Practical Issues In Lexical Analysis

- Until now, we have abstracted the lexical analysis task as the recognition problem: "Is $s \in L(r)$?"

- In reality:
  - We have a string t.
  - We have to repeatedly find maximal prefixes s of t such that $s \in L$, remove them from t, and return appropriate tokens and attributes.

- Also, we need to handle errors meaningfully.

Issue #1: Returning Tokens

- Tag each final state with the token that it matches, and return that token.
  - Some final states may not return tokens (e.g., whitespace, comments).
- Use a trap state to simplify finding the longest nonempty prefix of the input string that matches. (“Principle of maximal scan”)
- Example: recognizer for integer literals.

Issue #2: Lookahead and Pushback

- Suppose the input string is `123<x` (i.e., no whitespace characters).
  - The integer literal `123` is recognized only after scanning the next character.
  - Modern HLLs usually require a small number of lookahead characters.
  - Not so in the case in older languages such as Fortran:

```
DO 5 I = 1, 8
DO 5 I = 1.8
```

- Consider a recognizer for relational operators <, <=, =, >=, >.

- Need to push back the lookahead character on the input stream.

Issue #3: DFA Simulation

- Two approaches: table-driven, direct encoding.
  - Table-driven

    Encode the transition diagram as an array indexed by the current state and the class of the character being scanned.

    ```
    state = nextstate[state][map[nextchar]];
    ```

  - Direct coding

    Encode the transition diagram directly, using `while`-loops and `switch` statements.

Issue #3: DFA Simulation

- Two approaches: table-driven, direct encoding.
  - Table-driven

    Encode the transition diagram as an array indexed by the current state and the class of the character being scanned.

    ```
    state = nextstate[state][map[nextchar]];
    ```

  - Direct coding

    Encode the transition diagram directly, using `while`-loops and `switch` statements.
- Efficiency concerns
  - Minimize the cost of processing single source characters. Typically, there are 5-10 times as many characters as tokens in the input.
  - Minimize the cost of skipping single spaces.

Issue #4: Input Buffering

- Operations on input stream
  - Read the next character (frequent).
  - Push back lookahead token (occasional).
- Use specialized buffer management within the lexer.
  - E.g., double buffering with an EOF sentinel.

Issue #5: Conversion

- The process of obtaining the representation of a lexeme that has been scanned.
  - May involve finding a value, a symbol, or some other attribute(s) of the lexeme.

- Different kinds of tokens may need different kinds of conversion.
  - The lexeme `123`: *tokentype* = **INT**, *value* = 123.
  - The lexeme `abc`: *tokentype* = **ID**, *namestring* = “abc”.
  - The lexeme `while`: *tokentype* = **KWD**.
  - A string lexeme: May need to handle continuation characters or convert escape sequences.

Issue #6: Keyword Recognition

- Are keywords reserved?
  - Yes, in pretty much all modern programming languages.
  - But consider PL/I, where they were not.

```
IF THEN THEN THEN := ELSE; ELSE ELSE := THEN;
```

Issue #6: Keyword Recognition

- Are keywords reserved?
  - Yes, in pretty much all modern programming languages.
  - But consider PL/I, where they were not.

```
IF THEN THEN THEN := ELSE; ELSE ELSE := THEN;
```

- Two approaches
  - Separate keywords from identifiers in scanning.
  - Separate keywords from identifiers during conversion.