CSc 453
Lexical Analysis
(Scanning)

Saumya Debray
The University of Arizona
Tucson

Overview

- **Main task**: to read input characters and group them into "tokens."
- **Secondary tasks**:
  - Skip comments and whitespace;
  - Correlate error messages with source program (e.g., line number of error).
Overview (cont’d)

Implementing Lexical Analyzers

Different approaches:

- Using a scanner generator, e.g., lex or flex. This automatically generates a lexical analyzer from a high-level description of the tokens. (easiest to implement; least efficient)
- Programming it in a language such as C, using the I/O facilities of the language. (intermediate in ease, efficiency)
- Writing it in assembly language and explicitly managing the input. (hardest to implement, but most efficient)
Lexical Analysis: Terminology

- **token**: a name for a set of input strings with related structure.
  
  **Example**: “identifier,” “integer constant”

- **pattern**: a rule describing 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

Examples

**Input**: count = 123

**Tokens**:

- **identifier**: Rule: “letter followed by …”
  
  **Lexeme**: count

- **assg_op**: Rule: =
  
  **Lexeme**: =

- **integer_const**: Rule: “digit followed by …”
  
  **Lexeme**: 123
Attributes for Tokens

- If more than one lexeme can match the pattern for a token, the scanner must indicate the actual lexeme that matched.
- This information is given using an attribute associated with the token.

*Example:* The program statement

```plaintext
count = 123
```

eyields the following token-attribute pairs:

- `<identifier, pointer to the string “count”>`
- `<assg_op,>`
- `<integer_const, the integer value 123>`

Specifying Tokens: regular expressions

- **Terminology:**
  - *alphabet*: a finite set of symbols
  - *string*: a finite sequence of alphabet symbols
  - *language*: a (finite or infinite) set of strings.

- **Regular Operations on languages:**
  - **Union:** $R \cup S = \{x \mid x \in R \text{ or } x \in S\}$
  - **Concatenation:** $RS = \{xy \mid x \in R \text{ and } y \in S\}$
  - **Kleene closure:** $R^* = R$ concatenated with itself 0 or more times
    - $= \epsilon \cup R \cup RR \cup RRR \cup \ldots$
    - = strings obtained by concatenating a finite number of strings from the set $R$. 
Regular Expressions

A pattern notation for describing certain kinds of sets over strings:

Given an alphabet $\Sigma$:
- $\varepsilon$ is a regular exp. (denotes the language \{\varepsilon\})
- for each $a \in \Sigma$, $a$ is a regular exp. (denotes the language \{a\})
- if $r$ and $s$ are regular exps. denoting $L(r)$ and $L(s)$ respectively, then so are:
  - $(r) | (s)$ (denotes the language $L(r) \cup L(s)$)
  - $(r)(s)$ (denotes the language $L(r)L(s)$)
  - $(r)^*$ (denotes the language $L(r)^*$)

Common Extensions to r.e. Notation

- One or more repetitions of $r : r^+$
- A range of characters : `[a-zA-Z], [0-9]`
- An optional expression: $r?$
- Any single character: .
- Giving names to regular expressions, e.g.:
  - `letter = [a-zA-Z_]`
  - `digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
  - `ident = letter ( letter | digit )*`
  - `Integer_const = digit+`
Recognizing Tokens: Finite Automata

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

- \(\Sigma\) is a finite alphabet;
- \(Q\) is a finite set of states;
- \(T: Q \times \Sigma \to Q\) is the transition function;
- \(q_0 \in Q\) is the initial state; and
- \(F \subseteq Q\) is a set of final states.

Finite Automata: An Example

A (deterministic) finite automaton (DFA) to match C-style comments:
Formalizing Automata Behavior

To formalize automata behavior, we extend the transition function to deal with strings:

\[ T : Q \times \Sigma^* \rightarrow Q \]

\[ T(q, \varepsilon) = q \]

\[ T(q, aw) = T(r, w) \text{ where } r = T(q, a) \]

The language accepted by an automaton \( M \) is

\[ L(M) = \{ w \mid T(q_0, w) \in F \}. \]

A language \( L \) is regular if it is accepted by some finite automaton.

Finite Automata and Lexical Analysis

- The tokens of a language are specified using regular expressions.
- A scanner is a big DFA, essentially the “aggregate” of the automata for the individual tokens.
- Issues:
  - What does the scanner automaton look like?
  - How much should we match? (When do we stop?)
  - What do we do when a match is found?
  - Buffer management (for efficiency reasons).
Structure of a Scanner Automaton

How much should we match?

