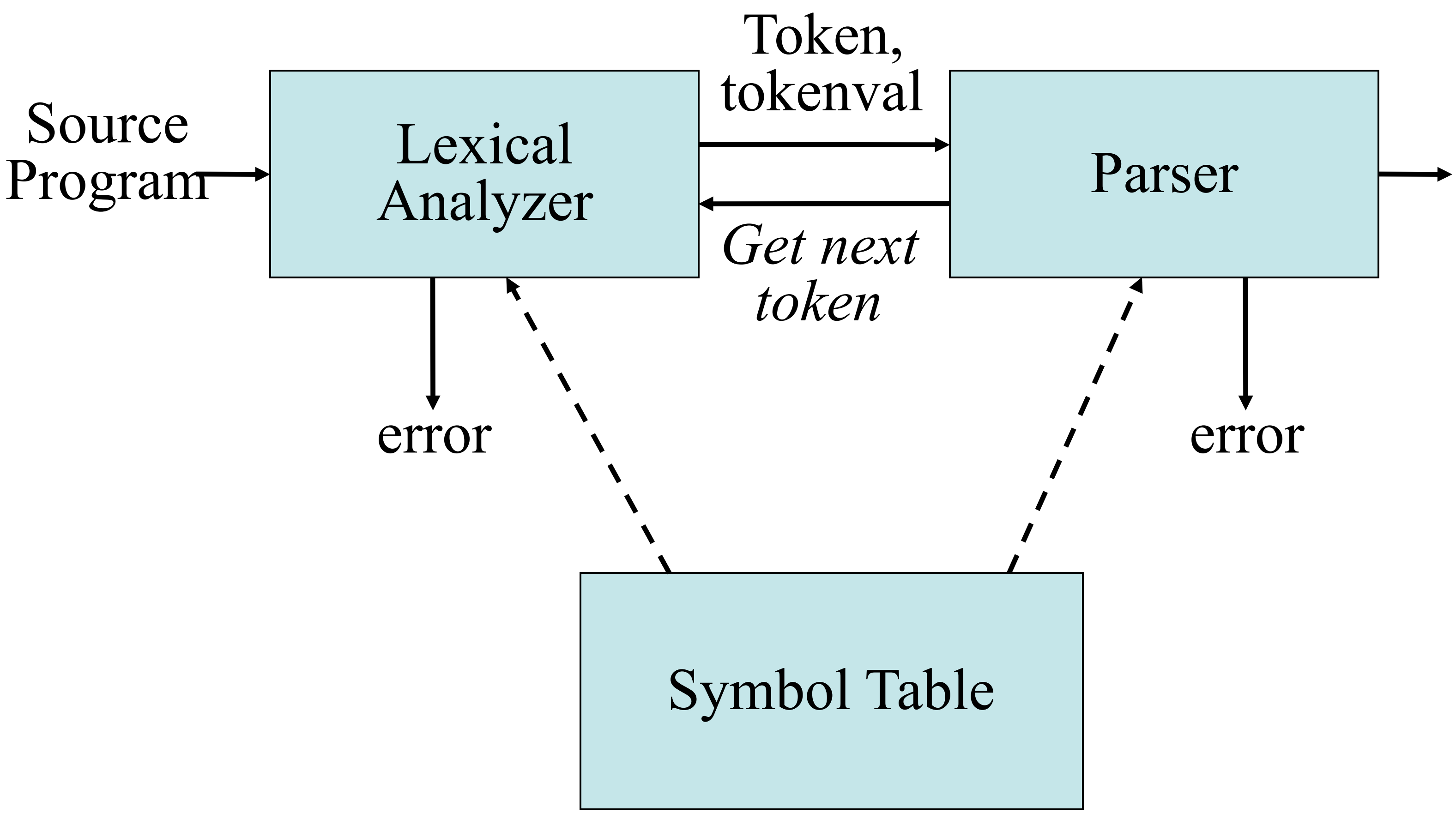
Lexical Analyzer (Scanner)

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Tokens, Lexemes, Patterns

- A *token* is a classification of lexical units
  - For example: **id** and **num**

- *Lexemes* are the specific character strings that make up a token
  - For example: **`abc`** and **`123`**

- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"

Token

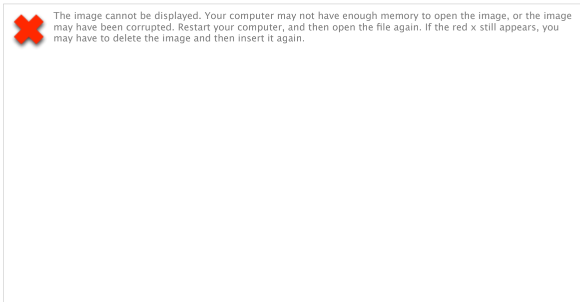
- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits
  - The actual string (newval) is called as *lexeme*.
  - Tokens: identifier, number, addop, delimeter, …
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the *attribute* of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.
- Some attributes:
  - <id,attr> where attr is pointer to the symbol table
  - <assgop,_> no attribute is needed (if there is only one assignment operator)
  - <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identifies a lexeme.
- ***Regular expressions*** are widely used to specify patterns.

