Lexical Analysis

Main task: to read input characters and divide them into a sequence of tokens.

Secondary tasks:
- skip whitespace where necessary;
- skip comments;
- correlate error messages with source program (e.g., line no. of error)

Approaches to implementing Lexical Analyzers

- Use a lexical analyzer generator, e.g., lex or flex. This automatically generates a lexical analyzer given some specifications.
- Write the lexical analyzer in a conventional systems programming language, using the I/O facilities of that language.
**Lexical Analysis: some terminology**

**token**: a name for a class of strings in the input.
   
   *Example*: “identifier”

**pattern**: a rule that describes the set of strings associated with a token.
   
   *Example*: “a letter followed by zero or more letters, digits, or underscores”.

**lexeme**: the actual input string that matches a pattern.
   
   *Example*: `count`.

```
count  =  1

 token: ident  token: assign_op  token: int_const
 pattern: ...  pattern: '='    pattern: ...
 lexeme: count  lexeme: =     lexeme: 1
```
**Attributes for tokens**

If more than one lexeme can match a token, the lexical analyser must provide information about the specific lexeme that matched. Such information is given as an **attribute** associated with the token.

**Example**: The program statement

```
count = 1
```

gives rise to the sequence of token-attribute pairs

```
{id, ptr to buffer containing lexeme}
{assg_op, }
{int_con, 1}
```

---

**Sec.2.2: Specifying Tokens: Regular Expressions**

**string**: a sequence of symbols drawn from an alphabet.

**language**: a set of strings over a fixed alphabet.

**Operations on Languages**

**union**: \(L \cup M = \{s \mid s \in L \text{ or } s \in M\}\).

**concatenation**: \(LM = \{st \mid s \in L, t \in M\}\).

**Kleene closure**: \(L^* = \bigcup_{i=0}^{\infty} L^i = \{\varepsilon\} \cup L \cup LL \cup \cdots\).
**Regular Expressions**: a notation for describing certain kinds of sets of strings.

**Rules for REs over an alphabet \( \Sigma \):**

1. \( \varepsilon \) is a RE : denotes \( \{ \varepsilon \} \)
2. if \( a \in \Sigma \) then \( a \) is a RE : denotes \( \{a\} \)
3. if \( r \) and \( s \) are REs denoting languages \( L_r \) and \( L_s \) respectively, then:
   (a) \( r \mid s \) is a RE : denotes \( L_r \cup L_s \);
   (b) \( rs \) is a RE : denotes \( L_rL_s \);
   (c) \( r^* \) is a RE : denotes \( L_r^* \);

Use parentheses when multiple operators are used in a regular expression.

---

**Extensions to Regular Expression Notation**

- **Giving names to regular expressions**:
  - letter = a | b | ... | z | A | B | ... | Z
  - digit = 0 | 1 | ... | 9
  - id = letter(letter | digit)*
  - intcon = digit+

- **One or more repetitions** : \( r^+ \)

- **Any character** : .

- **A range of characters** : [a-zA-Z], [0-9]

- **Optional subexpressions** : \( r? \)
Example

Identifiers like: A1BC_3A_B5
- Identifier cannot end in an underscore
- It cannot contain two consecutive underscores

Solution: Divide the identifier into three parts

   A  1BC    _3A_B5

   L  (L|D)*  (_(L|D)*)*

Sec. 2.3: Finite Automata

Definition: A (deterministic) finite automaton is a 5-tuple \( (Q, \Sigma, T, q_0, F) \), where:

- \( Q \): a (finite) set of states
- \( \Sigma \): a (finite) alphabet
- \( T: Q \times \Sigma \to Q \): transition function
- \( q_0 \in Q \): initial state
- \( F \subseteq Q \): set of final states
2.3.1: Deterministic Finite Automata

To define the behavior of a FA on a string, extend $T$ to apply to a state and a string:

$$\hat{T} : Q \times \Sigma^* \rightarrow Q$$

$$\hat{T}(q, \varepsilon) = q$$

$$\hat{T}(q, w a) = T(\hat{T}(q, w), a)$$

Language accepted by a DFA $M$:

$$L(M) = \{ w \in \Sigma^* \mid \hat{T}(q_0, w) \in F \}$$

Regular Languages: A language is regular if it is accepted by some DFA.

Sec. 2.4: Equivalence of REs and DFAs

DFAs are equivalent to REs:

- For every regular expression $r$, there is a DFA $M$ such that $L(r) = L(M)$.
- For every DFA $M$, there is a regular expression $r$ such that $L(M) = L(r)$.

