Lexical Analysis (Scanning)

Chapter 2
Lexical Analysis (Scanning)

• Basic Ideas
  - divide character stream into tokens
  - a token is the smallest logical unit in code
  - common categories:

  ```
  easy to enum.  
  keywords: “if”, “for”, “while”, ...
  special symbols: “+”, “-”, “=”, “[”, ...
  number: “4”, “23”, “6.63”, “001”, ...
  ID: “a”, “abs”, “sum”, ...
  ...
  ```

  hard to enum.
Scanning

- Token Categories
  - common categories: keywords, special symbols, number, and ID.
  - e.g.,

```c
int bigger(int a, int b)
{
    int c = 0;
    if (a > b)
        c = a;
    else
        c = b;
    return c;
}
```
Scanning

• Define Tokens
  - define different kinds of tokens in `enum`
Scanning

- Token Attributes
  - A token may carry attributes (e.g., `stringval`, `numberval`, ...)

```c
typedef enum
{
    IF,    // "if"
    ELSE,  // "else"
    PLUS,  // "+
    NUM,   // "23"
    ID,    // "a"
    ...
} TokenType;
```

```
if ... a > 5 ...
```

```
IF ... ID GREATER NUM ...
```

```
"a" "5" 5
```

Token Attributes
Scanning

• Token Attributes
  - A token may carry attributes (e.g., `stringval`, `numberval`, ...)

```c
typedef struct
{
    TokenType tokenval;
    char *stringval;
    int numval;
    ...
} TokenRecord;
```

```c
if ... a > 5 ...  # Token Attributes
```

```c
IF ... ID GREATER NUM ...
```

```
"a"  "5"  5
```
Scanning

- getToken()
  - scanner is often driven by the parser

Before

\[ a[i] = 4 + 2 \]

After

\[ a[i] = 4 + 2 \]

recognize and return \{ID, “a”\}
Regular Expressions
Regular Expressions

• Basics
  - A regex \( r \) represents a pattern of strings, where
  - the set of strings is called regular language \( L(r) \)
  - the character set is called alphabet \( \Sigma \)
  - Given an alphabet \( \Sigma \), we can construct regex \( r \):

\[
\begin{align*}
\text{if } r = a, a \text{ in } \Sigma & \quad \Rightarrow L(r) = \{a\} \\
\text{if } r = \phi & \quad \Rightarrow L(r) = \{ \} \quad \text{empty set} \\
\text{if } r = \varepsilon & \quad \Rightarrow L(r) = \{\varepsilon\} \quad \text{a set w/ an empty string}
\end{align*}
\]
Regular Expressions

- **Operations**
  - **alternation** “a|b”
  - **concatenation** “ab”
  - **repetition** “a*”

Given regex r and s,

\[ L(r|s) = L(r) \cup L(s) \]

Given regex r and s,

\[ L(rs) = L(r)L(s) \]

Given regex r,

\[ L(r^*) = \{\varepsilon\} \cup L(r) \cup L(rr) \cup L(rrr) \ldots \]
Regular Expressions

• Examples
  - What is the language of \((a|b)^*\) ?
    \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}

  - What is the language of \(a|b^*\) ?
    \{\varepsilon, a, b, bb, bbb, bbbbb, ...\}

Precedence: repetition > concatenation > alternation
Regular Expressions

• Names
  - As a notational simplification

\[(0|1|2|...|9) (0|1|2|...|9)^*\]

\[digit = 0|1|2|...|9\]
\[numseq = digit\ digit^*\]
Regular Expressions

• Extended Regex
  - one or more repetitions \( a^+ = aa^* \)
  - any character \( .b = (a | b | c)b \) if \( \Sigma = \{a, b, c\} \)
  - a range of characters
    \[
    [abc] \text{ or } [a-c] = (a | b | c) \\
    [acd] = (a | c | d)
    \]
  - not \( \neg (a | b) \) or \( ^ab \) = c if \( \Sigma = \{a, b, c\} \)
  - optional subexpressions
    \( (a | b)?c = ac | bc | c \)
Regular Expressions

• Exercise
  - What is the regex for US zip code?
    
    digit = [0-9]
    zip = digit{5}-digit{4}

  - What is the regex for any int between 2 and 36?
    
    digit = [0-9]
    zip = [2-9] | [12]digit | 3[0-6]
Regular Expression

• Exercise
  - Rewrite the regex with only three core operators (concatenation/alternation/repetition)

\[(x+y)? \cdot [^x-y]\]  \(\Sigma = \{x, y, z\}\)

\[xx^*y (x|y|z) z \mid (x|y|z)z\]