In general, find the longest match possible. E.g., on input 123.45, match this as
num_const(123.45)
rather than
num_const(123), ".", num_const(45).
Input Buffering

- Scanner performance is crucial:
  - This is the only part of the compiler that examines the entire input program one character at a time.
  - Disk input can be slow.
  - The scanner accounts for ~25-30% of total compile time.
- We need lookahead to determine when a match has been found.
- Scanners use *double-buffering* to minimize the overheads associated with this.

Buffer Pairs

- Use two $N$-byte buffers ($N =$ size of a disk block; typically, $N = 1024$ or $4096$).
- Read $N$ bytes into one half of the buffer each time. If input has less than $N$ bytes, put a special EOF marker in the buffer.
- When one buffer has been processed, read $N$ bytes into the other buffer ("circular buffers").
Buffer pairs (cont’d)

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

**Code:**

```
if (fwd at end of first half)
    reload second half;
    set fwd to point to beginning of second half;
else if (fwd at end of second half)
    reload first half;
    set fwd to point to beginning of first half;
else
    fwd++;
```

It takes two tests for each advance of the fwd pointer.

---

Buffer pairs: Sentinels

- **Objective:** Optimize the common case by reducing the number of tests to one per advance of fwd.
- **Idea:** Extend each buffer half to hold a **sentinel** at the end.
  - This is a special character that cannot occur in a program (e.g., EOF).
  - It signals the need for some special action (fill other buffer-half, or terminate processing).
Buffer pairs with sentinels (cont’d)

Code:
fwd++;     
if ( *fwd == EOF ) { /* special processing needed */ 
    if (fwd at end of first half) 
        ...
    else if (fwd at end of second half) 
        ...
    else    /* end of input */  
        terminate processing.
} 
common case now needs just a single test per character.

Handling Reserved Words

1. Hard-wire them directly into the scanner automaton:
   ● harder to modify;
   ● increases the size and complexity of the automaton;
   ● performance benefits unclear (fewer tests, but cache effects due to larger code size).

2. Fold them into “identifier” case, then look up a keyword table:
   ● simpler, smaller code;
   ● table lookup cost can be mitigated using perfect hashing.
Implementing Finite Automata 1

Encoded as program code:
- each state corresponds to a (labeled) code fragment
- state transitions represented as control transfers.

E.g.:
```c
while ( TRUE ) {
  ... 
  state_k: ch = NextChar(); /* buffer mgt happens here */
  switch (ch) {
    case ... : goto ...; /* state transition */
    ...
  }
  state_m: /* final state */
  copy lexeme to where parser can get at it;
  return token_type;
  ...
}
```

Direct-Coded Automaton: Example

```c
int scanner() {
  char ch;
  while (TRUE) {
    ch = NextChar(); /* initial state */
    switch (ch) {
      case 'a': goto state_2;
      case 'b': goto state_3;
      default: Error();
    }
    state_2: ...
    state_3: switch (ch) {
      case 'a': goto state_2;
      default: return SUCCESS;
    }
  } /* while */
}
```
Implementing Finite Automata 2

Table-driven automata (e.g., lex, flex):
- Use a table to encode transitions:
  \[ \text{next} \_ \text{state} = T(\text{curr} \_ \text{state}, \text{next} \_ \text{char}); \]
- Use one bit in state no. to indicate whether it’s a final (or error) state. If so, consult a separate table for what action to take.

\[
\begin{array}{c|c|c|c}
\text{Current state} & 1 & 2 & 3 \\
\hline
\text{next input character} & & & \\
\end{array}
\]

Table-Driven Automaton: Example

```c
#define isFinal(s)   ((s) < 0)
int scanner()
{
  char ch;
  int currState = 1;

  while (TRUE) {
    ch = NextChar( );
    if (ch == EOF) return 0; /* fail */
    currState = T[currState, ch];
    if (isFinal(currState)) {
      return 1; /* success */
    }
  } /* while */
}
```

\[
\begin{array}{|c|c|c|}
\hline
\text{state} & \text{input} & \text{T} \\
\hline
1 & 2 & 3 \\
2 & 2 & 3 \\
3 & 2 & -1 \\
\hline
\end{array}
\]
What do we do on finding a match?

- A match is found when:
  - The current automaton state is a final state; and
  - No transition is enabled on the next input character.

- Actions on finding a match:
  - if appropriate, copy lexeme (or other token attribute) to where the parser can access it;
  - save any necessary scanner state so that scanning can subsequently resume at the right place;
  - return a value indicating the token found.