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - $|s|$ is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - $\varnothing$ the empty set is a language.
  - $\{\varepsilon\}$ the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\quad \varepsilon\, s = s$
  - $s^n = s\,s\,s\,..\,s$ (n times) $\quad s^0 = \varepsilon$

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $\quad L^1 = L \quad$ $L^2 = LL$
- Kleene Closure
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- Positive Closure
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$



Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - `*` highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon ,0,00,000,0000,....\}$
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter → A | B | ... | Z | a | b | ... | z

  digit → 0 | 1 | ... | 9

  id → letter (letter | digit) *

  – If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

  (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit → 0 | 1 | ... | 9

  digits → digit $^+$

  opt-fraction → (. digits) ?

  opt-exponent → (E (+|-)? digits) ?

  unsigned-num → digits opt-fraction opt-exponent

Disambiguation Rules

1) **longest match rule:** from all tokens that match the input prefix, choose the one that matches the most characters

2) **rule priority:** if more than one token has the longest match, choose the one listed first

Examples:

- for8 — is it the for-keyword, the identifier “f”, the identifier “fo”, the identifier “for”, or the identifier “for8”?
  Use rule 1: “for8” matches the most characters.
- for — is it the for-keyword, the identifier “f”, the identifier “fo”, or the identifier “for”?
  Use rule 1 & 2: the for-keyword and the “for” identifier have the longest match but the for-keyword is listed first.

How Scanner Generators Work

- Translate REs into a finite state machine
- Done in three steps:
  1) translate REs into a no-deterministic finite automaton (NFA)
  2) translate the NFA into a deterministic finite automaton (DFA)
  3) optimize the DFA (optional)

Finite Automata

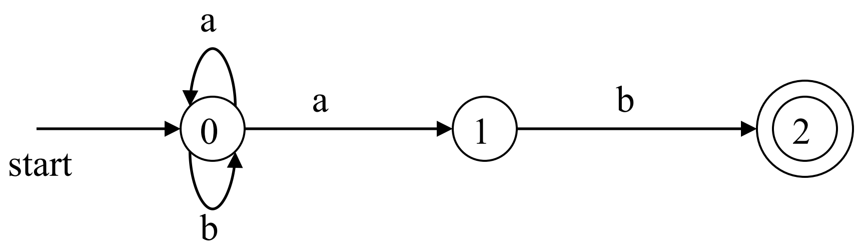
- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (Example)



Transition graph of the NFA

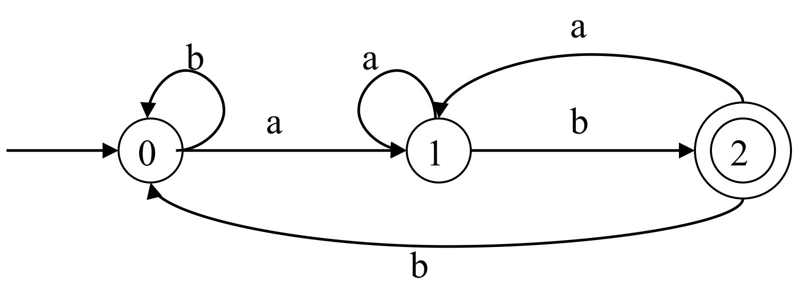
0 is the start state s_0
{2} is the set of final states F
$\Sigma = \{a,b\}$
$S = \{0,1,2\}$
Transition Function:

	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

The language recognized by this NFA is $(a|b)^*\ a\ b$

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
  - no state has ε- transition
  - for each symbol a and state s, there is at most one labeled edge a leaving s.
    i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by this DFA is also $(a|b)^*\ a\ b$

Implementing a DFA

- Le us assume that the end of a string is marked with a special symbol (say eos). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0                  { start from the initial state }
c ← nextchar            { get the next character from the input string }
while (c != eos) do     { do until the en dof the string }
    begin
        s ← move(s,c)   { transition function }
        c ← nextchar
    end
if (s in F) then        { if s is an accepting state }
    return "yes"
else
    return "no"
```

Implementing a NFA

