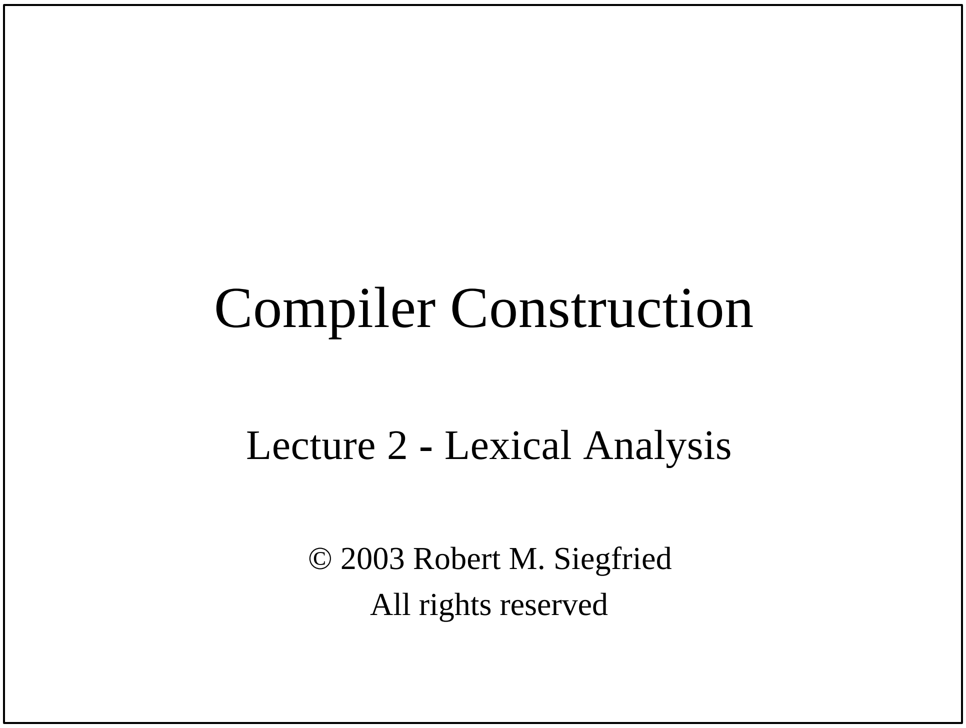
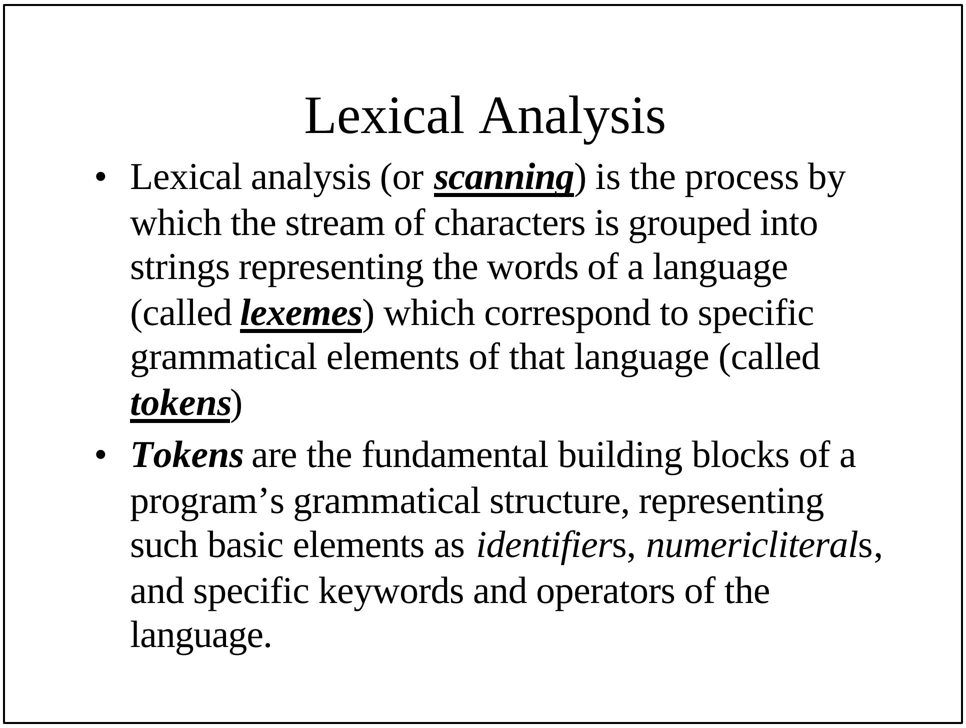


# Compiler Construction

## Lecture 2 - Lexical Analysis



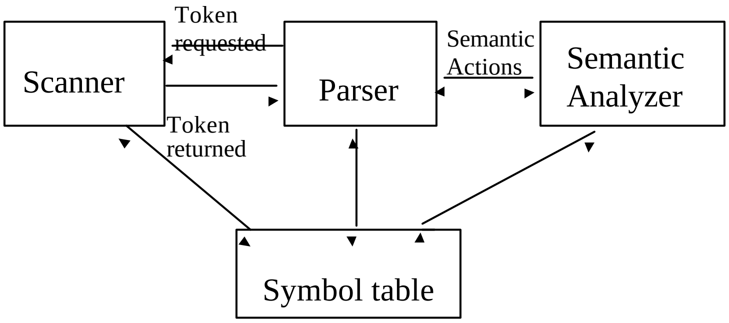
## Lexical Analysis

- Lexical analysis (or ***scanning***) is the process by which the stream of characters is grouped into strings representing the words of a language (called ***lexemes***) which correspond to specific grammatical elements of that language (called ***tokens***)
- ***Tokens*** are the fundamental building blocks of a program's grammatical structure, representing such basic elements as *identifier*s, *numericliteral*s, and specific keywords and operators of the language.

## Lexical Analysis (continued)

- Lexemes are the character strings assembled from the character stream of a program, and the token represents what component of the program's grammar they constitute.

