Role of the Lexical Analyzer

- Remove comments and white spaces (aka scanning)
- Macros expansion
- Read input characters from the source program
- Group them into lexemes
- Produce as output a sequence of tokens
- Interact with the symbol table
- Correlate error messages generated by the compiler with the source program
Scanner-Parser Interaction

source program → Lexical Analyzer

Lexical Analyzer → Parser

token

Parser → to semantic analysis

getNextToken

Symbol Table
Why Separating Lexical and Syntactic?

• Simplicity of design
• Improved compiler efficiency
  – allows us to use specialized technique for lexer, not suitable for parser
• Higher portability
  – Input-device-specific peculiarities restricted to lexer
Some Definitions

• **Token**: a pair consisting of
  – Token name: abstract symbol representing lexical unit [affects parsing decision]
  – Optional attribute value [influences translations after parsing]

• **Pattern**: a description of the form that different lexemes take

• **Lexeme**: sequence of characters in source program matching a pattern
### Token classes
- One token per keyword
- Tokens for the operators
- One token representing all identifiers
- Tokens representing constants (e.g. numbers)
- Tokens for punctuation symbols

#### Table

<table>
<thead>
<tr>
<th>Token</th>
<th>Informal Description</th>
<th>Sample Lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>characters i, f</td>
<td>if</td>
</tr>
<tr>
<td>else</td>
<td>characters e, l, s, e</td>
<td>else</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; or &gt; or &lt;= or &gt;= or == or !=</td>
<td>&lt;=, !=</td>
</tr>
<tr>
<td>id</td>
<td>letter followed by letters and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>any numeric constant</td>
<td>3.14159, 0, 6.02e23</td>
</tr>
</tbody>
</table>
| literal        | anything but ",", surrounded by ""'s                                    | "core dumped"
Example

\[ E = M \times C ** 2 \]