- Write the regex for strings in C programs (assume escape character \ is not allowed)
  
  E.g.  \(x = "hello, world!";\)

  "\[^""\\"]"
Token Specification
Token Specification

• Specify tokens with regex
  • Given the complexity, regex is perfect for this purpose

<table>
<thead>
<tr>
<th>easy to enum.</th>
<th>hard to enum.</th>
</tr>
</thead>
</table>
| **keywords:** “if”, “for”, “while”, ... | **special symbols:** “+”, “-”, “=”, “[”, ...
| **number:** “4”, “23”, “6.63”, “001”, ... | **ID:** “a”, “abs”, “sum”, ...
| ... | ... |
Token Specification

- **Numbers**
  - sequence of digits "23"
  - signed numbers "-12", "+17"
  - decimal numbers "1.24"
  - scientific numbers "2.74E+2"

\[
\begin{align*}
nat &= [0-9]+ \\
signedNat &= [+-]? \ nat \\
decimalNum &= signedNat \ . \ nat \\
scientificNum &= signedNat \ . \ nat \ E \ signedNat \\
number &= signedNat (\ . \ nat)? (E \ signedNat)?
\end{align*}
\]
Token Specification

• Reserved Words

\[
\text{reserved} = \text{if} \mid \text{while} \mid \text{do} \mid ... \\
\]

• Identifiers

- Begins with a letter; contains only letters and digits

\[
\begin{align*}
\text{letter} &= [a-zA-Z] \\
\text{digits} &= [0-9] \\
\text{identifier} &= \text{letter(letter|digit)}^* \\
\end{align*}
\]
Example -- Identifier

Valid Identifiers

• Composed of letters, digits, and underscores
• Cannot end in an underscore
• Cannot contain two underscores in a row

A1BC_3A_B5
Token Specification

- Comments
  - Typically are “skipped” during scanning
  - Still need to be recognized so they can be skipped

```
{this is a Pascal comment}  \{[^[\^\}]\}*\}

/* this is a C/C++ comment */

// this is a C/C++ comment  //^[\^\n]*
```
Comment  /*  ........  */

/*  this is the end  ****  not yet  **** not yet  *************/

/*  [^ *]*  (  ++  [^ /][^ *]*  )*  ++  / */
Token Specification

• Ambiguity
  - Token specification may contain ambiguities
  - Existing multiple ways to interpret the same substring

  "if" IF
  "if" identifier

  "<>" not equal
  "<"  ">"  less than, greater than

Keyword is preferred!  Longer token is preferred!

(principle of longest substring)
Token Specification

• Token Delimiters
  - Characters that imply a longer string cannot be a token
  - White spaces are delimiters
  - Comments could also be delimiters

"xtemp=ytemp"  "=" is not part of any token
"int x"  blank/newline/tab are neither
"do//if"  comments are neither

whitespace=(newline|blank|tab|comment)+
Token Specification

- Token Delimiters
  - A delimiter ends a token, but not part of that token
  - Should not be consumed, but just be examined - Lookahead
Finite Automata
Finite Automata

• Equivalence
  - a regex specifies a regular language
  - FA accepts a regular language
  - regex $\equiv$ FA

$$r = b(b|a)^*$$

$\Sigma = \{a, b\}$

- $b b a a b b$
- 1 2 2 2 2 2
- 2 accept!
Finite Automata

• Extensions and Simplification
  - name transitions w/ regex names (also, other and any)
  - error state

\[
\text{identifier} = \text{letter} (\text{letter} | \text{digit})^* \\
\text{letter} = [a-zA-Z] \\
\text{digit} = [0-9]
\]

“cs4all”
Finite Automata

• Extensions and Simplification
  - name transitions w/ regex names (also, *other* and *any*)
  - error state is often omitted

```plaintext
identifier = letter(letter|digit)*
letter = [a-zA-Z]
digit = [0-9]
```

“4all”

![Diagram of a finite automaton]

undefined transition: Error!
Finite Automata

• Exercise
  - FA for recognizing signed numbers

\[
\begin{align*}
digit &= [0-9] \\
nat &= digit^+ \\
signedNat &= (+|-)? nat
\end{align*}
\]
Finite Automata

Exercise

- FA for recognizing numbers

\( \text{digit} = [0-9] \)
\( \text{nat} = \text{digit}+ \)
\( \text{signedNat} = (+|-)? \text{nat} \)
\( \text{number} = \text{signedNat} ("." \text{nat})? (E \text{signedNat})? \)
Finite Automata

• Exercise
  - FA for recognizing comments

\{[^\^\}\]*\}

/* hello */
Finite Automata

• FA Actions
  - "normal" state: copy a character to a token buffer
  - accept state: return a token & go back to initial state
  - error state: generate an error

\[
\text{sum} = 4 + 2
\]

always output a token each time getting here?

always an error?