## The scanner's role in the compiler's front end


Scanner

Token requested

Token returned

Parser

Semantic Actions

Semantic Analyzer

Symbol table

# The scanner’s role in the compiler’s front end (continued)

- The parser is the driving force for much of the compiler’s front end.
- The parser requests a token from the scanner, which returns the token corresponding to the next lexeme.
- The parser requests a particular semantic action, which depends on what component of the grammar is being parsed.
- All parts of the front end add information into the symbol table; the scanner adds lexemes to the symbol table (when necessary) and the symbol table returns to the scanner the token corresponding to the lexeme.

# Lexical analysis - an example

Consider the statement:

```
for (i = 0; i < amount; i++) sum += x[i];
```

The character stream (with their ASCII values) is:

```
 f   o   r      (  i  =  0  ;      i     <     a  m  o  u  n  t ...
66  6F  52 20 28 69  3D  30 3B  20 69 20 3C 20 61 6D 6F 75 6E 74 ...
```

The scanner assembles the following lexemes:

**for ( i = 0 ; i < …**

and finds in the symbol table the corresponding tokens:

***for openparen identifier Assign NumLiteral***

***semicolon identifier lessthan … …***

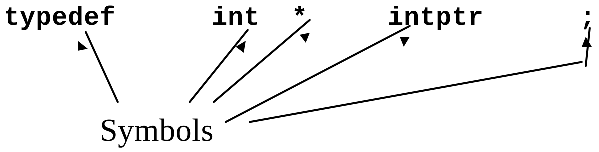
# Tokens and Lexemes

- In many instances, there is a one-to-one correspondence between the lexemes and tokens for reserved words and operators.
- User-defined identifiers are usually assigned the token of ***Identifier***.
- Numbers are usually assigned the token of ***NumericLiteral*** or something more specific like ***IntegerLiteral***.
- Characters and strings are assigned the token of ***CharacterLiteral***.

## A Brief Introduction to Formal Language Theory

***String*** - A sequence of symbols

e. g., ABabbCaC



Alphabet - A finite set of symbols.

e. g., **A**, ***B***, ***C***

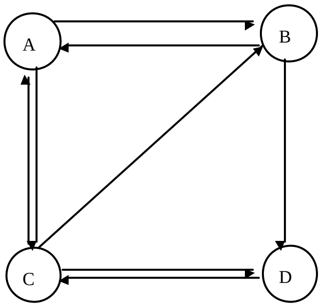
**1**, ***2***, ***3***

***ARRAY***, ***SET***, ***;***, ***OF***, **+**

## A Brief Introduction to Formal Language Theory (continued)

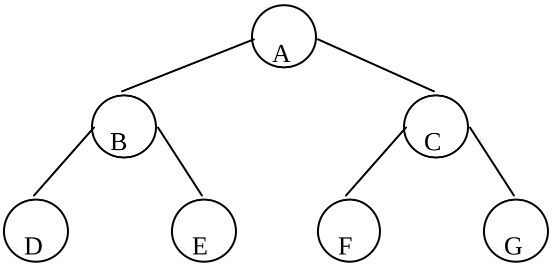
***Language*** - Any set of string over an alphabet.

***Graphs*** - A finite set of vertices and arcs.



## A Brief Introduction to Formal Language Theory (continued)

***Trees*** - a directed graph without circuits.



## A Brief Introduction to Formal Language Theory (continued)

***Terminals*** - any symbol in a given language’s alphabet. In formal language theory, they are represented by lower-case letters (i. e., ***a*** ***b***)

e. g., ***int***, ***while***, ***class*** are terminals in C++.

***Nonterminals*** - any set of combinations of terminals. A combination of terminals can be derived from a nonterminal, according to the productions (rules) of the grammar of the language. Usually represented by capital letters (i. e., ***A*** ***B***)

## A Brief Introduction to Formal Language Theory (continued)

Examples of nonterminals and the productions in which they appear:

Expression ::= Term (±Term)$^{*}$

Term ::= Factor ({*|/} Factor)$^{*}$

Factor ::= Identifier

Factor ::= Constant

***Variables*** - Any terminal or nonterminal usually represent by a Greek letter. ***ß***

## Chomsky Hierarchy

| Type | Language | Automaton |
|---|---|---|
| 0 | Recursively enumerable<br>(or unrestricted) | Turing Machine |
| 1 | Context-sensitive | Linear-bounded<br>Turing Machine |
| 2 | Context-free | Pushdown Automaton |
| 3 | Regular | Finite Automaton |

Type N automata are computer implementations designed to process type N language.

## Languages & Grammars

Type 0 - Recursive enumerable a **::= ß**,

where *a* and ß can be any string or variable.

Type 1 - Context-sensitive a **::= ß**,

where |*a* | < |ß| and at least one character in *a* is a nonterminal

Type 2 - Context-free **A ::= ß**,

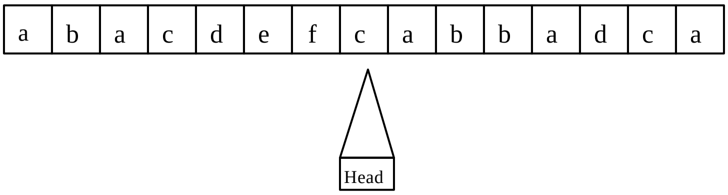
where there is one and only nonterminal and NO terminals on the left.

Type 3 - Regular **A ::= a** or **A ::= aB**

Most real programming languages are almost context-free (with a few context-sensitive traits)

# Automata

***Turing Machine*** - Given an input stream, it performs as many computations as necessary, finally deciding whether to accept (if the string is within the language).



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | c | d | e | f | c | a | b | b | a | d | c | a |

***Linearly-bounded Automaton*** - works like a Turing Machine, but limits space to the length of the input string.

# Automata (continued)

***Pushdown automaton*** - Uses a stack, and it can read from only the stack.

***Finite Automaton*** - The machine cannot write anything. After reading the string, it either accepts or rejects the string.

In theory, computers can be as powerful as a Turing Machine.

