Lexical Analysis
Part I
Chapter 3

Slides adapted from:
© Robert van Engelen, Florida State University

Input Stream

• Example:

  if (i == j)
  z = 0;
  else
  z = 1;

• The input is just a string of characters:

  	if (i == j)\n  \tz = 0;\ntelse\n  \tz = 1;
Preliminary Operations

- Remove comments by matching special markers
- Remove white spaces
- Trace line numbers (for error messages)

Interaction of the Lexical Analyzer with the Parser

Source Program → Lexical Analyzer → Symbol Table → Parser → ...
The Reason Why Lexical Analysis is a Separate Phase

• Simplifies the design of the compiler
  – LL(1) or LR(1) parsing with 1 token lookahead would not be difficult
• Provides efficient implementation
  – Systematic techniques to implement lexical analyzers by hand or automatically from specifications
• Portability
  – encapsulate language peculiarities into scanner

Bad Language Choices

• Whitespace is insignificant in Fortran

  – **DO 5   I = 1,25**
    “run code between here and line 5 a total of 25 times, setting ‘I’ to values from 1 to 25”

  – **DO 5   I = 1.25**
    “set the variable named ‘DO5I’ to have the value 1.25”
Attributes of Tokens

\[ y = 31 + 28x \]

Lexer

Tokens, Patterns, and Lexemes

- A token is a pair consisting of a token name (a classification label) and an optional attribute
  - Example: \( \langle \text{id}, "x" \rangle, \langle \text{num}, 28 \rangle \)
- Lexemes are the specific character strings that make up a token
  - Example: abc and 123
- Patterns are rules describing the set of lexemes belonging to a token
  - Example: “letter followed by letters and digits” and “non-empty sequence of digits”
Most Common Classes

- One token for each keyword
- Tokens for individual operators or classes of operators (e.g. comparison)
- One token for all identifiers (id)
- One or more tokens for constants (number, literal, …)
- Tokens for punctuation symbols

Examples

<table>
<thead>
<tr>
<th>Token</th>
<th>Pattern</th>
<th>Lexeme</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;, &gt;, &lt;=, ==, ...</td>
<td>&lt;, &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>3.14, 0, ...</td>
</tr>
<tr>
<td>literal</td>
<td>Anything but “” surrounded by “”</td>
<td>“core”</td>
</tr>
</tbody>
</table>
Exercise

• For the code fragment

\[
x = 0; \text{\texttt{\textbackslash{}n while}} (x < 10) \{ \text{\textbackslash{}n} x++; \text{\textbackslash{}n}\}
\]

count the number of tokens in each class: whitespace, keyword, identifier, number, other ({} () < ++ ; =)

Lexical Errors

• Hard to detect errors without aid of other components:

\[
\text{\texttt{\textbackslash{}fi}} (x == 10) \ldots
\]

• When scanner is unable to proceed, use error-recovery actions
  – Delete successive characters (panic mode)
  – Insert missing character
  – Replace character
  – Switch two characters
Input Buffering

- Involves two buffers that are alternatively reloaded
  - Pointer \texttt{lexemeBegin} marks the beginning of the current lexeme
  - Pointer forward scans ahead until a pattern match is found, and is retracted afterward

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{input_buffering_diagram}
\caption{Input Buffering Diagram}
\end{figure}

Specification of Patterns for Tokens: \textit{Language Definitions}

- An \textit{alphabet} $\Sigma$ is a finite set of symbols (characters)
- A \textit{string} $s$ is a finite sequence of symbols from $\Sigma$
  - $|s|$ denotes the length of string $s$
  - $\varepsilon$ denotes the empty string, thus $|\varepsilon| = 0$
- A \textit{language} is a specific set of strings over some fixed alphabet $\Sigma$
Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings $x$ and $y$ is denoted by $xy$
- The *exponentiation* of a string $s$ is defined by
  
  \[
  s^0 = \varepsilon \\
  s^i = s^{i-1}s \quad \text{for } i > 0
  \]

  note that $s\varepsilon = \varepsilon s = s$

Specification of Patterns for Tokens: *Language Operations*

- **Union**
  \[
  L \cup M = \{s \mid s \in L \text{ or } s \in M\}
  \]
- **Concatenation**
  \[
  LM = \{xy \mid x \in L, y \in M\}
  \]
- **Exponentiation**
  \[
  L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L, \ i > 0
  \]
- **Kleene closure**
  \[
  L^* = \bigcup_{i \geq 0} L^i
  \]
- **Positive closure**
  \[
  L^+ = \bigcup_{i \geq 1} L^i
  \]
Specification of Patterns for Tokens: *Regular Expressions*