```
S ← ε-closure({s0})            { set all of states can be accessible from s0 by ε-transitions }
c ← nextchar
while (c != eos) {
    begin
        s ← ε-closure(move(S,c))   { set of all states can be accessible from a state in S
        c ← nextchar                 by a transition on c }
    end
if (S∩F != Φ) then             { if S contains an accepting state }
    return "yes"
else
    return "no"
```

- This algorithm is not efficient.

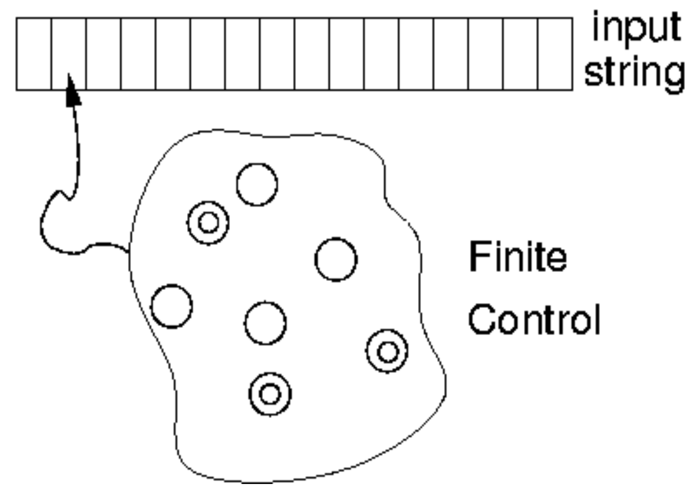
Converting A Regular Expression into A NFA (Thomson's Construction)

- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
  To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Recognizing Tokens: Finite Automata

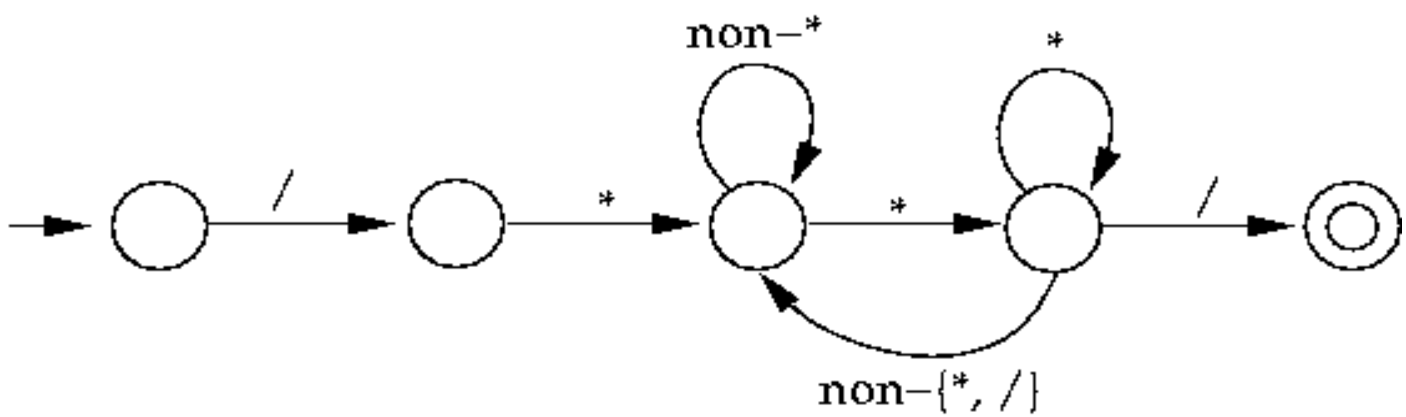
A *finite automaton* is a 5-tuple (Q, Σ, T, q0, F), where:

