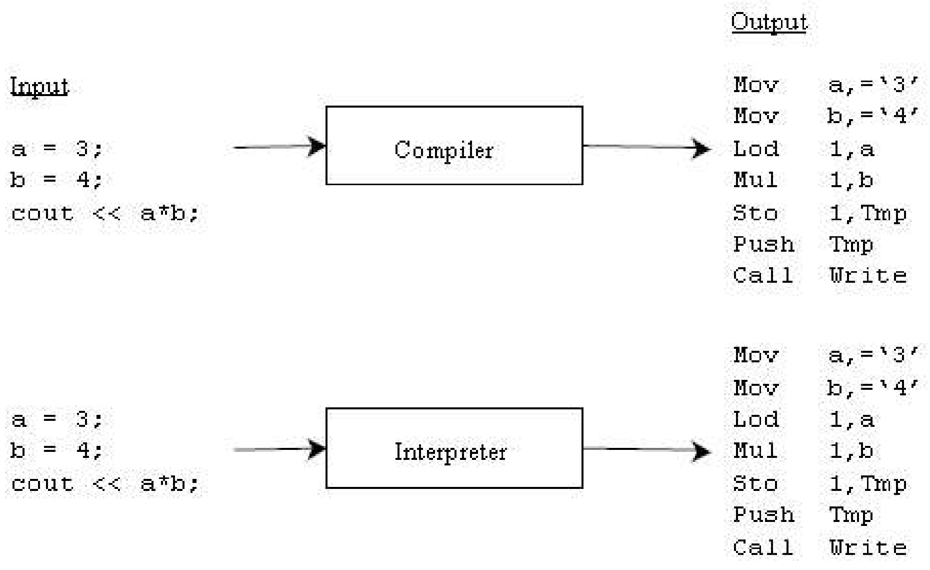
What is the difference between a single pass and multipass compiler?

Single-Pass Compiler	**Multi-pass Compiler**
A one-pass compiler is a compiler that passes through the source code of each compilation unit only once.	A multi-pass compiler is a type of compiler that processes the source code of a program several times.
A one-pass compiler does not "look back" at code it previously processed.	Each pass takes the result of the previous pass as the input, and creates an intermediate output.
It is also called narrow compiler.	It is sometimes called wide compiler.
While one-pass compilers may be faster than multi-pass compilers.	The wider scope thus available to these compilers allows better code generation.
Unable to generate as efficient programs, due to the limited scope available.	Some languages cannot be compiled in a single pass, as a result of their design.
The ability to compile in a single pass is often seen as a benefit because it simplifies the job of writing a compiler and one pass compilers are generally faster than multi-pass compilers.	In this way, the (intermediate) code is improved pass by pass, until the final pass emits the final code.
Pascal's compiler is an example of single-pass compiler.	C++ compiler is multi-pass compiler.

Ex:

➢ The difference between Compiler and Interpreter

The interpreter actually carries out the computations specified in the source program. In other words, the output of a compiler is a program, whereas the output of an interpreter is the source program's output.



```
a = 3;
b = 4;
cout << a*b;
```

```
Mov   a,='3'
Mov   b,='4'
Lod   1,a
Mul   1,b
Sto   1,Tmp
Push  Tmp
Call  Write
```

```
a = 3;
b = 4;
cout << a*b;
```

```
Mov   a,='3'
Mov   b,='4'
Lod   1,a
Mul   1,b
Sto   1,Tmp
Push  Tmp
Call  Write
```

➢ Optimization

For example, in the following program segment:

```
stmt1
go to label1
stmt2
stmt3
label2: stmt4
```

stmt2 and stmt3 can never be executed. They are unreachable and can be eliminated from the object program. A second example of optimization is shown below:

```
for (i=1; i<=100000; i++)
{ x = sqrt (y); // square root function
cout << x+i << endl;
}
```

In this case, the assignment to x need not be inside the loop since y doesn't change as the loop repeats (it is a **loop invariant**). In the global optimization phase, the compiler would move the assignment to x out of the loop in the object program:

```
x = sqrt (y); // loop invariant
for (i=1; i<=100000; i++)
cout << x+i << endl;
```

This would eliminate 99,999 unnecessary calls to the sqrt function at run time.

