Outline

1. Lexical Analysis
2. Regular Expressions
3. State Diagrams
4. Assignment
A token has a type and a value.

Types include **id**, **num**, **assign**, **lparen**, etc.

Values are used primarily with identifiers and numbers.

If we read “**count**”, the type is **id** and the value is “**count**”.

If we read “**123**”, the type is **num** and the value is “**123**”.

If we read “**=**”, the type is **assign** and the value is “**=**”.
Analyzing Tokens

- Each type of token can be described by a regular expression.
- Therefore, the set of all tokens can be described by a regular expression. (Why?)
- Regular expressions are accepted by DFAs.
- Therefore, the set of all tokens can be processed and accepted by a DFA.
Regular Expressions

- The set of all regular expressions may be defined in two parts.
- The basic part:
  - $\varepsilon$ represents the language $\{\varepsilon\}$.
  - $a$ represents the language $\{a\}$ for every $a \in \Sigma$.
  - Call these languages $L(\varepsilon)$ and $L(a)$, respectively.
Regular Expressions

- The recursive part: Let \( r \) and \( s \) denote regular expressions.
  - \( r \mid s \) represents the language \( L(r) \cup L(s) \).
  - \( rs \) represents the language \( L(r)L(s) \).
  - \( r^* \) represents the language \( L(r)^* \).

- In other words
  - \( L(r \mid s) = L(r) \cup L(s) \).
  - \( L(rs) = L(r)L(s) \).
  - \( L(r^*) = L(r)^* \).
Example (Identifiers)

- Identifiers in C++ can be represented by a regular expression.

\[
\begin{align*}
  r &= A | B | \cdots | Z | a | b | \cdots | z \\
  s &= 0 | 1 | \cdots | 9 \\
  t &= r(r | s)^* 
\end{align*}
\]
A regular definition of a regular expression is a “grammar” of the form

\[
\begin{align*}
  d_1 & \rightarrow r_1 \\
  d_2 & \rightarrow r_2 \\
  & \vdots \\
  d_n & \rightarrow r_n
\end{align*}
\]

where each \( r_i \) is a regular expression over \( \Sigma \cup \{ d_1, d_2, \ldots, d_{i-1} \} \).
Note that this definition does not allow recursively defined tokens.

In other words, \( d_i \) cannot be defined in terms of \( d_i \), not even indirectly.
We may now describe C++ identifiers as follows.

- **letter** → A | B | ··· | Z | a | b | ··· | z
- **digit** → 0 | 1 | ··· | 9
- **id** → letter(letter | digit)*
After writing a regular expression for each kind of token, we may combine them into one big regular expression describing all tokens.

\[
\begin{align*}
\text{id} & \rightarrow \text{letter(letter | digit)}^* \\
\text{num} & \rightarrow \text{digit(digit)}^* \\
\text{relop} & \rightarrow < | > | == | != | >= | <= \\
\text{token} & \rightarrow \text{id} | \text{num} | \text{relop} | \ldots
\end{align*}
\]
State Diagrams

- A regular expression may be represented by a state diagram.
- The state diagram provides a good guide to writing a lexical analyzer program.
Example (State Diagrams)

- **id**: letter | digit
- **num**: digit
- **token**: letter | digit
  - letter
  - digit
  - digit
Unfortunately, it is not that simple.
At what point may we stop in an accepting state?
Do not read “count” as 5 identifiers: “c”, “o”, “u”, “n”, “t”.
When we stop in an accepting state, we must be able to determine the type of token processed.
Did we read the id token “count” or did we read the if token “if”? 
Consider state diagrams to accept relational operators
\(==, \! =, <, >, \leq, \text{ and } \geq.\)
Example (State Diagrams)

- Combine them into a single state diagram.

Diagram:

- Node 1: relop
- Node 2: =
- Node 3: < | >
- Node 4: =

Arrows:
- 1 → 2
- 1 → 3
- 2 → 4
- 3 → 4
State Diagrams

- When we reach an accepting state, how can we tell which operator was processed?
- In general, we design the diagram so that each kind of token has its own accepting state.
State Diagrams

- If we reach state 3, how do we decide whether to continue to state 4?
- We read characters until the current character does not match any pattern, i.e., it would lead to the dead state.
- At that point, we accept the string, minus the last character.
- Later, processing resumes with the last character.
The Maximal Munch Principle

Process as many symbols as possible and still be able to match a regular expression.
Example

Example (State Diagrams)

- relop
  - = = other
  - ! = other
  - < = other
  - > = other
  - other

Other transitions not shown in the diagram.
Assignment

Homework

- Read Sections 3.1 - 3.4.