# Deterministic Finite Automata

***Definition*** – A finite deterministic automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

Q is a finite set of states

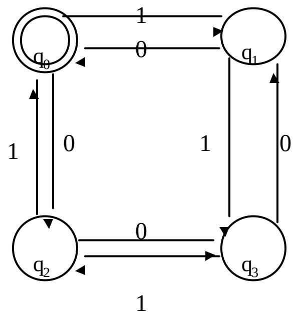
$\Sigma$ is a finite alphabet

$q_0 \in Q$ is the initial state

$F \subseteq Q$ is a set of final states

$\delta$ is a transition function

# Deterministic Finite Automata (continued)



$\Sigma = \{ 1, 0\}$

$Q = \{ q_0, q_1, q_2, q_3 \}$

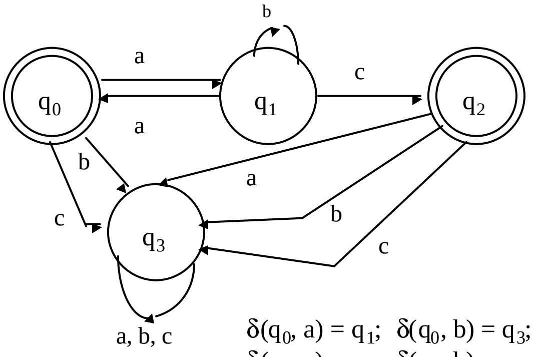
$F = \{ q_0 \}$

$\delta(q_0, 0) = q_2$; $\delta(q_0, 1) = q_1$; $\delta(q_1, 0) = q_{0;}$ $\delta(q_1, 1) = q_3$

$d(q_2, 0) = q_3$; $\delta(q_2, 1) = q_{0;}$ $\delta(q_3, 0) = q1$; $\delta(q_3, 1) = q_2$

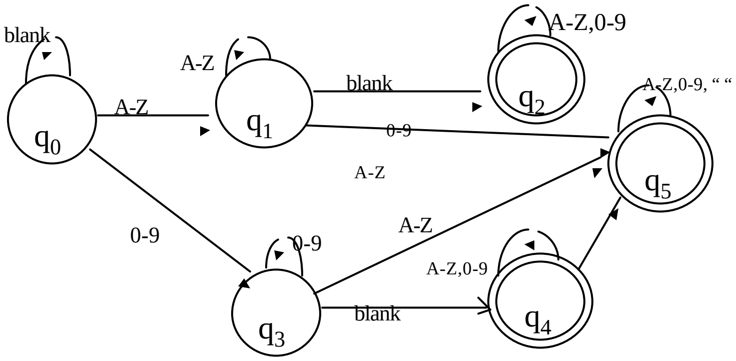
$\delta$ maps $Q \times \Sigma$ into Q

# Deterministic Finite Automata (continued)



$\delta(q_0, a) = q_1$; $\delta(q_0, b) = q_3$; $\delta(q_0, c) = q_3$
$\delta(q_1, a) = q_0$; $\delta(q_1, b) = q_1$; $\delta(q_1, c) = q_2$
$\delta(q_2, a) = q_3$; $\delta(q_2, b) = q_3$; $\delta(q_2, c) = q_3$
$\delta(q_3, a) = q_3$; $\delta(q_3, b) = q_3$; $\delta(q_3, c) = q_3$

# What language will this DFA accept?



$\Sigma$ = {A-Z, 0-9, blank}

## Regular Expressions

Regular expressions have an indefinite repetition of symbols in its accepted language.

(01)* = {01, 0101, 010101, 01010101, 0101010101, ...}

1*001* = {0000, 100, 10001, 1000, 1001, 100001, 10000, ...}

(0+10)* = {0, 10, 100, 1010, 010, 0100...}

(1+ε)(0+10)* = {1, 010, 100, 1100, 010, 1010, ...}

(0+1)*101 = {101, 00101, 0101, 10101, 00101, 1011101, 000101, ...}

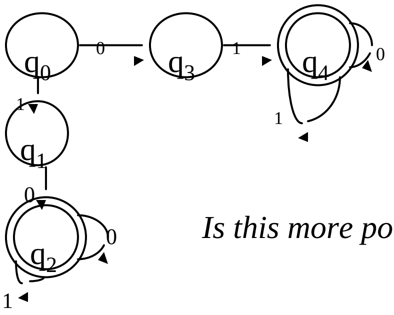
$0^+1^+2^+$ = 00*11*22* = {001112, 012, 0112, 00111222, ...}

b*(aa)*c* = {?}

## Deterministic vs. Nondeterministic Automata

A deterministic finite automaton makes one and only move in a given state-symbol combination.

A ***Nondeterministic*** finite automaton can make 0 or more moves for such a combination.

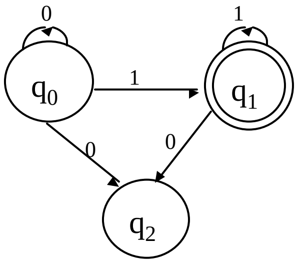


M = (Q, Σ, δ, $q_0$, F) as before but δ(Q, Σ) may be a set or undefined

*Is this more powerful than a DFA?*

# Nondeterministic Finite Automata

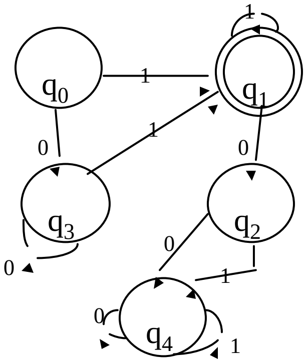
A nondeterministic finite automaton will have an equivalent deterministic automaton.