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>
Dealing With Errors

Lexical analyzer unable to proceed: no pattern matches

- Panic mode recovery: delete successive characters from remaining input until token found
- Insert missing character
- Delete a character
- Replace character by another
- Transpose two adjacent characters
Example

```cpp
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

What tokens will be generated from the above C++ program?
Buffering Issue

• Lexical analyzer may need to look at least a character ahead to make a token decision.
• Buffering: to reduce overhead required to process a single character
switch (*forward++) {
    case eof:
        if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
Tokens Specification

- We need a **formal** way to specify patterns: **regular expressions**
- **Alphabet**: any finite set of symbols
- **String** over alphabet: finite sequence of symbols drawn from that alphabet
- **Language**: countable set of strings over some fixed alphabet
1. A **prefix** of string $s$ is any string obtained by removing zero or more symbols from the end of $s$. For example, $ban$, $banana$, and $\epsilon$ are prefixes of $banana$.

2. A **suffix** of string $s$ is any string obtained by removing zero or more symbols from the beginning of $s$. For example, $nana$, $banana$, and $\epsilon$ are suffixes of $banana$.

3. A **substring** of $s$ is obtained by deleting any prefix and any suffix from $s$. For instance, $banana$, $nan$, and $\epsilon$ are substrings of $banana$.

4. The **proper** prefixes, suffixes, and substrings of a string $s$ are those, prefixes, suffixes, and substrings, respectively, of $s$ that are not $\epsilon$ or not equal to $s$ itself.

5. A **subsequence** of $s$ is any string formed by deleting zero or more not necessarily consecutive positions of $s$. For example, $baan$ is a subsequence of $banana$. 
## Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition and Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of $L$ and $M$</td>
<td>$L \cup M = { s \mid s$ is in $L$ or $s$ is in $M }$</td>
</tr>
<tr>
<td>Concatenation of $L$ and $M$</td>
<td>$LM = { st \mid s$ is in $L$ and $t$ is in $M }$</td>
</tr>
<tr>
<td>Kleene closure of $L$</td>
<td>$L^* = \bigcup_{i=0}^{\infty} L^i$</td>
</tr>
<tr>
<td>Positive closure of $L$</td>
<td>$L^+ = \bigcup_{i=1}^{\infty} L^i$</td>
</tr>
</tbody>
</table>

Zero or one instance: $r?$ is equivalent to $r|\varepsilon$

Character class: $a|b|c|\ldots|z$ can be replaced by $[a-z]$
$\text{a|c|d|h}$ can be replaced by $[\text{acdhd}]$
Operations

Trailing context: \( \text{exp1}/\text{exp2} \)

- Match \( \text{exp1} \) only if followed by \( \text{exp2} \)
- \( \text{exp2} \) is NOT consumed and remained to be returned in subsequent tokens
- Only one “/” is permitted per pattern
- Example: \( \text{a/b} \) matches \( \text{a} \) in string \( \text{ab} \) but will not match anything in \( \text{a} \) or \( \text{ac} \)
Examples

Which language is generated by:
• \((a|b)(a|b)\)
• \(a^*\)
• \((a|b)^*\)
• \(a|a^*b\)
Example

Presenting number that can be integer with option floating point and exponential parts?
Example

Presenting number that can be integer with option floating point and exponential parts?

Let’s analyze some possible solutions

\([-+]?[0-9.]*\) matches too much, like 1.2.3.4
\([-+]?[0-9]+\.)?[0-9]+\) matches too little, misses .12 or 12.
\([-+]?[0-9]+\.)?[0-9]*\) doesn't match 12.
\([-+]?[0-9]+\.)?[0-9]*\) doesn't match .12
\([-+]?[0-9]*\.)?[0-9]*\) matches nothing, or a dot with no digits at all
Examples

Write regular definition of all strings of lowercase letters in which the letters are in ascending order.
Tokens Recognition

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if } expr \text{ then } \text{stmt} \\
& \quad | \text{if } expr \text{ then } \text{stmt} \text{ else } \text{stmt} \\
& \quad | \epsilon \\
\text{expr} & \rightarrow \text{term} \text{ relop} \text{ term} \\
& \quad | \text{term} \\
\text{term} & \rightarrow \text{id} \\
& \quad | \text{number} \\
\text{digit} & \rightarrow [0-9] \\
\text{digits} & \rightarrow \text{digit}^+ \\
\text{number} & \rightarrow \text{digits} ( \ . \ \text{digits} )? \ ( \ E \ [+-]? \ \text{digits} )? \\
\text{letter} & \rightarrow [A-Za-z] \\
\text{id} & \rightarrow \text{letter} ( \ \text{letter} \ | \ \text{digit} )^* \\
\text{if} & \rightarrow \text{if} \\
\text{then} & \rightarrow \text{then} \\
\text{else} & \rightarrow \text{else} \\
\text{relop} & \rightarrow \text{<} \ | \ \text{>} \ | \ \text{<=} \ | \ \text{>=} \ | \ = \ | \ <> \\
\text{ws} & \rightarrow ( \ \text{blank} \ | \ \text{tab} \ | \ \text{newline} )^+ \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>LEXEMES</th>
