THE LEXICAL Analyzer

- **Main role:** split the input character stream into tokens
  - Usually even interacts with the symbol table, inserting identifiers in it (especially useful for languages that do not require declarations)
  - This simplifies the design and portability of the parser

- **A token** is a data structure that contains:
  - The *token name* = abstract symbol representing a kind of lexical unit
  - A possibly empty set of *attributes*

- A *pattern* is a description of the form recognized in the input as a particular token

- A *lexeme* is a sequence of characters in the source program that matches a particular pattern of a token and so represents an instance of that token

Most programming languages feature the following tokens

- One token for each keyword
- One token for each operator or each class of operators (e.g., relational operators)
- One token for all identifiers
- One or more tokens for literals (numerical, string, etc.)
- One token for each punctuation symbol (parentheses, commata, etc.)

EXAMPLE OF TOKENS AND ATTRIBUTES

```c
printf("Score = %d\n", score);
```

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf</td>
<td>id</td>
<td>pointer to symbol table entry</td>
</tr>
<tr>
<td>(</td>
<td>open_paren</td>
<td></td>
</tr>
<tr>
<td>&quot;Score = %d\n&quot;</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td></td>
</tr>
<tr>
<td>score</td>
<td>id</td>
<td>pointer to symbol table entry</td>
</tr>
<tr>
<td>)</td>
<td>cls_paren</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
<td></td>
</tr>
</tbody>
</table>

```c
E = M * C ** 2
```

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>id</td>
<td>pointer to symbol table entry</td>
</tr>
<tr>
<td>=</td>
<td>assign</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>id</td>
<td>pointer to symbol table entry</td>
</tr>
<tr>
<td>*</td>
<td>mul</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>id</td>
<td>pointer to symbol table entry</td>
</tr>
<tr>
<td>**</td>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>int_num</td>
<td>numerical value 2</td>
</tr>
</tbody>
</table>

INPUT BUFFERING

- Buffering is often used to speed up the process of recognizing lexemes
  - Also facilitates the process of looking ahead beyond the current lexeme

- **Typical buffer arrangement:**
  - Two buffers of size $N = \text{the size of a disk sector (usually 4096 bytes)}$
  - One buffer is loaded while the other is being processed
  - One system call fills in a whole buffer
  - Two pointers per buffer: `lexemeBegin` (the beginning of the current lexeme) and `forward` (moves forward until a pattern is found, but can also move backward)

- **Problem:** each time we advance the forward pointer we need to tests:
  - one for the current character, the other for the end of the buffer

  - **Solution:** place a special *sentinel* character (e.g., `EOF`) at the end of the buffer
  - A single test will then suffice
**Specification of Tokens**

- Token patterns are simple enough so that they can be specified using regular expressions.
- **Alphabet** $\Sigma$: a finite set of symbols (e.g., binary digits, ASCII).
- **Strings** (not sets!) over an alphabet; empty string: $\varepsilon$.
  - Useful operation: concatenation ($\cdot$ or juxtaposition).
  - $\varepsilon$ is the identity for concatenation ($\varepsilon w = w = w \varepsilon$).
- **Language**: a countable set of strings.
  - Abuse of notation: For $a \in \Sigma$ we write $a$ instead of $\{a\}$.
  - Useful elementary operations:
    - Union ($\cup$ or $+$) and concatenation ($\cdot$ or juxtaposition): $L_1 \cup L_2 = \{ w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2 \}$.
    - Exponentiation: $L^n = \{ w^n : w \in L \}$ (so that $L^0 = \{ \varepsilon \}$).
    - Kleene closure: $L^* = \bigcup_{n \geq 0} L^n$.
    - Positive closure: $L^+ = \bigcup_{n > 0} L^n$.
- An expression containing only symbols from $\Sigma$, $\varepsilon$, $\emptyset$, union, concatenation, and Kleene closure is called a regular expression.
  - A language described by a regular expression is a regular language.