The NFA ***M*** is defined as
m = (Q, Σ, $q_0$, δ, F) where
Q = { $q_0$, $q_1$, $q_2$}
Σ = { 0, 1 }
F = { $q_1$ }

| d | **$q_0$** | **$q_1$** | **$q_2$** |
|---|---|---|---|
| 0 | {$q_0$, $q_2$} | $q_2$ | ϕ |
| 1 | $q_1$ | $q_1$ | ϕ |

# Nondeterministic Finite Automata (continued)



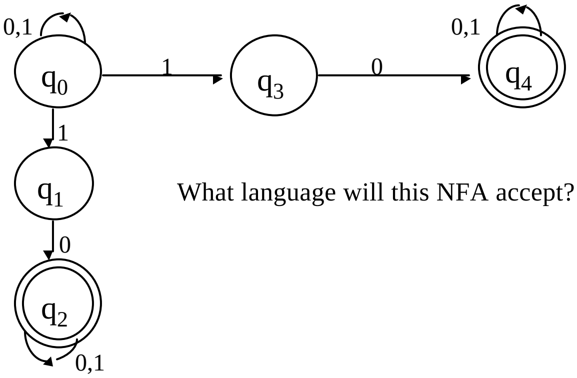
The equivalent Deterministic Finite Automaton

# The equivalence of DFAs and NFAs

A DFA and an NFA are equivalent if their 5-tuple are equivalent.

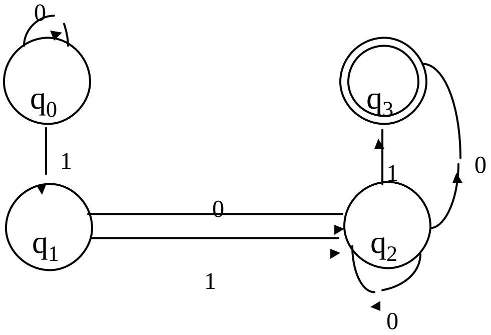
They are also equivalent if they accept the same language.

E. g.,



What language will this NFA accept?

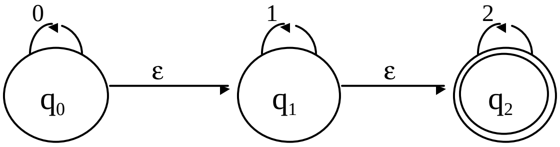
# The equivalence of DFAs and NFAs (continued)



*What language will this DFA accept?*

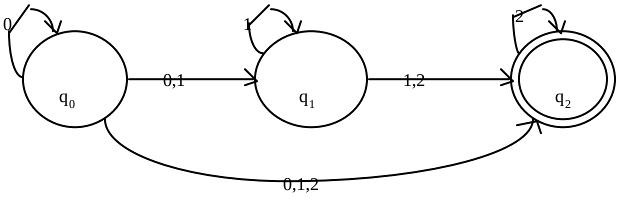
# NFA with epsilon-moves

NFAs can contain ε-moves, where ε is the empty string. It takes us to another state without having to read another character.



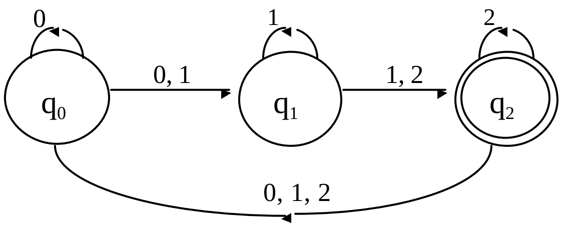
# NFAs with epsilon moves (continued)

To convert this to the equivalent NFA without ε-moves, we need to find ε-closure(q), the set of all states p which can be reached from q by ε-moves.



## NFAs with epsilon moves (continued)

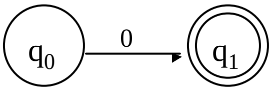
To convert this to the equivalent NFA without ε-moves, we need to find ε-closure(q), the set of all states which can be reached from q by ε-moves.



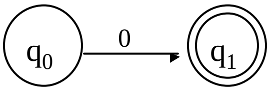
## Why are DFAs, NFAs and NFAs with epsilon moves important?

- We can automate the construction of NFAs with epsilon moves for regular expressions.
- From there, we can build NFAs with epsilon moves, and in turn, a DFA.
- A DFA is easy to implement in a computer program procedure.

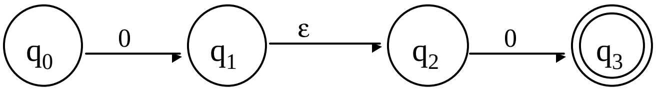
# A few basic NFAs



r = 0



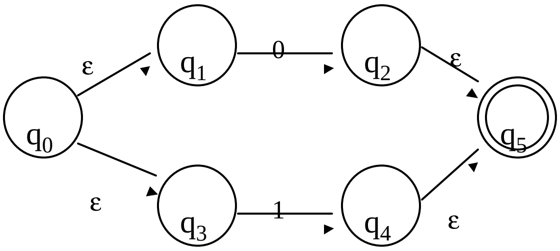
r = 1



r = 0

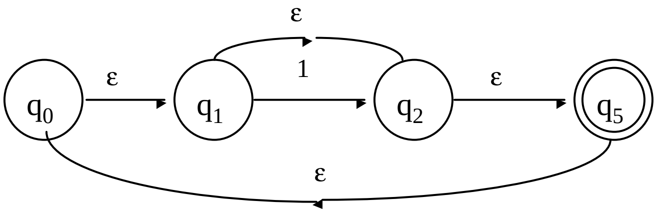
r = 1

# A few basic NFAs (continued)



r = 0+1

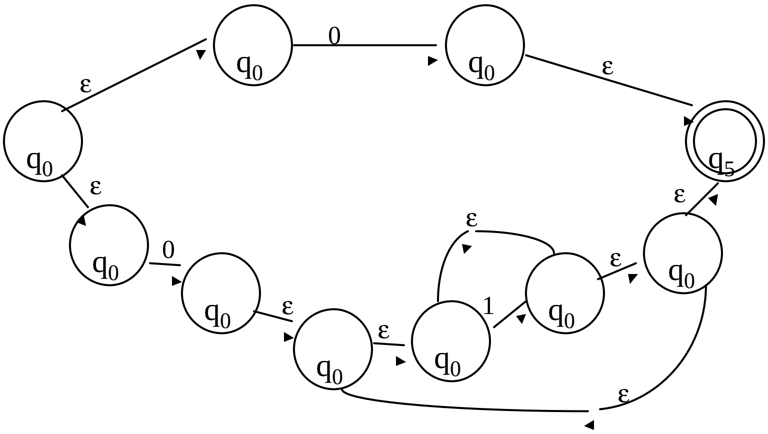
# A few basic NFAs (continued)



r = 1*

# Combining the basic NFAs

r = 0 + 0*1



## Transition Diagrams

- Transition diagrams are a special form of finite automaton, incorporating features that belong in a compiler's scanner:
  - Actions associated with final states.
  - Backup from a state, allowing for a lookahead character being returned to the input stream.
  - Transitions can be labeled as belonging to "other", indicating any class of character not explicitly accounted for.