<u>2. Lexical Analysis Phase</u>

The first phase of a compiler is called lexical analysis. Because this phase scans the input string without backtracking (i.e. by reading each symbol once, and processing it correctly), it is often called a **lexical scanner**. As implied by its name, lexical analysis attempts to isolate the **"words"** in an input string. We use the word "word" in a technical sense. A word, also known as a **lexeme**, a lexical item, or a lexical token, is a string of input characters which is taken as a unit and passed on to the next phase of compilation. The Fig. 5 shows the role of lexical analysis.

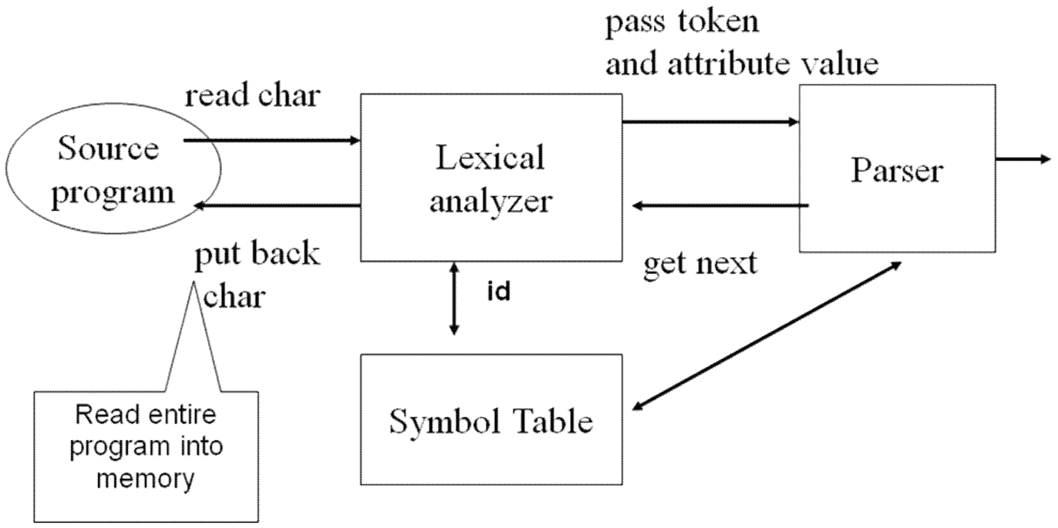


Fig. 5: The Role of a Lexical Analyzer

Examples of words are:

(1) Keywords - while, if, else, for ...etc. These are words which may have a particular predefined meaning to the compiler, as opposed to identifiers which have no particular meaning. Reserved words are keywords which are not available

to the programmer for use as identifiers. In most programming languages, such as Java and C, all keywords are reserved. PL/1 is an example of a language which has no reserved words.
(2) Identifiers - words that the programmer constructs to attach a name to a construct, usually having some indication as to the purpose or intent of the construct. Identifiers may be used to identify variables, classes, constants, functions, etc.
(3) Operators - symbols used for arithmetic, character, or logical operations, such as +, - , =,! =, etc. Notice that operators may consist of more than one character.
(4) Numeric constants - numbers such as 124, 12.35, 0.09E-23, etc. These must be converted to a numeric format so that they can be used in arithmetic operations, because the compiler initially sees all input as a string of characters. Numeric constants may be stored in a table.
(5) Character constants - single characters or strings of characters enclosed in quotes.
(6) Special characters - characters used as delimiters such as .,(,),{,},;. These are generally single-character words.
(7) Comments - Though comments must be detected in the lexical analysis phase, they are not put out as tokens to the next phase of compilation.
(8) White space - Spaces and tabs are generally ignored by the compiler, except to serve as delimiters in most languages, and are not put out as tokens.
(9) Newline - In languages with free format, newline characters should also be ignored, otherwise a newline token should be put out by the lexical scanner.

An example of C++ source input, showing the word boundaries and types is given below:

```
while ( x33 <= 2.5e+33 - total ) calc ( x33 ) ; // !
  1   6  2  3     4    3   2   6  2   6  2  6 6
```

During lexical analysis, a symbol table is constructed as identifiers are encountered. This is a **data structure** which stores each identifier once; regardless of the number of times it occurs in the source program. It also stores information about the identifier, such as the kind of identifier and where associated run-time information (such as the value assigned to a variable) is stored. This data structure is often organized as a **binary search tree**, or **hash table**, for efficiency in searching.