---

**Examples of Regular Definitions**

- **letter**: $[A−Za−z]$.
- **digit**: $[0−9]$.
- **id**: letter_ $(\text{letter}_- | \text{digit})^*$.
- **digits**: digit*.
- **fraction**: . digits.
- **exp**: $E [\{+-\}]$ digits.
- **number**: digits fraction? exp?.
- **if**: $i \ f$.
- **then**: $t h e n$.
- **else**: $e l s e$.
- **rel_op**: $< | > | <= | >= | == | !=$.

---

**Syntactic Sugar for Regular Expressions**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Regular expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r^+$</td>
<td>$rr^*$</td>
</tr>
<tr>
<td>$r?$</td>
<td>$r</td>
</tr>
<tr>
<td>$[a_1 a_2 \cdots a_n]$</td>
<td>$a_1</td>
</tr>
<tr>
<td>$[a_1 a_2 \cdots a_n]$</td>
<td>$[a_1</td>
</tr>
</tbody>
</table>

---

**State Transition Diagrams**

- Also called deterministic finite automata (DFA).
- Finite directed graph.
- Edges (transitions) labeled with symbols from an alphabet.
- Nodes (states) labeled only for convenience.
- One initial state.
- Several accepting states.
- A string $c_1 c_2 c_3 \ldots c_n$ is accepted by a state transition diagram if there exists a path from the starting state to an accepting state such that the sequence of labels along the path is $c_1, c_2, \ldots, c_n$.

  - Same state might be visited more than once.
  - Intermediate states might be final.
- The set of exactly all the strings accepted by a state transition diagram is the language accepted (or recognized) by the state transition diagram.
software realization

big practical advantages of dfa: very easy to implement:
structure to define a vocabulary and a function to obtain the input tokens

typename vocab; /* alphabet + end-of-string */
const vocab EOS; /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */

variable (state) changed by a simple switch statement as we go along

int main (void) {
typedef enum {s0, s1, ... } state;
state s = s0; vocab t = gettoken();
while ( t != eos ) {
    switch (s) {
    case s0: if (t == ... ) s = ...; break;
    if (t == ...) s = ...; break;
    ...
    case s1: ...
    ...
    } /* switch */
    t = gettoken(); } /* while */
/* accept iff the current state s is final */
}

examples of state transition diagrams

when returning from *-ed states must retract last character

lex, the lexical analyzer generator

the lex language is a programming language particularly suited for working with regular expressions
actions can also be specified as fragments of c/c++ code

the lex compiler compiles the lex language (e.g., scanner.l) into c/c++ code (lex.yy.c)
the resulting code is then compiled to produce the actual lexical analyzer
the use of this lexical analyzer is through repeatedly calling the function yylex() which will return a new token at each invocation
the attribute value (if any) is placed in the global variable yylval
additional global variable: yytext (the lexeme)

structure of a lex program:

declarations

% translation rules
% auxiliary functions

declarations include variables, constants, regular definitions
transition rules have the form

pattern { action }

where the pattern is a regular expression and the action is arbitrary c/c++ code
**Lex Behaviour**

- **LEX** compile the given regular expressions into one big state transition diagram, which is then repeatedly run on the input.
- **LEX** conflict resolution rules:
  - Always prefer a longer to a shorter lexeme.
  - If the longer lexeme matches more than one pattern then prefer the pattern that comes first in the **LEX** program.
- **LEX** always reads one character ahead, but then retracts the lookahead character upon returning the token.
  - Only the lexeme itself is therefore consumed.

**Nondeterministic State Transition Diagrams**

- **Deterministic** = for any pair (state, input symbol) there can be at most one outgoing transition.
- A **nondeterministic** diagram allows for the following situation:
  - The acceptance condition remains unchanged:
    - A string $c_1c_2c_3\ldots c_n$ is accepted by a state transition diagram if there exists some path from the starting state to an accepting state such that the sequence of labels along the path is $c_1$, $c_2$, $\ldots$, $c_n$.
- Why nondeterminism?  
  - **Simplifies the construction** of the diagram.
    - A nondeterministic diagram can be **much smaller** than the smallest possible deterministic state diagram that recognizes the same language.
  - **Also known as** nondeterministic finite automata (NFA).