## Transition Diagrams(continued)

In drawing transition diagrams, it is helpful to use an alternate approach to describing regular expressions:

a|b denotes a ***or*** b.

ab denotes a ***followed by*** b

(a|b)* denotes a followed by b ***zero or more times***

(a|b)c denotes a or b followed by c

## Transition Diagrams(continued)

The different lexical categories or ***classes*** can be described in this fashion:

letter : (a | b | c | d | e .... A | B | C | D | E ..| X | Y | Z)

digit:   ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )

other: ( ! | @ | # | $ | % | ^ | & | * | ( | ) | _ | + | = | - |
` | ~ | { | } | \ | ” | ’ | : | ; )

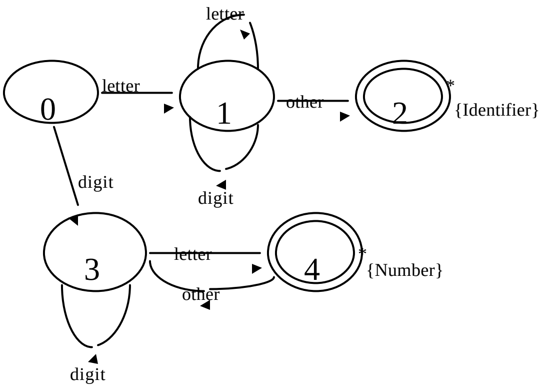
identifier :    letter (letter | digit)*

integer : digit digit*

real: (digit digit* . digit digit*) |
(digit digit* . digit digit* (E | e) (+|- | ) digit digit*)

## Transition Diagrams(continued)

The transition diagram for our language shown before becomes:



## Practical Issues in Lexical Analysis

There are several important practical issues that arise in the design of a scanner:

- Lookahead
- Case sensitivity
- Skipping of lead blanks and comments
- Use of first characters

## Lookahead characters

Since you cannot determine if you have read beyond the end of a lexeme until you have done so, you must be prepared to handle the “lookahead” character. There are two approaches available:

Start with a lookahead character and fetch a new one every time the lookahead character is “consumed” by the lexeme.

•Use two functions to manipulate the input stream, one to “get” the next character and one to “unget” the next character, returning it temporarily to the input stream.

## Lookahead characters (continued)

```
// gettc() -      Fetches a character from a
//                file.  It uses get and adjusts
//                the line number count when
//                necessary.
char  scanner::gettc(void)
{
  char      c;

  //  If we' re at the end of file, return a null
  //  byte, which serves to mark the end of file.
  if (infile.eof())
            c = '\0';
```

## Lookahead characters (continued)

```
  //  If the next character is a newline,
  //  increment the line count
  else if ((c = infile.get()) == '\n')
      linenum++;
  // Return the character converted to lower case
  return(tolower(c));
}
```

## Lookahead characters (continued)

```
// ungettc() -    Returns a character to the
//                file.  Uses ungetc and will
//                adjust line number count.
void  scanner::ungettc(char c)
{
  //  If it's a newline, decrement the line
  //  count; we haven't gone to the next line
  //  yet.
  if (c == '\n')
      --linenum;
  //  Put it back into the input stream.
  infile.putback(c);

}
```

## Case sensitivity

- Although “a” and “A” are regarded as the same character in the English language, they are represented by different ASCII codes. For a compiler to be case insensitive, we need to consider these both as the same letter.
- The easiest way to do this is to convert all letters to the same case.
- Not all languages do this, e.g., C.

## Skipping lead blanks and comments

- Before reading the first significant character in a lexeme, it necessary to skip past both lead blanks as well as comments.
- One must assume that the scanner can encounter either or both repeatedly and interchangeably before reading the first significant character.

### Skipping lead blanks and comments (continued)

```
// firstchar() -  Skips past both white space
//                and comments until it finds
//                the first non-white space
//                character outside a comment.
char  scanner::firstchar(void)
{
      char  c;
      bool  goodchar = false;


      //    If we're at the end of the file,
      //    return the EOF marker so that we'll
      //    return the EOF token
      if (infile.eof())
                  return(EndOfFile);
```

## Skipping lead blanks and comments (continued)

```
      //    We're looking for a non-white space
      //    character that is outside a comment.
      //    Keep scanning until we find one or
      //    reach the end of the file.
      while (!goodchar) {
            //    Skip the white space in the
            //    program
            while (!infile.eof()
                        && isspace(c = gettc()))
                  ;

            //    Is it a comment or a real
            //          first character?
            if  (c != '{')
                  goodchar = true;
```

## Skipping lead blanks and comments (continued)

```
            else
                  // Skip the comment
                  while (!infile.eof()
                        && (c = gettc()) != '}')
                        ;
      }

      //    If we're at the end of file, return
      //    the EOF marker. Otherwise, return
      //    the character.
      if (infile.eof())
            return(EndOfFile);
      else
            return(c);
}
```

## Use of first character

- In most programming languages, the first character of a lexeme indicates the nature of the lexeme and token associated with it.
- In most instances, identifiers and reserved words begin with a letter (followed by zero or more letters and digits), numbers begin with a digit and operators begin with other characters.