- Σ is a finite alphabet;
- Q is a finite set of states;
- T: $Q \times \Sigma \rightarrow Q$ is the transition function;
- q0 $\in Q$ is the initial state; and
- $F \subseteq Q$ is a set of final states.

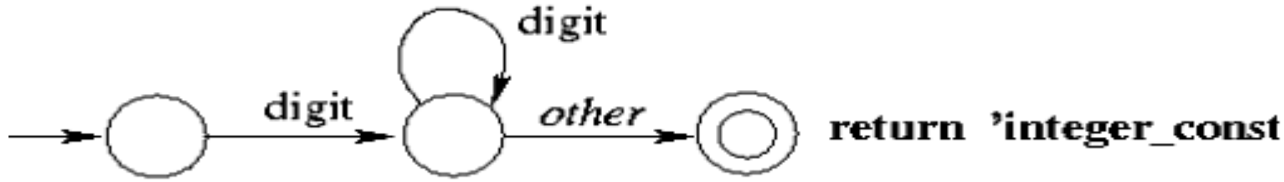


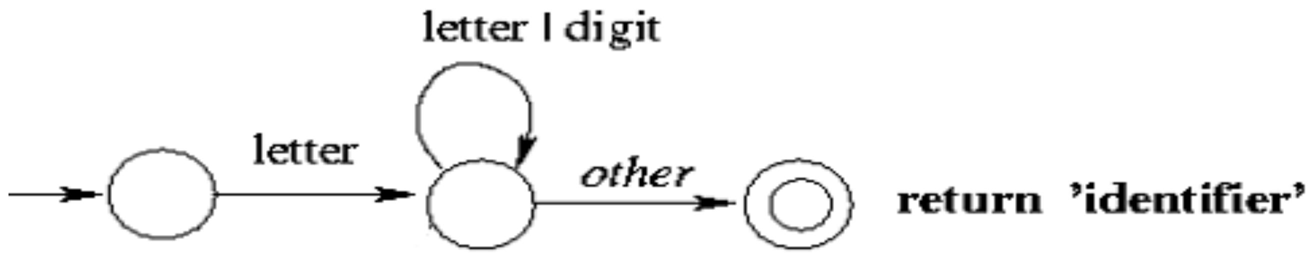
Finite Automata: An Example

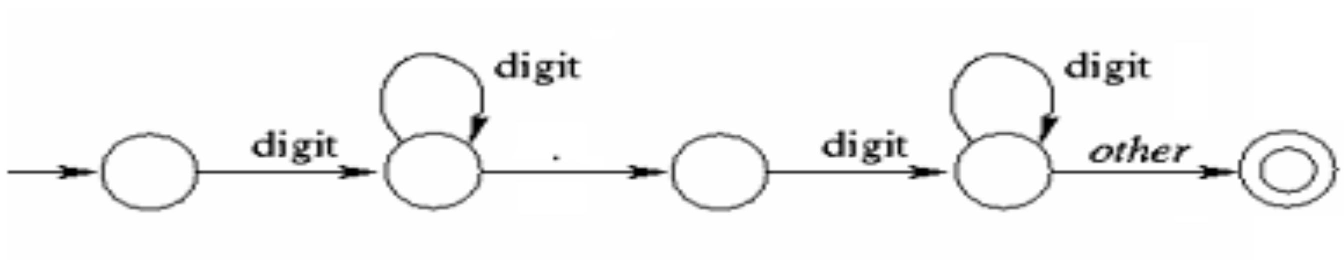
A (deterministic) finite automaton (DFA) to match C - style comments:



Identify Integer Constant, Real Constant, Identifier.

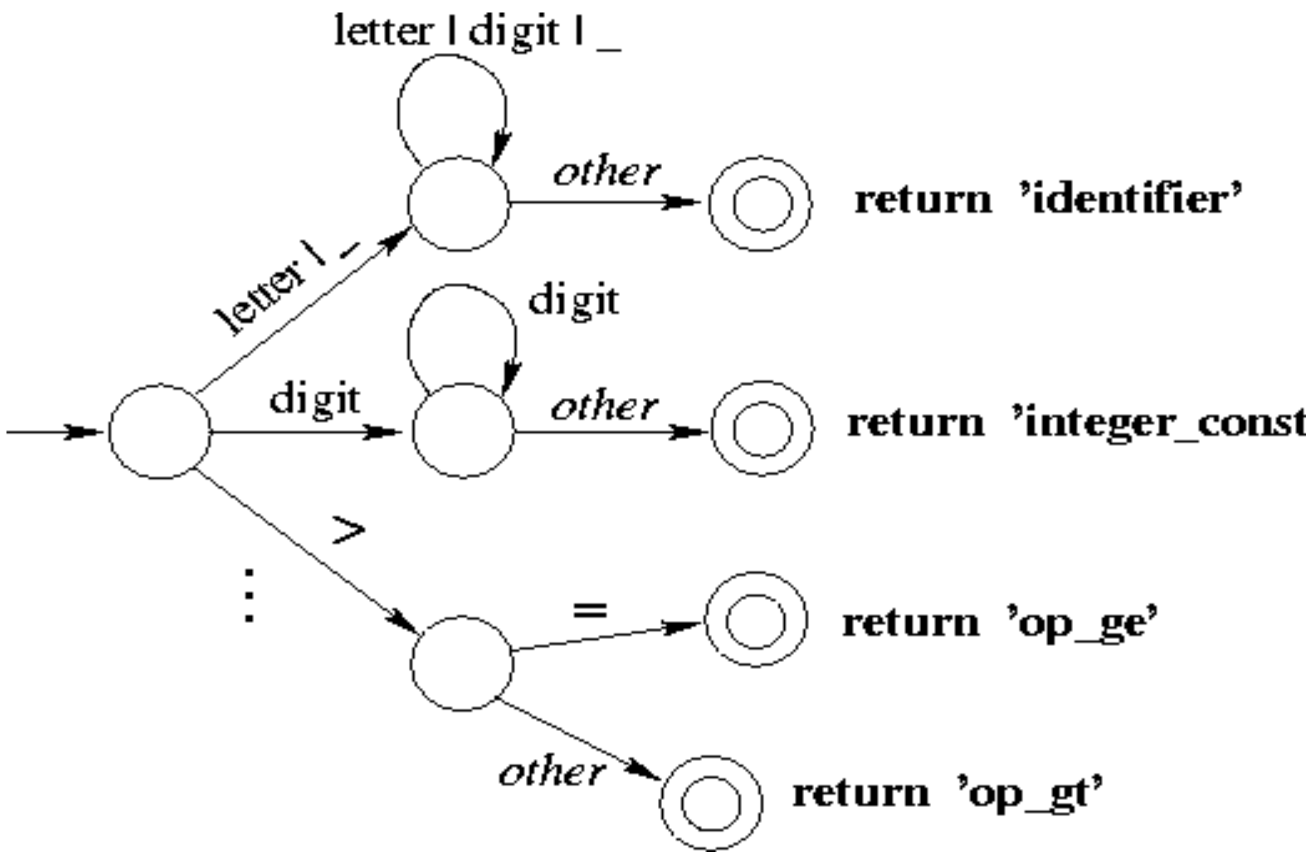






return 'float_const'

Structure of a Scanner Automaton



Implementing Finite Automata 1

Encoded as program code:

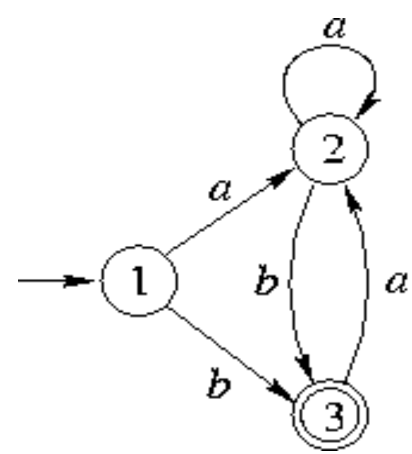
- each state corresponds to a (labeled code fragment)
- state transitions represented as control transfers.

E.g.:

```
while ( TRUE ) {
    …
    state_k: ch = NextChar();   /* buffer mgt happens here */
        switch (ch) {
          case … : goto  ...;     /* state transition */
            …
        }
    state_m:           /* final state */
        copy lexeme to where parser can get at it;
        return token_type;
    …
}
```

Direct-Coded Automaton: Example

```
int scanner()
{  char ch;
   while (TRUE) {
      ch = NextChar( );
      state_1: switch (ch) {         /* initial state */
           case ‘a’ : goto state_2;
           case ‘b’ : goto state_3;
           default : Error();
         }
      state_2: …
      state_3: switch (ch) {
           case ‘a’ : goto state_2;
           default : return SUCCESS;
         }
   } /* while */
}
```



Implementing Finite Automata 2

Table-driven automata (e.g., ***lex***, ***flex***):

- Use a table to encode transitions:

  next_state = *T*(curr_state, next_char);

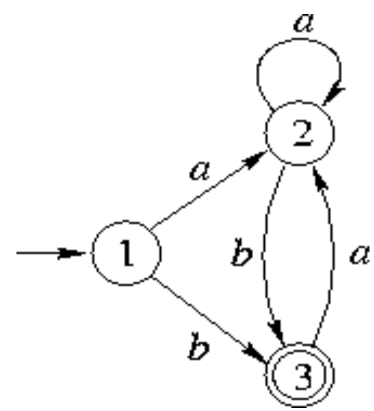
- Use one bit in state no. to indicate whether it's a final (or error) state. If so, consult a separate table for what action to take.

T	*next input character*		
Current state			

Table-Driven Automaton: Example

```
#define isFinal(s)      ((s) < 0)
int scanner()
{  char ch;
   int currState = 1;

   while (TRUE) {
      ch = NextChar( );
      if (ch == EOF) return 0;  /* fail */
      currState = T [currState, ch];
      if (IsFinal(currState)) {
         return 1;  /* success */
      }
   } /* while */
}
```



T		*input*	
		a	*b*
state	1	2	3
	2	2	3
	3	2	-1

What do we do on finding a match?

- A match is found when:
  - The current automaton state is a final state; and
  - No transition is enabled on the next input character.
- Actions on finding a match:
  - if appropriate, copy lexeme (or other token attribute) to where the parser can access it;
  - save any necessary scanner state so that scanning can subsequently resume at the right place;
  - return a value indicating the token found.

Incorporating symbol table :

- Each entry in the symbol table array **symbtable** is a record consisting of two fields: lexptr – pointing to the beginning of a lexeme, and token.

- Additional fields can hold attribute value.

Handling Reserved Words

1. Hard-wire them directly into the scanner automaton:
   - harder to modify;
   - increases the size and complexity of the automaton;
   - performance benefits unclear (fewer tests, but cache effects due to larger code size).
2. Fold them into "identifier" case, then look up a keyword table:
   - simpler, smaller code;
   - table lookup cost can be mitigated using perfect hashing.

- **The symbol table interface :**
  - Routines for storing and retrieving lexemes.
  - When lexeme is saved we also save token associated with it.
  - Following operations will perform on symbol table :
    - Insert (s, t) : Returns index of new entry for string s, token t.
    - Lookup (s) : Returns index of the entry for string s, 0 if s is not found.
- **Handling reserved word :**
  - Use symbol table routine to handle reserved word.
  - E.g. Consider tokens div and mod. We initiate symbol table using the calls
    - Insert (“div” , div);
    - Insert (“mod”, mod);
  - Any subsequent call lookup (“div”) returns the token div, so div can not be used as an identifier
  - Any reserved keywords can be handled this way.

Implementing Lexical Analyzers

Different approaches:

- Using a scanner generator, e.g., **lex** or **flex**. This automatically generates a lexical analyzer from a high-level description of the tokens.

  (easiest to implement; least efficient)

- Programming it in a language such as C, using the I/O facilities of the language.

  (intermediate in ease, efficiency)

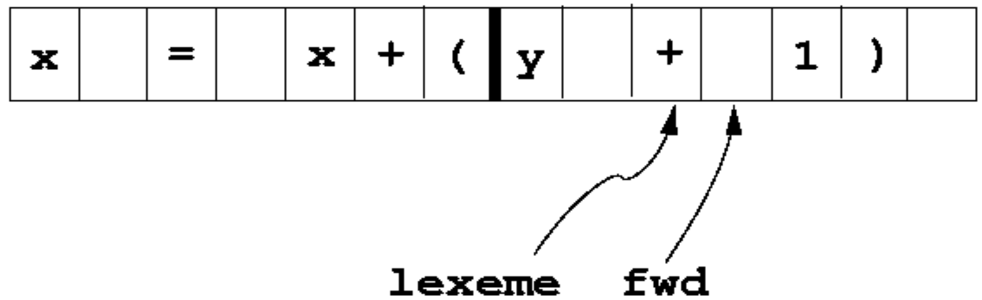
Implementing Lexical Analyzers

- Identify keywords, identifiers – id table
- Identify operators – operator table
- Identify Preprocessor Directives – Keyword # in id table
- Identify comments

Input Buffering

- Scanner performance is crucial:
  – This is the only part of the compiler that examines the entire input program one character at a time.
  – Disk input can be slow.
  – The scanner accounts for ~25-30% of total compile time.
- We need look ahead to determine when a match has been found.
- Scanners use <u>*double-buffering*</u> to minimize the overheads associated with this.

Buffer Pairs



- Use two *N*-byte buffers (*N* = size of a disk block; typically, *N* = 1024 or 4096).
- Read *N* bytes into one half of the buffer each time. If input has less than *N* bytes, put a special EOF marker in the buffer.
- When one buffer has been processed, read *N* bytes into the other buffer ("*circular buffers*").

FLex

Overview of Lex

- lex is a program (generator) that generates lexical analyzers, (widely used on Unix).
- It is mostly used with Yacc parser generator.
- Written by Eric Schmidt and Mike Lesk.
- It reads the input stream (**specifying the lexical analyzer**) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions); then produces C code for a lexical analyzer that scans for identifiers.

Cont.

- **Purpose:** to construct the scanner
- **Input:** a table of regular expressions and corresponding program fragments
  - Used to construct a deterministic finite automaton
- **Output:** a scanner, written in C, which
  - Reads an input stream (source language program)
  - Partitions input stream into strings which match regular expressions
  - Produces an output stream (list of tokens)

Skeleton of a Lex Specification (.l file)

x.l

lex.yy.c is generated after running

