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:

  ```
  \tif (i == j)\n  \tz = 0;\n  \telse\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

- Get next token
- Token, tokenval
- error
- error
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 possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input

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”
Input Buffering

• Involves two buffers that are alternatively reloaded
  – Pointer lexemeBegin marks the beginning of the current lexeme
  – Pointer forward scans ahead until a pattern match is found

Attributes of Tokens

\[ y = 31 + 28 \times x \]
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}, \text{"x"} \rangle, \langle \text{num}, 28 \rangle
- Lexemes are the specific character strings that make up a token
  - Example: \text{abc} and \text{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”

Exercise

- For the code fragment

\[
x = 0;\text{\newblock}while (x < 10) \{\text{\newblock}tx++;\text{\newblock}\}
\]

count the number of tokens in each class: Whitespace, Keyword, Identifier, Number, Other (\{ \} ( ) < ++ ; =)
Specification of Patterns for Tokens: *Language Definitions*

- An *alphabet* $\Sigma$ is a finite set of symbols (characters)
- A *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 *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 *exponentation* of a string $s$ is defined by

  $s^0 = \varepsilon$
  $s^i = s^{i-1}s$ 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**
  \[ L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L, \quad i > 0 \]

- **Kleene closure**
  \[ L^* = \bigcup_{i=0}^{\infty} L^i \]

- **Positive closure**
  \[ L^+ = \bigcup_{i=1}^{\infty} L^i \]

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 \( L(r)^* \)
  - \( (r) \) is a regular expression denoting \( L(r) \)

- **A language defined by a regular expression is called a regular set**
Specification of Patterns for Tokens: Regular Expressions

- 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*

\[ ab, \]
\[ a\epsilon b, \]
\[ \epsilon a\epsilon b, \]
\[ \epsilon a\epsilon b, \]
\[ \epsilon a\epsilon \epsilon b \]

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:

  \[
  \begin{align*}
  \text{letter} & \rightarrow \text{A} | \text{B} | \ldots | \text{Z} | \text{a} | \text{b} | \ldots | \text{z} \\
  \text{digit} & \rightarrow 0 | 1 | \ldots | 9 \\
  \text{id} & \rightarrow \text{letter} ( \text{letter} | \text{digit} )^* 
  \end{align*}
  \]

- Regular definitions are not recursive:

  \[
  \text{digits} \rightarrow \text{digit} \text{digits} | \text{digit} \quad \text{wrong!}
  \]

---

Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

  \[
  \begin{align*}
  r^+ &= rr^* \\
  r? &= r | \varepsilon \\
  [r_1, r_2, \ldots, r_q] &= r_1 | r_2 | \ldots | r_q \\
  [a-z] &= a | b | c | \ldots | z 
  \end{align*}
  \]

- Examples:

  \[
  \begin{align*}
  \text{digit} & \rightarrow [0-9] \\
  \text{num} & \rightarrow \text{digit}^* ( , \text{digit}^* )? ( \text{E} ( + | - )? \text{digit}^* )?
  \end{align*}
  \]
Toy Grammar

Grammar

\[ stmt \rightarrow if \ expr \ then \ stmt \]
\[ \mid if \ expr \ then \ stmt \ else \ stmt \]
\[ \mid \epsilon \]
\[ expr \rightarrow term \ relop \ term \]
\[ \mid term \]
\[ term \rightarrow id \]
\[ \mid number \]

Recognition of Tokens

Regular definitions

\[ digit \rightarrow [0-9] \]
\[ digits \rightarrow digit^+ \]
\[ number \rightarrow digits (. digits)? (E [+-]? digits )? \]
\[ letter \rightarrow [A-Za-z] \]
\[ id \rightarrow letter ( letter \mid digit )^* \]
Recognition of Tokens

Regular definitions (cont’d)

\[
\begin{align*}
\text{if} & \rightarrow \text{if} \\
\text{then} & \rightarrow \text{then} \\
\text{else} & \rightarrow \text{else} \\
\text{relop} & \rightarrow < \mid \leq \mid > \mid \geq \mid = \\
\text{blank} & \rightarrow \ ' \ ' \\
\text{tab} & \rightarrow \text{\textbackslash t} \\
\text{newline} & \rightarrow \text{\textbackslash n} \\
\text{ws} & \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+
\end{align*}
\]

Exercise

- Choose the regular languages that are equivalent to \((0 \mid 1)^*1(0 \mid 1)^*\)
  
  - \((01 \mid 11)^*(0 \mid 1)^*\)  
  - \((0 \mid 1)^*(10 \mid 11 \mid 1)(0 \mid 1)^*\)  
  - \((1 \mid 0)^*1(1 \mid 0)^*\)  
  - \((0 \mid 1)^*(0 \mid 1)(0 \mid 1)^*\)
Exercise