## Use of first character (continued)

```
// gettoken() -   Scan out the token strings of
//                the language and return the
//                corresponding token class to the
//                parser.
tokentype   scanner::gettoken(int &tabindex)
{
      char  c;

      // If this is the end of the file, send the
      // token that indicates this

      if ((c = lookahead) == EndOfFile)
            return(tokeof);
```

## Use of first character (continued)

```
      // If it begins with a letter, it is a word.
      // If begins with a digit, it is a number.
      //  Otherwise, it is an error.
      lookahead = gettc();
      if (isalpha(c))
            return(scanword(c, tabindex));
      else if (isdigit(c))
            return(scannum(c, tabindex));
      else
            return(scanop(c, tabindex));

}
```

## Scanning for reserved words and identifiers

- Once the scanner determines that the first character is a letter, it continues to read characters and concatenate them to the lexeme until it encounters a character other than a letter or digit.
- If the resultant lexeme is not in the symbol table, it must be a new identifier.

## Scanning for reserved words and identifiers (continued)

```
// scanword() -   Scan until you encounter
//                something other than a letter.
tokentype   scanner::scanword(char c,
                                       int &tabindex)
{
      char  lexeme[LexemeLen];
      int         i = 0;

      // Build the string one character at a time.
      // It keeps scanning until either the end of
      // file or until it encounters a non-letter
      lexeme[i++] = c;
      while ((c = lookahead) != EndOfFile
            && (isalpha(c) || isdigit(c)))      {
            lexeme[i++] = c;
            lookahead = gettc();
      }
```

```
      //    Add a null byte to terminate the
      //    string and get the lookahead that
      //    begins the next lexeme.
      lexeme[i] = '\0';
      ungettc(lookahead);
      lookahead = firstchar();

      //    If the lexeme is already in the symbol
      //    table, return its tokenclass.  If it
      //    isn't, it must be an identifier whose
      //    type we do not know yet.
      if (st.installname(lexeme, tabindex))
            return(st.gettok_class(tabindex));
      else  {
            st.setattrib(tabindex, stunknown,
                                  tokidentifier);
            return(tokidentifier);
      }
}
```

## Scanning for numeric literals

- After determining that the lexeme begins with a digit, the scanner reads characters, concatenating them to the lexeme until it encounters a non-digit.
- If it is a period, it will concatenate this to the lexeme and resume reading characters until it encounters another non-digit.
- If it is an “E”, it must then read the exponent.
- The token associated with the lexeme is either number or the number’s type.

### Scanning for numeric literals (continued)

```
// scannum() -    Scan for a number.
tokentype   scanner::scannum(char c,int &tabindex)
{
      int         ival, i = 0;
      bool        isitreal = false;
      float       rval;
      char        lexeme[LexemeLen];

      // Scan until you encounter something that
      // cannot be part of a number or the end of
      // file
      lexeme[i++] = c;
      while ((c = lookahead) != EndOfFile
                              && isdigit(c))    {
            lexeme[i++] = c;
            lookahead = gettc();
      }
```

### Scanning for numeric literals (continued)

```
// Is there a fractional part?
if (c == '.')     {
      isitreal = true;
      lexeme[i++] = c;
      while ((c = lookahead) != EndOfFile
                        && isdigit(c))    {
            lexeme[i++] = c;
            lookahead = gettc();
      }
}

//    Add a null byte to terminate the
//    string and get the lookahead that
//    begins the next lexeme.
ungettc(lookahead);
lexeme[i] = '\0';
lookahead = firstchar();
```

### Scanning for numeric literals (continued)

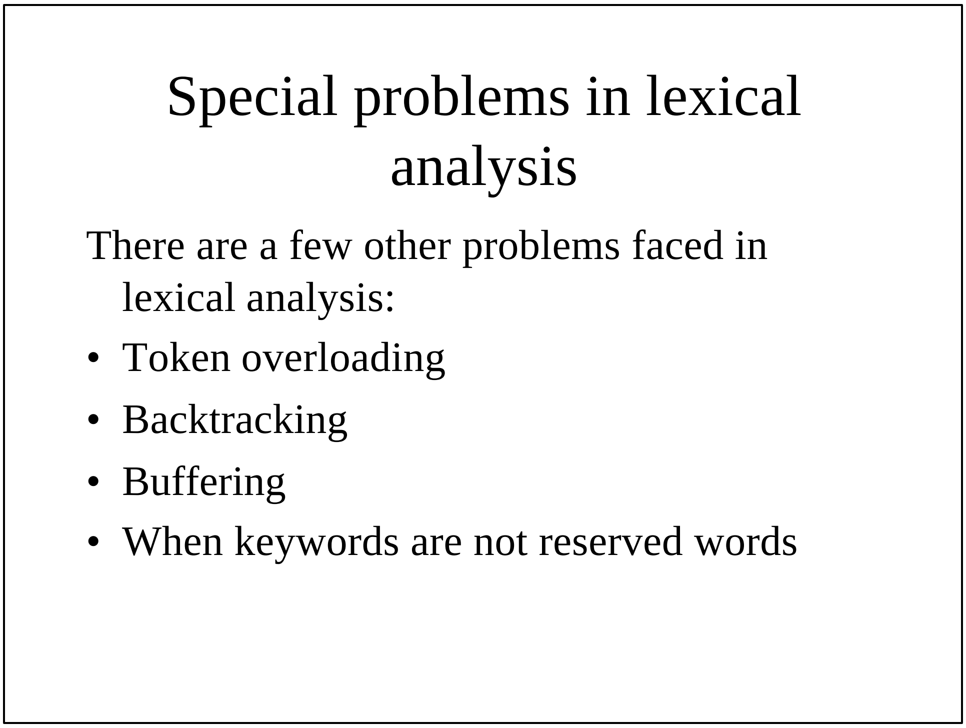
```
// If there is no fractional part, it is an
// integer literal constant.  Otherwise, it
// is a real literal constant. Firstly, is
// it already in the symbol table?
if (st.installname(lexeme, tabindex))
      return(st.gettok_class(tabindex));
// If not, is it real?
else if (isitreal)      {
      st.setattrib(tabindex, stunknown,
                              tokconstant);
      st.installdatatype(tabindex,
                        stliteral, dtreal);
      rval = atof(lexeme);
      st.setvalue(tabindex, rval);
      return(st.gettok_class(tabindex));
}
```