```
> lex x.l
```

%{

< C global variables, prototypes, comments >

%}

This part will be embedded into lex.yy.c

[DEFINITION SECTION]

%%

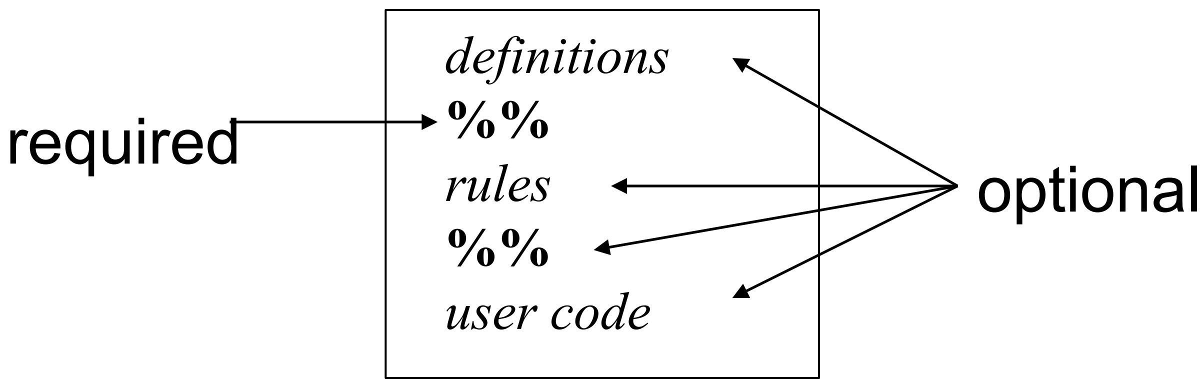
[RULES SECTION]

Define how to scan and what action to take for each token

Any user code.

Lex Source

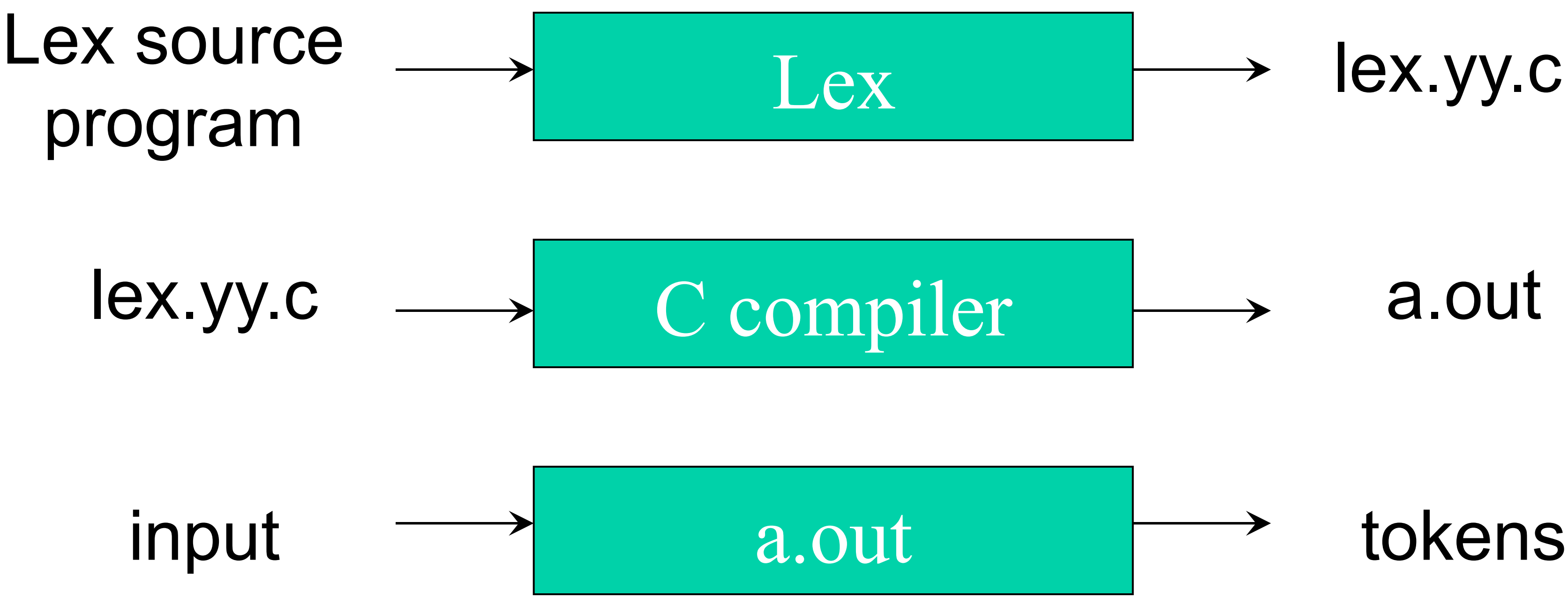
- Lex source is separated into three sections by %% delimiters



Shortest possible legal flex input:

```
%%
```

In Context of C



The Shortest Lex program

```
%%
```

- This program contains no definitions, no rules, and no user subroutines!
- It copies the input to the output without change.

Lex Program to Delete White Space at End of Lines

```
%%
[ \t]+$      ;
```

`\t` means "tab"

`[ \t]` means "either 'space' or 'tab'"

`[ \t]+` means "a string of one or more 'spaces' or 'tabs'"

`$` means "end of line"

`[ \t]+$` means "a string of one or more 'spaces' or 'tabs' followed by 'end of line'"

There is no code fragment, so the text which matches the pattern is erased and not replaced with anything.

Lex Program to Compress White Space

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

Ex. Identifier in Pascal

```
Digit                    [0-9]
Letter                   [a-zA-Z]
%%
{Letter}({Digit} | {Letter})*   printf(“\n The
  found identifier is = %s“, yytext);
```

Definition Section

- A series of:
  - *name definitions*, each of the form

    name *definition*

    e.g.:

    ```
    DIGIT          [0-9]
    CommentStart   "/*"
    ID             [a-zA-Z][a-zA-Z0-9]
    ```

    These definitions can be used in rules section as

    {DIGIT}+ {….

  - stuff to be copied verbatim into the flex output (e.g., declarations, **#include**s):
    - enclosed in %{ … }%, or
    - indented

Rules Section

- The *rules* portion of the input contains a sequence of rules.
- Each rule has the form

  patterns   *actions*

  where:

  - *Patterns are* regular expression which describes a pattern to be matched on the input
  - *pattern* must be un-indented
  - *actions* are either a single C command or a sequence enclosed in braces. It must begin on the same line of patterns.

Count no.of chars and lines

```
%{
int charcount=0,linecount=0;
%}
%%
. charcount++;
\n {linecount++; charcount++;}
%%
int main()
{
yylex();
printf("There were %d characters in %d lines\n",
charcount,linecount);
return 0;
}
```

Count no.of chars, words and lines

```
%{
int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]
%%
{letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

Patterns

- Essentially, extended regular expressions.
  - Syntax: similar to grep (see man page)

Metacharacters

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal "`a+b`" (C escapes still work)
`[]`	character class

Pattern matching: Examples

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc abcc abccc ...</td></tr>
<tr><td>a(bc)+</td><td>abc abcbc abcbcbc ...</td></tr>
<tr><td>a(bc)?</td><td>a abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a-z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: -, a, z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>one of: a, ^, b</td></tr>
<tr><td>[a|b]</td><td>one of: a, |, b</td></tr>
<tr><td>a|b</td><td>one of: a, b</td></tr>
</table>

Operators

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- If they are to be used as text characters, an escape should be used

  `\$` = "`$`"

  `\\` = "`\`"

- Every character except *blank*, *tab* (`\t`), *newline* (`\n`) and the list above is always a text character

Precedence of Operators

- Level of precedence
  - Kleene closure (*), ?, +
  - concatenation
  - alternation (|)
- All operators are left associative.
- Ex: a*b|cd* = ((a*)b)|(c(d*))

Regular Expression

x	match the character 'x'
.	any character (byte) except newline
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ]	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z]	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n]	any character EXCEPT an uppercase letter or a newline

Regular Expression

r*	zero or more r's, where r is any regular expression
r+	one or more r's
r?	zero or one r's (that is, "an optional r")
r{2,5}	anywhere from two to five r's
r{2,}	two or more r's
r{4}	exactly 4 r's
{name}	the expansion of the "name" definition (see above)
"[xyz]\"foo“	the literal string: [xyz]"foo
\X	if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'X' (used to escape operators such as '*')