- **Basis symbols:**
  - \( \varepsilon \) is a regular expression denoting language \{\varepsilon\}
  - \( a \in \Sigma \) is a regular expression denoting \{a\}
- If \( r \) and \( s \) are regular expressions denoting languages \( L(r) \) and \( M(s) \) respectively, then
  - \( r \mid s \) is a regular expression denoting \( L(r) \cup M(s) \)
  - \( rs \) is a regular expression denoting \( L(r)M(s) \)
  - \( r^* \) is a regular expression denoting \( L(r)^* \)
  - \( \overline{r} \) is a regular expression denoting \( \overline{L(r)} \)
- A language defined by a regular expression is called a *regular set*

---

**Example:** \( r = ab \mid c, \ L(r) = ?? \)

\[
\begin{align*}
a & \quad L(a) = \{a\} \\
b & \quad L(b) = \{b\} \\
ab & \quad L(ab) = L(a)L(b) = \{ab\} \\
c & \quad L(c) = \{c\} \\
ab \mid c & \quad L(ab \mid c) = L(ab) \cup L(c) \\
& \quad = \{ab\} \cup \{c\} = \{ab, c\}
\end{align*}
\]
Specification of Patterns for Tokens: *Regular Expressions*

- Example: \( r = a(b \mid c), \ L(r) = \ ?? \)

\[
\begin{align*}
  b & \quad L(b) = \{b\} \\
  c & \quad L(c) = \{c\} \\
  b \mid c & \quad L(b \mid c) = L(b) \cup L(c) = \{b, c\} \\
  (b \mid c) & \quad L((b \mid c)) = L(b \mid c) = \{b, c\} \\
  a & \quad L(a) = \{a\} \\
  a(b \mid c) & \quad L(a(b \mid c)) = L(a) L((b \mid c)) \\
                 & = \{a\} \ \{b, c\} = \{ab, ac\}
\end{align*}
\]

Specification of Patterns for Tokens: *Regular Expressions*

- \( ab, \) 
- \( a\varepsilon b, \) 
- \( \varepsilon a\varepsilon b, \) 
- \( \varepsilon a\varepsilon b, \) 
- \( \varepsilon a\varepsilon b \)

All the above regular expressions are equivalent, i.e., they generate the same language.
Specification of Patterns for Tokens: Regular Definitions

• Regular definitions introduce a naming convention:
  \[ d_1 \rightarrow r_1 \]
  \[ d_2 \rightarrow r_2 \]
  ...
  \[ d_n \rightarrow r_n \]
  where each \( r_i \) is a regular expression over \( \Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\} \)

• Any \( d_j \) in \( r_i \) can be textually substituted in \( r_i \) to obtain an equivalent set of definitions

Specification of Patterns for Tokens: Regular Definitions

• Example:
  \[
  \text{letter} \rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
  \text{digit} \rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
  \text{id} \rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^* 
  \]

• Regular definitions are not recursive:
  \[
  \text{digits} \rightarrow \text{digit} \text{digits} \mid \text{digit} \quad \text{wrong!}
  \]
Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:
  \[ r^+ = rr^* \]
  \[ r? = r | \epsilon \]
  \[ [r_1, r_2, \ldots, r_q] = r_1 | r_2 | \ldots | r_q \]
  \[ [a-z] = a | b | c | \ldots | z \]

- Examples:
  ```
  digit \rightarrow [0-9]
  num \rightarrow (+ | -)? digit^+ (. digit^+)? ( E (+ | -)? digit^+ )?
  ```

---

Example: Toy Grammar

Grammar

\[
stmt \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \epsilon
\]

\[
expr \rightarrow \text{term relop term} \\
| \text{term}
\]

\[
term \rightarrow \text{id} \\
| \text{number}
\]
Example: Recognition of Tokens

Regular definitions

digit  \rightarrow [0-9]
digits  \rightarrow \text{digit}^+
number  \rightarrow \text{digits} (\cdot \text{digits})? (E \ [+\ -]? \text{digits})?
letter  \rightarrow [A-Za-z]
   \text{id}  \rightarrow \text{letter} (\text{letter} | \text{digit})^*

Example: Recognition of Tokens

Regular definitions (cont’d)

if  \rightarrow \text{if}
then  \rightarrow \text{then}
else  \rightarrow \text{else}
relop  \rightarrow < | <= | <> | > | >= | =
blank  \rightarrow \ ' '
       \rightarrow \text{tab}
newline  \rightarrow \text{\n}
ws  \rightarrow (\text{blank} | \text{tab} | \text{newline})^+
Exercise

• Choose the regular languages that are equivalent to \((0 \mid 1)^*1(0 \mid 1)^*\)

  \begin{align*}
  &- (01 \mid 11)^*(0 \mid 1)^* \\
  &- (0 \mid 1)^*(10 \mid 11 \mid 1)(0 \mid 1)^* \\
  &- (1 \mid 0)^*1(1 \mid 0)^* \\
  &- (0 \mid 1)^*(0 \mid 1)(0 \mid 1)^*
  \end{align*}