Scanning for numeric literals (continued)

```
      // Must be an integer literal
      else  {
            st.setattrib(tabindex, stunknown,
                                    tokconstant);
            st.installdatatype(tabindex,
                        stliteral, dtinteger);
            ival = atoi(lexeme);
            st.setvalue(tabindex, ival);
            //ungettc(lookahead);
            return(st.gettok_class(tabindex));
      }
      ungettc(lookahead);
      return(st.gettok_class(tabindex));
}
```

## Scanning for operators and characters literals

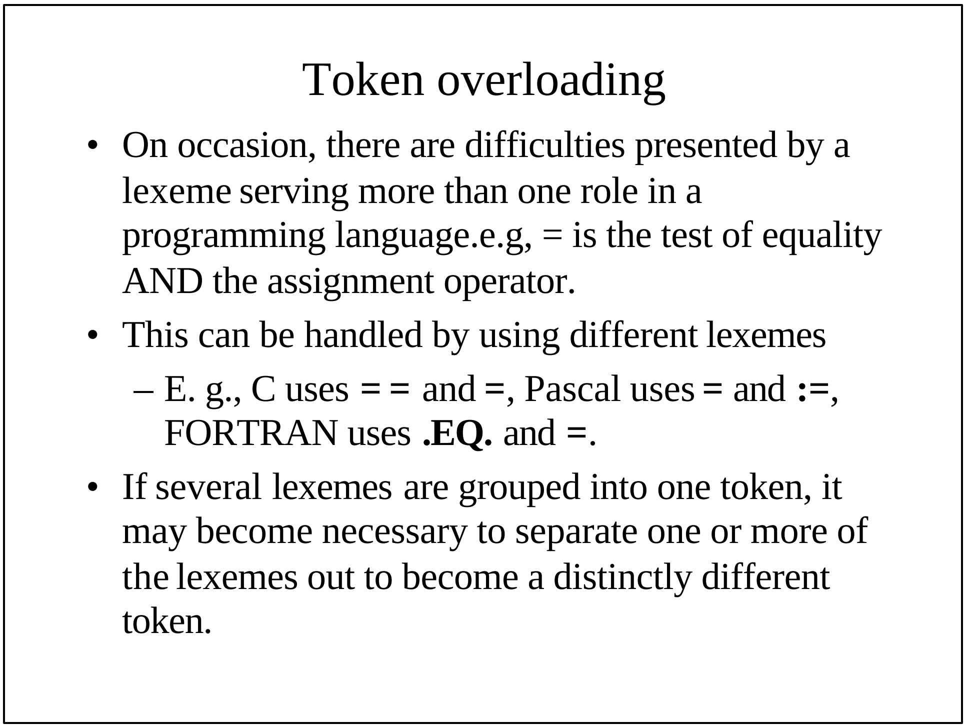
- If the first character is neither a letter nor a digit, the lexeme must be one of the following:
  - an operator
  - a character literal
  - a string literal
- In scanning an operator:
  - we should be cognizant of how many characters it may contain.
  - we may wish to hand-code the token that will be returned by the symbol table.
- In scanning a literal, we read characters until encountering the appropriate closing quotation mark.



# Special problems in lexical analysis

There are a few other problems faced in lexical analysis:

- Token overloading
- Backtracking
- Buffering
- When keywords are not reserved words



# Token overloading

- On occasion, there are difficulties presented by a lexeme serving more than one role in a programming language.e.g, = is the test of equality AND the assignment operator.
- This can be handled by using different lexemes
  - E. g., C uses **= =** and **=**, Pascal uses **=** and **:=**, FORTRAN uses **.EQ.** and **=**.
- If several lexemes are grouped into one token, it may become necessary to separate one or more of the lexemes out to become a distinctly different token.

# Backtracking

- In rare instances, it may become necessary to backtrack and re-scan the text of the program.

  E.g., the ***DO*** statement in FORTRAN

  **DO 101  I = 1, 50**

  is initially read as

  **DO101 = 1**

  until the , is encountered.

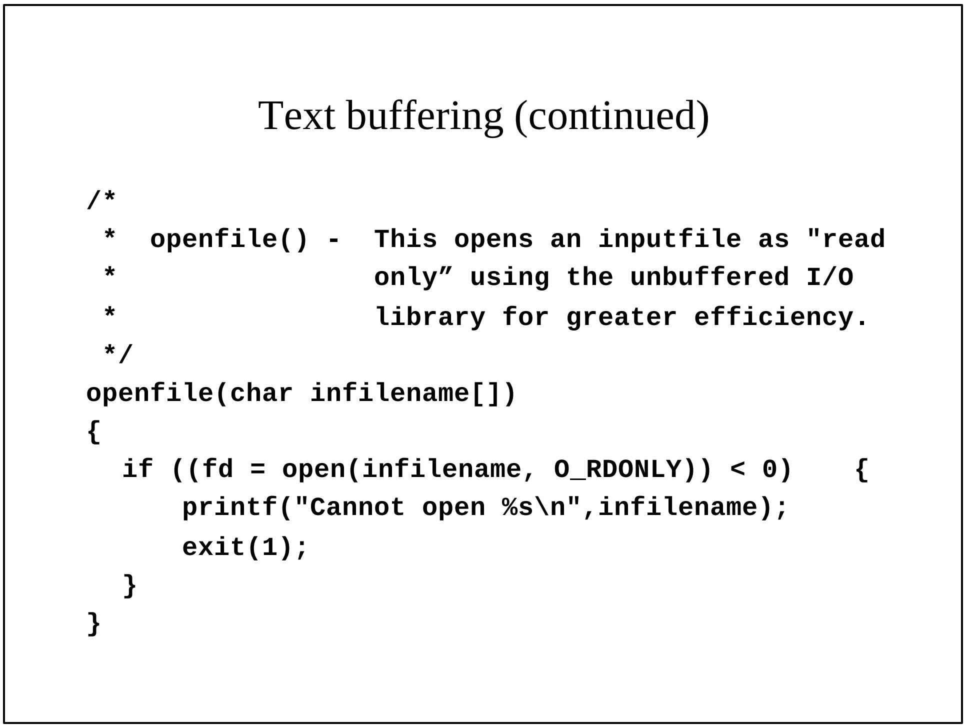
# Text buffering

- Reading file input is a time-consuming process.  This makes the buffering of input text crucial to the efficiency of a compiler.
- In most instances, file input is buffered on modern operating systems, rendering the issue less important than a decade ago.

