovery actions
    - Deleting an extra character
    - Inserting a missing character
    - Replacing an incorrect character by a correct character
    - Swapping two adjacent character
  - Attempt to repair the input using single error transformations

Implementing a lexical analyzer

assembly language

system-programming language

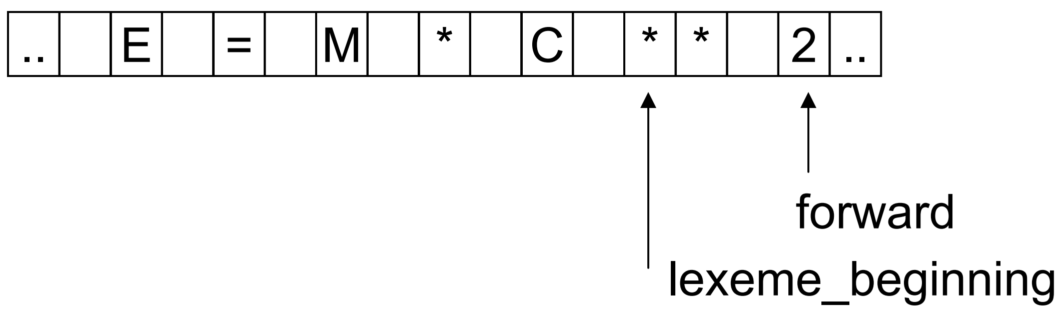
lexical-analyzer generator

Harder to implement

Faster

Managing Input Buffers

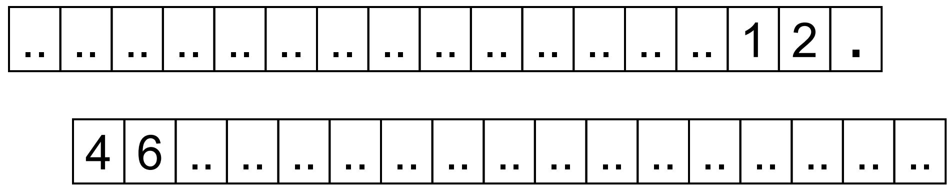
- **Option 1:** Read one char from OS at a time.
- **Option 2:** Read N characters per system call
  - e.g., N = 4096
- Manage input buffers in Lexer
  - More efficient
- Often, we need to look ahead



- But! Due to look ahead we need to push back the lookahead characters
  - Need specialized buffer managing technique to improve efficiency

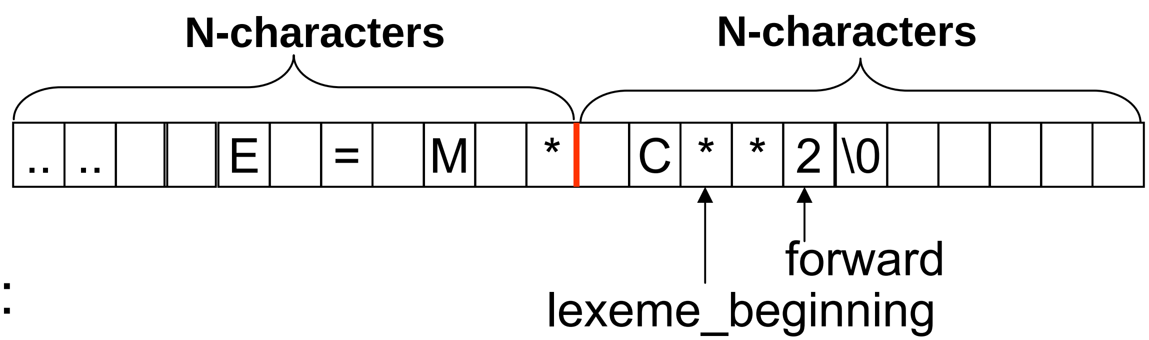
Buffer Pairs

- Token could overlap / span buffer boundaries



12.46

- Solution: Use a paired buffer of N characters each



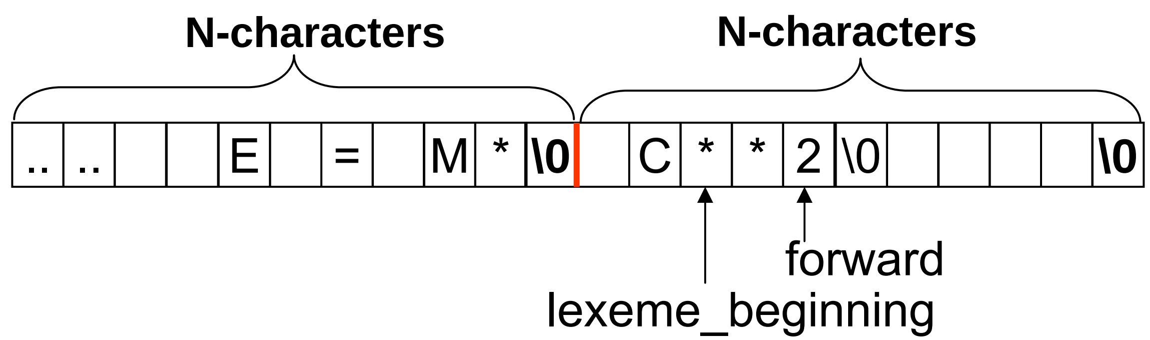
- Deficiency:

Code:

```
if (forward at end of buffer1) then
        relaod buffer2;
        forward = forward +1;
else if (forward at end of buffer2) then
        relaod first half;
        move forward to the beginning of the buffer1
else
        forward = forward +1
```

Sentinels

- Technique: Use “Sentinels” to reduce testing
- Choose some character that occurs rarely in most inputs e.g. **‘\0’**




```
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
```

Terminology

- **Alphabet (Σ) : AKA character class**
  - A set of symbols ("characters")
  - *Examples:* Σ = { **1, 0**} : binary alphabet
    Σ = { **1, 2, 3, 4, 5, 6** } : Alphabet on dice outcome
- **String : AKA Sentence** or **word**
  - Sequence of symbols
  - Finite in length
  - *Example:* **abbadc** Length of s = |s|
- **Empty String (ε)**
  - It is a string
  - $| \varepsilon | = 0$
- **Language**
  - A set of strings over some fixed alphabet
  - *Examples:* L**1** = { **a, baa, bccb** }
    L**2** = { }
    L**3** = {ε}
    Note the difference
    L**4** = {ε, **ab, abab, ababab, abababab,...** }

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

Terminology

- **Prefix** ...of string s
  - String obtained by removing zero or more trailing symbols
  - s = **hello**
  - *Prefixes: ε, h, he, hel, hell, hello*
- **Suffix** ...of string s
  - String obtained by deleting zero or more of the leading symbols
  - s = **hello**
  - *Suffixes: hello, ello, llo, lo, o, ε*
- **Substring** ...of string s
  - String obtained by deleting a prefix and a suffix
  - s = **hello**
  - *Substrings: ε. ell, hel, llo, hello, ….*
- **Proper prefix / suffix / substring** ... of s
  - String s1 that is respectively prefix, suffix, or substring of s such that s1 $\neq$ s and s1 $\neq \varepsilon$
- **Subsequence**….of string s
  - String formed by deleting zero or more not necessarily contiguous symbols
  - S=hello
  - Subsequence: hll, eo, hlo, etc.

Terminology

"Concatenation"

Strings: x, y

Concatenation: xy

Example:

$x = \text{abb}$

$y = \text{cdc}$

$xy = \text{abbcdc}$

$yx = \text{cdcabb}$

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 $x = \text{ab}$

$x^0 = \varepsilon$

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

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

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

Terminology

- Language
  - A set of strings
  - L = { ... }
  - M = { ... }
- Union of two languages
  - $L \cup M$ = { s | s is in L 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 | s is in L and t is in M }
  - *Example:*
  - L = { **a**, **ab** }
  - M = { **c**, **dd** }
  - L M = { **ac**, **add**, **abc**, **abdd** }

Kleene closure

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

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

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

Positive closure

- *Let:* L = { **a**, **bc** }
- *Example:* $L^{\mathbf{0}} = \{ \varepsilon \}$

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

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

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

  ...etc...

  $L^{\mathbf{N}} = L^{\mathbf{N-1}}L = LL^{\mathbf{N-1}}$
- *The "Positive Closure" of a language:*

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

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

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

Example

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"*

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

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

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

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.

Regular Expressions and Language

- (Over alphabet Σ)
- And, given a regular expression **R**, what is L(**R**) ?
- ε is a regular expression.
  - L(ε) = { ε }
- 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.

How to “Parse” Regular Expressions

- **Precedence:**
  - ***** 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)**

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 $\subset$ 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 vs 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 $\cong$ S

  "R is equivalent to S"

- From now on, we'll just say R **=** S to mean R $\cong$ S

Algebraic law of regular expressions

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

$\mid$ is commutative

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

$\mid$ 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 $\mid$

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

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

ε is the identity for concatenation

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

$*$ is idempotent

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

Relation between $*$ and ε

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

Regular Definition

- If Σ is an alphabet of basic symbols then a regular definition is a sequence of the following form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots\ldots..$$
$$d_n \rightarrow r_n$$

where

- Each d_i is a new symbol such that $d_i \notin \Sigma$ and $d_i \neq d_j$ where j < I
- Each r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1})$

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", ... }

Each definition may only use names *previously* defined.

$\Rightarrow$ 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): ?

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

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]

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

Nonregular sets

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:

1. Hand-coded routines
2. Finite State Automata
3. 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.)