<th>TOKEN NAME</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any ws</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>–</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>–</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>–</td>
</tr>
<tr>
<td>Any id</td>
<td>id</td>
<td>Pointer to table entry</td>
</tr>
<tr>
<td>Any number</td>
<td>number</td>
<td>Pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>relop</td>
<td>LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relop</td>
<td>LE</td>
</tr>
<tr>
<td>=</td>
<td>relop</td>
<td>EQ</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>relop</td>
<td>NE</td>
</tr>
<tr>
<td>&gt;</td>
<td>relop</td>
<td>GT</td>
</tr>
<tr>
<td>&gt;=</td>
<td>relop</td>
<td>GE</td>
</tr>
</tbody>
</table>
Implementation:
Transition Diagrams

• Intermediate step in constructing lexical analyzer

• Convert patterns into flowcharts called transition diagrams
  – nodes or circles: called states
  – Edges: directed from state to another, labeled by symbols
Means retract the forward pointer
letter or digit

start

 lettre

other

* return(getToken(), installID())
• Examine symbol table for the lexeme found
• Returns whatever token name is there

• Places ID in symbol table if not there.
• Returns a pointer to symbol table entry
Reserved Words and Identifiers

• Install reserved words in symbol table initially

  OR

• Create transition diagram for each keyword, then prioritize the tokens so that keywords have higher preference
Implementation of Transition Diagram

TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
               or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                if ( c == '<' ) state = 1;
                else if ( c == '=' ) state = 5;
                else if ( c == '>' ) state = 6;
                else fail(); /* lexeme is not a
                                break;
            case 1: ...
                ...
            case 8: retract();
                retToken.attribute = GT;
                return(retToken);
        }
    }
}
Using All Transition Diagrams: The Big Picture

• Arrange for the transition diagrams for each token to be tried sequentially
• Run transition diagrams in parallel
• Combine all transition diagrams into one
The First Part of the Project

Lex source program

\[ \text{lex.l} \rightarrow \text{Lex compiler} \rightarrow \text{lex.yy.c} \]

\[ \text{lex.yy.c} \rightarrow \text{C compiler} \rightarrow \text{a.out} \]

Input stream

\[ \text{a.out} \rightarrow \text{Sequence of tokens} \]
The First Part of the Project

- Lex source program: `lex.l`
- Translation rules: `lex.yy.c`
- Auxiliary functions: `a.out`

Diagram:
- Lex source program `lex.l` → Lex compiler → `lex.yy.c`
- `lex.yy.c` → C compiler → `a.out`
- `a.out` → Sequence of tokens
- Input stream → `a.out`
declarations

/* definitions of manifest constants
   LT, LE, EQ, NE, GT, GE,
   IF, THEN, ELSE, ID, NUMBER, RELOP */
%

/\ regular definitions */
delim [ \t\n]  
ws {delim}+ 
letter [A-Za-z]  
digit [0-9]  
id {letter}{(letter)\{digit\})*  
number {digit}+(\.\{digit\}+)?(E[+-]?\{digit\}+)?
%

{ws} {/* no action and no return */}
if {return(IF);}
then {return(THEN);}
else {return(ELSE);}
{id} {yyval = (int) installID(); return(ID);}
{number} {yyval = (int) installNum(); return(NUMBER);}
"<" {yyval = LT; return(RELOP);}
"<=" {yyval = LE; return(RELOP);}
"==" {yyval = EQ; return(RELOP);}
"<>" {yyval = NE; return(RELOP);}
">" {yyval = GT; return(RELOP);}
">=" {yyval = GE; return(RELOP);}
%

int installID() { /* function to install the lexeme, whose
   first character is pointed to by yytext,
   and whose length is yylen, into the
   symbol table and return a pointer
   there to */
}

int installNum() { /* similar to installID, but puts numerical
   constants into a separate table */
}
Anything between these 2 marks is copied as it is in lex.yy.c

/* definitions of manifest constants */
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
%

/* regular definitions */
delim [ \t\n ]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}{(letter|digit)*
number {digit}+/(\.digit+)?(E[+-]?{digit}+)?
%

{ws} {/* no action and no return */}
if {/* return(IF);*/
then {return(THEN);}
else {return(ELSE);}
id {yyval = (int) installID(); return(ID);}
number {yyval = (int) installNum(); return(NUMBER);}
"<" {yyval = LT; return(RELOP);}
"<=" {yyval = LE; return(RELOP);}
"==" {yyval = EQ; return(RELOP);}
"<>" {yyval = NE; return(RELOP);}
">" {yyval = GT; return(RELOP);}
">=" {yyval = GE; return(RELOP);}
%

int installID() { /* function to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer there to */
}

int installNum() { /* similar to installID, but puts numerical constants into a separate table */
}
Lecture of Today

- Sections 3.1 to 3.5
- First part of the project assigned