Exercise

• Twelve-hour times of the form “04:13PM”. Minutes should always be a two digit number, but hours \textit{may} be a single digit.
Exercise (cont’d)

• Correct or wrong?
  – (0 + 1)?[0-9]:[0-5][0-9](AM + PM)
  – ((0 + ε)[0-9] + 1[0-2]):[0-5][0-9](AM + PM)
  – (0*[0-9] + 1[0-2]):[0-5][0-9](AM + PM)
  – (0?[0-9] + 1(0 + 1 + 2):[0-5][0-9](A + P)M

Coding Regular Definitions in Transition Diagrams

• Transition Diagrams represent regular expressions by means of:
  – States: summarizing all we need to know about characters in between *lexemeBegin* pointer and *forward* pointer
  – Edges: labeled transitions from state to state
  – *Initial* state and set of *final* states
  – Operator * to retract *forward* pointer
Coding Regular Definitions in Transition Diagrams

\[
\text{relop} \rightarrow < | \leq | \leq | > | \geq | =
\]

[Diagram showing transition from start to states with transitions for <, ≤, ≤, ≥, >, =, and actions returning relop values like LE, NE, LT, EQ, GE, GT]

Recognition of Reserved Words

- **id** recognizes also language reserved words (if, else, …)
- Install all reserved words in symbol table initially

\[
id \rightarrow \text{letter} \ (\text{letter} \ | \text{digit})^*
\]

[Diagram showing transitions from start to states for letter and other with action returning gettoken(), install_id()]
Recognition of Reserved Words

• Alternative solution: Create transition diagrams for each reserved word
• Must prioritize tokens

Coding Transition Diagrams

```
TOKEN getRelOp() {
    TOKEN retToken = new(RELOP)
    // state a global variable
    while (1) {
        switch (state) {
            case 0:
                c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail();
                break;
            case 1:
                ...
            case 8:
                retract();
                retToken.attribute = GT;
                return(retToken);
            ...
        }
    }
}
```
Coding Transition Diagrams

- Method `fail()` resets `forward` pointer to `lexemeBegin`
- Change value of `state` to another transition diagram for another token
- If no other transition diagram, initiate error-correction phase

Exercise (difficult)

- Comments wrapped by `/* */`
  Allow strings wrapped by pairs of "

`letter` → `[A–Za–z]`
`digit` → `[0–9]`
...
`string` → "(letter | digit | ...)"
...

The Lex and Flex Scanner Generators

• *Lex* and its newer cousin *flex* are scanner generators
• Systematically translate regular definitions into C source code for efficient scanning
• Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex and Flex

lex source program \texttt{lex.l} \rightarrow \texttt{lex.yy.c} \rightarrow \texttt{lex.yy.c} \rightarrow \texttt{lex.yy.c}

\texttt{lex.yy.c} \rightarrow \texttt{C} compiler \rightarrow \texttt{a.out} \rightarrow \texttt{a.out}

\texttt{a.out} \rightarrow \texttt{sequence of tokens}

\texttt{a.out} \rightarrow \texttt{input stream} \rightarrow \texttt{sequence of tokens}
Lex Specification

• A lex specification consists of three sections separated by “%%”:

  regular definitions
  C declarations enclose by %{ %}
  %
  translation rules (see next slide)
  %
  user-defined auxiliary C procedures

Lex Specification

• The translation rules (second section) are of the form:

  \[
  \begin{align*}
  p_1 & \{ action_1 \} \\
  p_2 & \{ action_2 \} \\
  \cdots & \\
  p_n & \{ action_n \}
  \end{align*}
  \]

  where \( p_i \) is a regular expression and might use regular definitions from first section
Example

No declaration
no regular
definition

No auxiliary
Procedure
(optional)

No translation
rule

flex copy.1
gcc lex.yy.c -o primo -lfl
./primo < prova.txt > out.txt

Example

No declaration
no regular
definition

No auxiliary
procedure

Translation
rule but
no action

flex deletethis.1
gcc -o scan lex.yy.c -lfl
./scan
This deletethis is not deletethis useful.
This is not useful.
^D
Example

```c
#include <stdio.h>

main()
{
    yylex();
}

[0-9]+ { printf("%s\n", yytext); } .|\n    { }

flex -o replacer.yy.c replacer.l
gcc -o replacer replacer.yy.c -lfl
./replacer
This replaced is not very replaced useful.
Please dontreplacedatall.
```