- Twelve-hour times of the form “04:13PM”. Minutes should always be a two digit number, but hours may be a single digit.

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

relop → < | <= | <> | > | >= |

\begin{center}
\begin{tikzpicture}
  \node (start) {start};
  \node (1) [right of=start] {1};
  \node (2) [below right of=1] {2};
  \node (3) [below left of=1] {3};
  \node (4) [above right of=2] {4};
  \node (5) [below right of=4] {5};
  \node (6) [below left of=5] {6};
  \node (7) [below right of=6] {7};
  \node (8) [below left of=7] {8};

  \path
    (start) edge [above] node {<} (1)
    (1) edge [above] node {=} (2)
    (1) edge [below] node {other} (3)
    (2) edge [above] node {return(relop, LE)} (2)
    (3) edge [above] node {return(relop, NE)} (3)
    (3) edge [below] node {return(relop, LT)} (3)
    (4) edge [above] node {=} (5)
    (5) edge [above] node {return(relop, EQ)} (5)
    (5) edge [below] node {other} (6)
    (6) edge [above] node {return(relop, GE)} (6)
    (6) edge [below] node {other} (7)
    (7) edge [above] node {return(relop, GT)} (7)
    (8) edge [above] node {return(relop, LT)} (8)
    (8) edge [below] node {return(relop, LT)} (8);
\end{tikzpicture}
\end{center}

Recognition of Reserved Words

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

\begin{center}
\begin{tikzpicture}
  \node (start) {start};
  \node (9) [right of=start] {9};
  \node (10) [below right of=9] {10};
  \node (11) [below left of=10] {11};

  \path
    (start) edge [above] node {letter} (9)
    (9) edge [above] node {letter or digit} (10)
    (10) edge [above] node {return(gettoken(), install_id)} (11)
    (11) edge [below] node {return(gettoken(), install_id)} (11);
\end{tikzpicture}
\end{center}
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

Input stream → *lex* or *flex* compiler → *lex.yy.c* → *C* compiler → *a.out* → 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:

\[
\begin{align*}
p_1 & \{ \text{action}_1 \} \\
p_2 & \{ \text{action}_2 \} \\
\vdots \\
p_n & \{ \text{action}_n \}
\end{align*}
\]

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

No declaration
no regular
definition

%%
replaced
printf("replaced");
%%

No auxiliary
procedure

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

Translation
rule with
both pattern
and action

Example

Translation
rules

Contains
the matching
lexeme

Contains
the lexical
analyzer

lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l

%{  
#include <stdio.h>
%
%
[0-9]+ { printf("%s
", yytext); }  
}|n { }
%
main()
{ yylex(); }  
}
Regular Expressions in Lex

- `x` match the character `x`
- `.` match any character except newline
- `\.` match the character .
- "string" match contents of string of characters
- `^` match beginning of a line
- `$` match the end of a line

- `[xyz]` match one character `x`, `y`, or `z`
- `[^xyz]` match any character except `x`, `y`, and `z`
- `[a-z]` match one of `a` to `z`
- `[a-z]{-}[aeiou]` 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

Regular Expressions in Lex

$^r$ 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

Regular definition

```c
digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*

{digit}+ { printf("number: %s\n", yytext); }
{id} { printf("ident: %s\n", yytext); }
. { printf("other: %s\n", yytext); }

main()
{
 yylex();
}
```
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 \quad \text{match } r \text{ only if in start condition } s

\langle s_1, s_2 \rangle r \quad \text{match } r \text{ only if in any of the start condition } s_1, s_2

\langle * \rangle r \quad \text{match } r \text{ 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
/* 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

/* 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}+    
  {yytext, atoi( yytext )};
}

{DIGIT}+"."{DIGIT}*  
  {yytext, atof( yytext )};
}

if|then|begin|end|procedure|function   
  {yytext };
}
Example

[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

%%

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}+
letter [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}  {yyval = install_id(); return ID;}
{number} {yyval = install_num(); 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 install_id()
{...}

Return token to parser
Token attribute

Install \texttt{yytext} as identifier in symbol table