“longest matching principle”

“delimiter”
Finite Automata

- Adjusted FA
  - “error” state becomes an accept state
  - which has no further transition edges
  - [other] is from lookahead

---

s u m [ i ] = 4 + 2

---

```
start  
  ↓  
letter  
  ↓  
id  
  ↓  
[other]  
  ↓  
done  
```

- letter
- other
- digit
- any

a lookahead char

action: return ID
Finite Automata

• Recognize Multiple Types of Tokens
  - cannot track the states of all different FAs - too expensive!
  - solution: merge different FAs

![Diagram of Finite Automata]

- return GE
- return LE
- return EQ
Finite Automata

• Recognize Multiple Tokens
  - tokens starting with different characters
  - easier to merge: simply combine their starting states

![Diagram of finite automaton with transitions for >, <, and =, leading to different states labeled return GE, return LE, and return EQ.](image)
Finite Automata

- Recognize Multiple Tokens
  - tokens starting with the same character

\[< \quad = \quad \text{return LE} \]
\[< \quad > \quad \text{return NE} \]
\[< \quad \text{return LT} \]
Finite Automata

- **Recognize Multiple Tokens**
  - it becomes an **NFA** (*non-deterministic finite automaton*)
  - expensive to run an NFA!
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```

```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)

```
Finite Automata

• Recognize Multiple Tokens
  - it becomes an NFA (*non-deterministic finite automaton*)
  - NFA $\rightarrow$ DFA (deterministic ...)
  - DFA adjustment

![Diagram](image_url)

- return LE
- return NE
- return LT
Finite Automata

• Example:
  - Draw the FA for recognizing the following two kinds of tokens in a token string:

\[
\text{LT : <} \\
\text{LE : <=}
\]

\[
\begin{array}{c:c}
\text{<} & \text{<} \\
\text{=} & \text{=} \\
\text{<} & \text{<}
\end{array}
\]

\[
\begin{array}{c:c}
\text{1} & \text{<} \\
\text{2} & \text{=} \\
\text{3} & \text{[other]}
\end{array}
\]
Finite Automata

- Implementation
  - hard-coded

```
state = 1
while (!EOF)
{
    switch(state)
    case 1:
        if (advance() == '<')
            state = 2
        else
            error & break
    case 2:
        if (advance() == '=')
            state = 3
        else
            state = 4
    case 3:
        output token LE
        state = 1
    case 4:
        output token LT
        stepback()
        state = 1
}
```
Finite Automata

- Implementation
  - transition table

<table>
<thead>
<tr>
<th>state</th>
<th>&lt;</th>
<th>=</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>err</td>
<td>c = advance()</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>c = advance()</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>output LE</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>output LT; c = stepback()</td>
</tr>
<tr>
<td>err</td>
<td>-</td>
<td>-</td>
<td>print error; break</td>
</tr>
</tbody>
</table>

state = 1

while (!EOF)
{
    state = Trans[state][c]
    action(state)
}

}
Putting it All Together
Putting It All Together

• [Louden Ch. 2]

\[ LE = \leq \]
\[ LT = < \]

Regex2DFA Separately

Combine DFAs to NFA

NFA2DFA conversion

Adjust DFA

Diagram:

1. \( < \rightarrow 2 \)
2. \( = \rightarrow 3 \)
3. \( [\text{other}] \rightarrow 4 \)
Putting It All Together (flex)

- flex-generated scanner

- $LE = \leq$
- $LT = <$

Regex2DFA Separately

Combine DFAs to NFA

NFA2DFA conversion

Diagram:

1. <
2. =
3. 

States: 1, 2, 3
Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

```
< < = = <
```

```
1 2 err
```

```
latest accept output: LT
```

```
1 < 2
```

```
= 3
```

Diagram:
```
1 -> 2 <-
```
```
3 =
```

```
Putting It All Together (flex)

- **flex-generated scanner**
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

![Diagram of a scanner state machine]

```
<  <  =  =  <
```

```
1  2  3  err
```

Latest accept output: LE
Putting It All Together (flex)

- **flex-generated scanner**
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

no accept state! real error!