$\Rightarrow$ the class of languages accepted by DFAs is exactly the same as the class of regular languages.
Example of a DFA:

Want to match: C-style comments

DFA:

```
RE: / * (not *)* ( *+ (not /) (not *)* )* *+ /`
```

Specification and recognition of tokens:

- tokens specified using regular expressions.
- our aim is to write a program to recognize tokens.
- use an intermediate step: “transition diagram” or “finite automaton”.

Actually, it isn’t enough for the finite automaton to just accept (a portion of) its input: also need to do the following:

1. At the end, when we see a character that does not match (i.e., we are done matching this token), we have to “retract” one character.
2. We may have to also return (a pointer to) the lexeme for the token.
Handling tokens with optional parts:

If parts of a token can be optional, we should try to return the longest possible string for it. E.g. if we have

\[ \text{num} \rightarrow \text{digit}^+ (\ \cdot\ \text{digit}^+ | \varepsilon) \]

then the input 123.45 can be matched as either

\[ \text{num}(123), \ \cdot, \ \text{num}(45) \]

or as

\[ \text{num}(123.45) \]

The second match is the desired one.

Handling tokens with optional parts: cont’d

Conceptually, given the regular expression

\[ \text{num} \rightarrow \text{digit}^+ (\ \cdot\ \text{digit}^+ | \varepsilon) \]

we have two finite automata for \text{num}:

![Finite automata diagram]

CSE Department
Since we want to match the longest sequence possible, we combine these as follows:

![Diagram](image)

(this arc is not taken if ‘.’ is seen)

**Implementation Issues**

- Handling reserved words
- Implementing finite automaton
- Input buffering
Handling Reserved Words

**option 1**: have separate finite automaton for each reserved word.

1. Large no. of states in finite automaton, hence large code size.
2. Changes are harder to make.

**option 2**: put reserved words in a table, search this table when an identifier is found.

1. Simpler, easier to test and debug.
2. Fewer states, smaller code size.
3. Easier to modify

Implementing finite automata:

- Each state is implemented by a segment of code.
- If there are edges leaving a state, then the code for that state reads a character and selects an edge to follow (e.g. by `switch` statement).
- If the next input character does not match any outgoing edge, invoke a routine `fail()` to retract the pointer into the input buffer to the starting position, and try the next finite automaton.
- **To return tokens**: return the type of the token, i.e. `id`, `num`, etc.; use a global to hold the associated lexeme, if necessary.
Implementing Finite Automaton

Table Driven
- Index into a table using <state#, input ASCII code>
- Table entry gives the next state
- Use negative entries to signal actions like errors

Implementation Concerns: Input Buffering

Because lexical analysis involves reading the source program one character at a time, its speed is a concern in compiler design.

Lexical analyzers typically need some lookahead to determine whether a match has been attained. To minimize the overhead associated with this, specialized buffering schemes can be used.
Buffer Pairs:

- Basic idea: use a buffer divided into two $N$-character portions: conceptually these form a circular buffer. Typically, $N$ is the size of a disk block, e.g. 1024 or 4096.

- Read $N$ characters at a time into one half of the buffer each time. If input contains less than $N$ characters, put a special EOF marker in the buffer.

- Maintain two pointers into the buffer:
  - lexeme points to the beginning of the current lexeme;
  - fwd scans forward until the end of the lexeme is found.

Buffer Pairs: cont’d

- move fwd forward until the end of a lexeme is found;

- after a lexeme is processed, both pointers are set to the character immediately past that lexeme;

- comments and whitespace can be treated as lexemes that yield no token.

- If the forward pointer is about to move past the end of one half of the buffer, the other half is filled with $N$ new characters.
**Input Buffering**: cont’d

\[
x = x \cdot (y + 1)
\]

The code is of the form

```
if fwd at end of first half then
    reload second half;
    fwd := fwd + 1;
else if fwd at end of second half then
    reload first half;
    move fwd to beginning of first half;
else fwd := fwd + 1;
```

*Note*: Except at the end of each buffer half, it takes two tests for each advance of the fwd pointer.

---

**Buffer Pairs: sentinels**

```
count = 1; EOF EOF
```

*Objective*: Try to reduce the no. of tests per advance of fwd from 2 to 1.

*Idea*: Extend each buffer half to hold a sentinel at each end. *This must be a character that cannot occur in the input program* ⇒ use the EOF marker.
Code:

\[
\begin{align*}
\text{fwd} & := \text{fwd} + 1; \\
\text{if} & \text{ fwd points at EOF then} \\
& \text{ if fwd at end of first half then} \\
& \quad \text{reload second half;} \\
& \quad \text{fwd} := \text{fwd} + 1; \\
& \text{else if fwd at end of second half then} \\
& \quad \text{reload first half;} \\
& \quad \text{move fwd to beginning of first half;} \\
& \text{else} \quad /* \text{ must be end of input */} \\
& \quad \text{terminate lexical analysis.}
\end{align*}
\]