Translation rules

Contains the matching lexeme

Invokes the lexical analyzer

```c
{ include <stdio.h> 
   
[0-9]+ { printf("%s\n", yytext); } .|\n   { }

main()
{ yylex(); }

lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```
Regular Expressions in Lex

\texttt{x} \quad \text{match the character } x

. \quad \text{match any character except newline}

\textbackslash . \quad \text{match the character } .

"string" \quad \text{match contents of string of characters}

^ \quad \text{match beginning of a line}

$ \quad \text{match the end of a line}

[xyz] \quad \text{match one character } x, y, \text{ or } z

\text{ (use } \textbackslash \text{ to escape } -, \textbackslash, ^) [^xyz] \quad \text{match any character except } x, y, \text{ and } z

[a-z] \quad \text{match one of } a \text{ to } z

[a-z]\{-\}[aeiou] \quad \text{match a lower case consonant}
Regular Expressions in Lex

- $r^*$ closure (match zero or more occurrences)
- $r^+$ positive closure (match one or more occurrences)
- $r?$ optional (match zero or one occurrence)
- $r\{3\}$ match three occurrences
- $r\{3,7\}$ match three to seven occurrences
- $r\{3,\}$ match three or more occurrences

- $^r$ match only if at the start of the line
- $r$ match only if at the end of the line
- $r_1r_2$ match $r_1$ then $r_2$ (concatenation)
- $r_1|r_2$ match $r_1$ or $r_2$ (union)
- $(r)$ grouping
- $\{d\}$ match the regular expression defined by $d$
- $r_1/r_2$ match $r_1$ when followed by $r_2$
Example

```c
{%
#include <stdio.h>
int chars = 0; int words = 0; int lines = 0;
%
word [^ \t\n\r\f\v]+
%
{word} { words++; chars += strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }
%
main()
{
    yylex();
    printf("lines: %8d\nwords: %8d\ncharacters: %8d\n", lines, words, chars); }
%
%
Translation rules

Translation rules

Regular definition

Regular definition

Example

```
### Functions and global variables

- **yylex()**: scanning routine
- **yytext**: string with matching lexeme
- **yyin**: input file (default: stdin)
- **yyout**: output file (default: stdout)
- **yyleng**: length of *yytext*
- **unput(c)**: places *c* back in *yyin*
- **input()**: reads next char
- **ECHO**: the default echo function

### Start conditions

- Flex allows conditional activation for rules
- START conditions placed in first part
  - `%s` inclusive conditions
  - `%x` exclusive conditions
- Use BEGIN to activate START condition
Start conditions

\[ \langle s \rangle r \] match \( r \) only if in start condition \( s \)

\[ \langle s_1, s_2 \rangle r \] match \( r \) only if in any of the start condition \( s_1, s_2 \)

\[ \langle \ast \rangle r \] match \( r \) only if in any start condition

Example

```plaintext
%s AA BB CC
%^a \{ECHO; BEGIN AA;\}
%^b \{ECHO; BEGIN BB;\}
%^c \{ECHO; BEGIN CC;\}
\n\{ECHO; BEGIN 0;\}
<AA>magic \{printf("first");\}
<BB>magic \{printf("second"); \}
<CC>magic \{printf("third");\}
. ;
```

Inclusive start conditions
Start condition activation
Start condition reset
Conditionally activated rules
Example

/* scanner for a toy Pascal-like language */

{%
/* need this for the call to atof() below */
#include <math.h>
%

DIGIT [0-9]
ID [a-z][a-z0-9]*

Example

%%

(DIGIT)+
{ printf("An integer: %s (%d)\n", yytext, atoi( yytext ) );
}

(DIGIT)+".(DIGIT)*
{ printf("A float: %s (%g)\n", yytext, atof( yytext ) );
}

if|then|begin|end|procedure|function
{ printf("A keyword: %s\n", yytext );
}
Example

```c
{ID}  printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"  {  
  printf( "An operator: %s\n", yytext ); }  

"{[^}\n]*"
{
  /* eat up one-line comments */

[ \t\n]+  /* eat up whitespace */
.
  printf("Unrecognized character:
    %s\n", yytext );

```

Example

```c
%%

int main( int argc, char **argv )
{
  ++argv, --argc;  /* skip over program name */
  if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
  else
    yyin = stdin;
  yylex();
}
```
Example

{% /* definitions of manifest constants */
#define LT (256)
...
%
}
delim [ \t\n]
ws {delim}+
text [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.\{digit\}+)?(E[+\-]{digit}+)?

Example

%%
{ws} { }
if {return IF;}
then {return THEN;}
else {return ELSE;}
{id} {yytval = install_id(); return ID;}
{number} {yytval = install_num(\; return NUMBER;}
"<" {yytval = LT; return RELOP;}
"<=" {yytval = LE; return RELOP;}
"==" {yytval = EQ; return RELOP;}
"<>" {yytval = NE; return RELOP;}
">" {yytval = GT; return RELOP;}
">=" {yytval = GE; return RELOP;}
%
int install_id()
{...}

Return token to parser
Token attribute
Install \texttt{yytext} as identifier in symbol table