## Text buffering (continued)

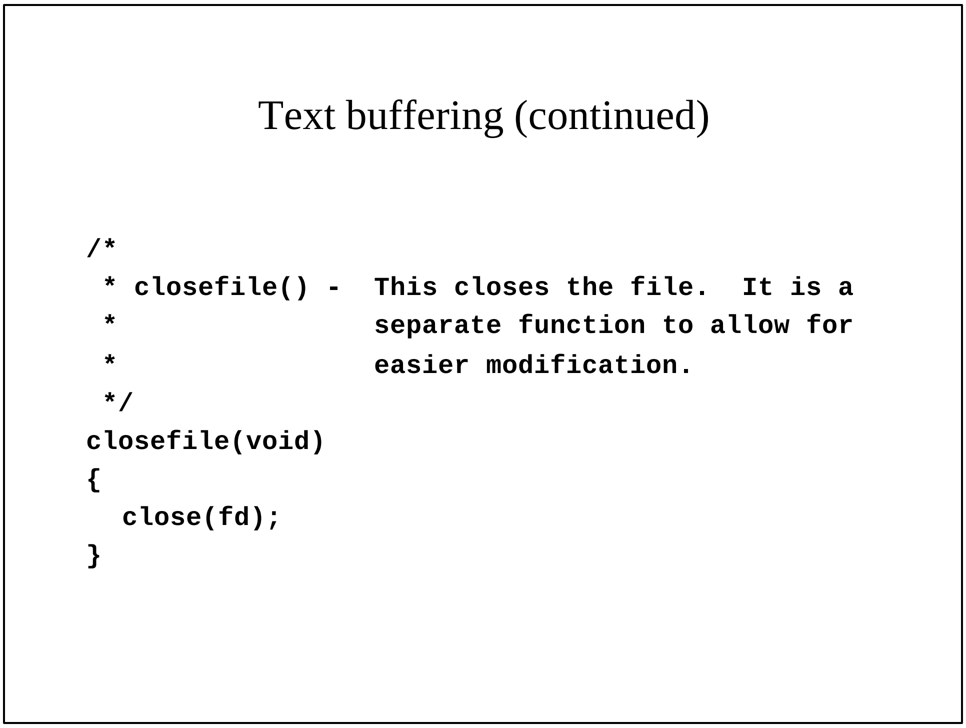
```
#define           NUMBYTES    512
#define           NUMBUFFERS  2
#define           MAXSTACK    2
#define gettch()(top > 0 ? buffer[--top]: fetchchar())

int         bytesread, fd, c, top, linenum = 1;
char        buf[NUMBUFFERS][NUMBYTES],
            buffer[MAXSTACK], inputstring[MAXLINE];
```

```
/*
 *  ungettch() -  This function, together with the
 *                macro gettch(),   allows the
 *                program to push and pop
 *                characters to and from the
 *                lookahead buffer.
 */
ungettch(char c)
{
  if (top > MAXSTACK)   {
      printf("\nToo many characters \"ungotten\””
                        “\n");
      exit(1);
  }
  buffer[top++] = c;
}
```



## Text buffering (continued)

```
/*
 *  openfile() -  This opens an inputfile as "read
 *                only” using the unbuffered I/O
 *                library for greater efficiency.
 */
openfile(char infilename[])
{
  if ((fd = open(infilename, O_RDONLY)) < 0)    {
      printf("Cannot open %s\n",infilename);
      exit(1);
  }
}
```



## Text buffering (continued)

```
/*
 * closefile() -  This closes the file.  It is a
 *                separate function to allow for
 *                easier modification.
 */
closefile(void)
{
  close(fd);
}
```

# Text buffering (continued)

```
/*
 * fetchchar() -  This function uses two buffers
 *                of 512 bytes (one block in
 *                MS-DOS), and unbuffered I/O
 *                to fetch a single character at
 *                a time.  When it reaches the
 *                end of the buffer, it gets
 *                another 512 bytes until end of
 *                file.
 */
```

# Text buffering (continued)

```
int         fetchchar(void)
{
  static int      nextchar = NUMBYTES,
                  thisbuf = NUMBUFFERS-1;
  if (nextchar >= bytesread)
      /*  Buffer is full
            Fill the next buffer */
      if ((bytesread = read(fd,
            buf[thisbuf
      = (thisbuf == NUMBUFFERS-1)?0:thisbuf+1],
                              NUMBYTES-1)) <= 0)
            /* Reached end of file. */
            return(EOF);
```

## Text buffering (continued)

```
      else
             /* Reset buffer pointer */
            nextchar = 0;
  return(buf[thisbuf][nextchar++]);
}
```

## When keywords are not reserved words

- The keywords of a programming language are usually reserved, i. e., they cannot be used by a programmer as an identifier, a user-defined variable, data type, etc.
- There are programming languages where this is not the case, making programs difficult to understand and making it difficult to return the proper token. E. g.,

-
  ```
  IF THEN  THEN THEN = ELSE;
           ELSE ELSE = THEN;
  ```

## Scanner generators

- Scanner generators automatically generate a scanner given the lexical specifications and software routines given by the user.
- Scanner generators take advantage of the fact that a scanner is essentially an implementation of a finite automaton and can thus be created in an automated fashion.
- LEX is an example of such a software tool.