Regular Expression

\0	a NUL character (ASCII code 0)
\123	the character with octal value 123
\x2a	the character with hexadecimal value 2a
(r)	match an r; parentheses are used to override precedence (see below)
rs	the regular expression r followed by the regular expression s; called "concatenation"
r|s	either an r or an s
^r	an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
r$	an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Two Notes on Using Lex

1. Lex matches token with longest match

Input: *abc*
Rule: `[a-z]+`

→ Token: *abc* (not "*a*" or "*ab*")

2. Lex uses the first applicable rule

Input: *post*

```
Rule1: "post"          {printf ("Hello,"); }
Rule2: [a-zA-Z]+       {printf ("World!"); }
```

→ It will print Hello, (not "World!")

Features

- Some limitations, Lex cannot be used to recognize nested structures such as parentheses, since it only has states and transitions between states.

  - Echo is an action and predefined macro in lex that writes code matched by the pattern.

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Features (cont)

- Text enclosed by %{ and %} is assumed to be C code and is copied verbatim
- Line which begins with white space is assumed to be a comment and is ignored
- Other lines are assumed to be definitions
- All input characters which are not matched by a lex rule are copied to the output stream (the file lex.yy.c which contains function yylex)
- Definitions from the definitions section are physically substituted into the rules

Example

A flex program to read a file of (positive) integers and compute the average:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt    [0-9]
%%
{dgt}+   return atoi(yytext);
%%
void main()
{
   int val, total = 0, n = 0;
   while ( (val = yylex()) > 0 ) {
      total += val;
      n++;
   }
   if (n > 0) printf("ave = %d\n", total/n);
}
```

Example

A flex program to read a file of (positive) integers and compute the average:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt    [0-9]
%%
```

rules

```
{dgt}+   return atoi(yytext);
%%
```

user code

```
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n",
      total/n);
}
```

Definition for a digit
(could have used builtin definition [:digit:] instead)

Rule to match a number and return its value to the calling routine

Driver code
(could instead have been in a separate file)

Example

A flex program to read a file of (positive) integers and compute the average:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt  [0-9]
%%
```

rules

```
{dgt}+  return atoi(yytext);
%%
```

user code

```
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf(“ave = %d\n”, total/n);
}
```

defining and using a name

Example

A flex program to read a file of (positive) integers and compute the average:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
%%
{dgt}+ return atoi(yytext);
%%
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

rules

user code

defining and using a name

char * yytext;

a buffer that holds the input characters that actually match the pattern

Example

A flex program to read a file of (positive) integers and compute the average:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
%%
```

rules

```
{dgt}+  return atoi(yytext);
%%
```

user code

```
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

defining and using a name

char * yytext;

a buffer that holds the input characters that actually match the pattern

Invoking the scanner: **yylex()**

Each time yylex() is called, the scanner continues processing the input from where it last left off. Returns 0 on end-of-file.

Matching the Input

- When more than one pattern can match the input, the scanner behaves as follows:
  - the longest match is chosen;
  - if multiple rules match, the rule listed first in the flex input file is chosen;
  - if no rule matches, the default is to copy the next character to **stdout**.
- The text that matched (the “token”) is copied to a buffer **yytext**.

Matching the Input (cont’d)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

```
#include <stdio.h>  /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf(“Error!\n”);  /* no arguments */
  }
  printf(“%d args given\n”, argc);
  return 0;
}
```

Matching the Input (cont’d)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

longest match:

```
#include <stdio.h>  /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf(“Error!\n”);  /* no arguments */
  }
  printf(“%d args given\n”, argc);
  return 0;
}
```

Matching the Input (cont’d)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

```
#include <stdio.h>  /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf(“Error!\n”);  /* no arguments */
  }
  printf(“%d args given\n”, argc);
  return 0;
}
```

longest match: Matched text shown in blue

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - the default output of default main() is **stdout**.

- E.g.

```
[a-z]+          printf("%s", yytext);
[a-z]+          ECHO;
[a-zA-Z]+       {words++; chars += yyleng;}
```

Lex Library Routines

- yylex()
  - The default main() contains a call of yylex(), a function of lex.yy.c file generated after using command lex
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

Review of Lex Predefined Variables

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

To count no of Identifiers

```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*        count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

- White space must separate the defining term and the associated expression.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "%{" and "%}" markers.
- substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.

User Subroutines Section

- You can use your Lex routines in the same ways you use routines in other programming languages.

```
%{
  void foo();
%}
letter      [a-zA-Z]
%%
{letter}+   foo();
%%
…
void foo() {
  …
}
```

User Subroutines Section (cont'd)

- The section where main() is placed

```
%{
  int counter = 0;
%}
letter    [a-zA-Z]

%%
{letter}+ {printf("a word\n"); counter+
  +;}

%%
main() {
  yylex();
```

Usage

- To run Lex on a source file, type
```
lex scanner.l
```
- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type
```
cc lex.yy.c -ll
```
- To run the lexical analyzer program, type
```
./a.out < inputfile
```