**Software Realization**

- As for the deterministic version, except that we have to keep track of a set of states at any given time.
  ```c
typedef enum { Q0, Q1, Q2, Q3 } state;

int main (void) {
  vocab t = gettoken(); StateSet A; A.include(Q0);
  while (t != EOS) {
    StateSet NewA;
    for (state s in A) {
      switch (s) {
        case Q0: NewA.include(Q0);
          if (t == 'm') NewA.include(Q1); break;
        case Q1: if (t == 'a') NewA.include(Q2); break;
        case Q2: if (t == 'n') NewA.include(Q3); break;
        case Q3: break;
      }
      A = NewA; t = gettoken();
    } /* accept iff (Q3 in A) */
  }
}
```

**Software Realization (cont’d)**

- This kind of implementation is fine for “throw-away” automata.
  - Text editor search function searches for a pattern in the text.
    - The next search is likely to be different so a brand new automaton needs to be created.
- Some times the automaton is created once and then used multiple times.
  - The lexical structure of a programming language is well established.
    - Lexical analysis in a compiler is accomplished by an automaton that never changes.
    - In such a case it is more efficient to **precalculate the set of states**.
      - Exactly as in the previous program.
      - Except that we no longer have an input to guide us, so we calculate the sets NewA for all possible inputs.
      - We obtain a DFA that is **equivalent** to the given NFA (i.e., recognizes the same language).
ε-TRANSITIONS

- Useful at times to have “spontaneous” transitions = transitions that change the state without any input being read = ε-transitions
- Only available for nondeterministic state transition diagrams!
- Example of usefulness: Construct the state transition diagram for the language
  \[
  \{0, 1\}^* 01 \{0, 1\}^* + \{w \in \{0, 1\}^* : w \text{ has an even number of } 1\text{'s}\}
  \]
- Even better ε-transitions can be eliminated afterward

From Regular Expressions to FA

- Construct a finite automaton for every elementary regular expression (ε, x ∈ Σ, ()):  
  \[
  \begin{array}{c}
  \epsilon \\
  x \\
  \end{array}
  \]
- Then starting from component finite automata we show how we can construct finite automata for each possible operator appearing in regular expressions (+, *, )
  - Useful operation: merging two states
  - Properties to be maintained:
    - One accepting state
    - Initial state different from the accepting state
    - No transitions out of the accepting state

ELIMINATING ε-TRANSITIONS

For every diagram \( M \) with ε-transitions a new diagram without ε-transitions can be constructed as follows:

- Make a copy \( M' \) of \( M \) where the ε-transitions have been removed.
- Remove states that have only ε-transitions coming in except for the starting state
- Add transitions to \( M' \) as follows: whenever \( M \) has a chain of ε-transitions followed by a “real” transition on \( x \):
  \[
  \begin{array}{c}
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \end{array} \xrightarrow{\cdot} \begin{array}{c}
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \end{array} \xrightarrow{x} \begin{array}{c}
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \end{array} \xrightarrow{\cdot} \begin{array}{c}
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \epsilon \\
  \end{array}
  \]
- Note that \( q \) and \( p \) may be any states
- In particular this step is also used in the case where \( q = p \)
- If \( M \) has a chain of ε-transitions from a state \( r \) to an accepting state, then \( r \) is made to be an accepting state of \( M' \).
All regular expressions can be converted step by step to the equivalent finite automaton by using these constructions:

- **Concatenation**
- **Closure**

### Example: $(\varepsilon + 0)^*0$

- FA for $\varepsilon$
- FA for $0$
- FA for $\varepsilon + 0$
- FA for $(\varepsilon + 0)^*$
- FA for $0$
- FA for $0^*$
- FA for $(\varepsilon + 0)^*0$
- FA for $1$
- FA for $1(\varepsilon + 0)^*0$

The finite automaton thus obtained can either be converted into a deterministic finite automaton or realized as is.