When **compiling block** structured languages such as Java, C, or Algol, the symbol table processing is more involved. Since the **same identifier** can have different declarations in different blocks or procedures, both instances of the identifier must be recorded. This can be done by **setting up a separate symbol table for each block**, or by specifying block scopes in a single symbol table. This would be done during the **parse or syntax analysis phase** of the compiler; the scanner could simply **store the identifier in a string space array** and **return a pointer** to its first character.
Numeric constants must be converted to an appropriate internal form. For example, the constant **"3.4e+6"** should be thought of as a string of six characters which needs to be translated to floating point (or fixed point integer) format so that the **computer can perform appropriate arithmetic operations with it**. As we will see, this is not a trivial problem, and most compiler writers make use of **library routines to handle this**.

The output of this phase is a **stream of tokens**, one token for each word encountered in the input program. Each **token consists of two parts**: **(1)** a **class** indicating which **kind of token** and **(2)** a **value** indicating which member of the class. The above example might produce the following **stream of tokens**:

Token Class	**Token Value**
1	[code for while]
6	[code for ( ]
2	[ptr to symbol table entry for x33]
3	[code for <=]
4	[ptr to constant table entry for 2.5e+33
3	[code for -]
2	[ptr to symbol table entry for total]
6	[code for)]
2	[ptr to symbol table entry for calc]
6	[code for ( ]
2	[ptr to symbol table entry for x33]
6	[code for)]
6	[code for ;]

Note that the comment is not put out. Also, some token classes might not have a value part. For example, a left parenthesis might be a token class, with no need to specify a value.

Some variations on this scheme are certainly possible, allowing greater efficiency. For example, **when an identifier is followed by an assignment operator**, a single assignment token could be put out. The value part of the token would be a symbol table pointer for the identifier. Thus the input string "x =", would be put out as a single token, rather than two tokens. Also, each keyword could be a distinct token class, which would increase the number of classes significantly, but might simplify the syntax analysis phase.
Note that the lexical analysis phase does not check for proper syntax. The input could be

} while if ({

and the lexical phase would put out five tokens corresponding to the five words in the input. (Presumably the errors will be **detected in the syntax analysis phase**). If the source language is **not case sensitive**, the scanner must accommodate this feature. For example, the following would all represent the same keyword: then, tHeN, Then, THEN. A preprocessor could be used to translate all alphabetic characters to upper (or lower) case.

<u>Example:</u>

Identify the lexemes that make up the tokens in the following programs. Give reasonable attribute values for the tokens (using the table below):

Tokens	**Tokens type**	**Token Value**

```
void swap(int i, int j)
{
 int t;
 t=i;
 i=j;
 j=t;
}
```

Tokens	Tokens type	Token Value
void	keyword	-
swap	id	Pointer of swap in the entry table
(	Opened pranth	-
int	Int-keyword	-
i	id	Pointer of i in the entry table
,	comma	-
int	Int-keyword	-
j	id	Pointer of j in the entry table
)	Closed pranth	-
{	Opened curly	-
int	Int-keyword	-
t	id	Pointer of t in the entry table
;	semicolon	-
t	id	Pointer of t in the entry table
=	Equal	-
i	id	Pointer of i in the entry table
;	semicolon	-
i	id	Pointer of i in the entry table
=	Equal	-
j	id	Pointer of j in the entry table
;	semicolon	-
j	id	Pointer of j in the entry table
=	Equal	-
t	id	Pointer of t in the entry table
;	semicolon	-
}	Closed curly	-

- **Pattern: A rule that describes a set of strings**
- **Token: A set of strings in the same pattern**
- **Lexeme: The sequence of characters of a token**

Token	Sample Lexemes	Pattern
if	if	if
id	abc, n, count,…	letters+digit
NUMBER	3.14, 1000	numerical constant

Tokens true or false
fi(a==f(x))…

<u>Exercise: Divide the following C + + program:</u>
1.
float limitedSquare(x) float x {

/* returns x-squared, but never more than 100 */

return (x<=-10.0 || x>=10.0)?100:x*x;
}

into appropriate lexemes, which lexemes should get associated lexical values? What should those values be?
2.
Show the lexical tokens corresponding to each of the following C/C++ source inputs:
(a) **for (I=1; I<5.1e3; I++) func1(X);**
(b) **if (Sum!=133) /* Sum = 133 */**
(c)) while (1.3e-2 if &&
(d) **if 1.2.3 < 6**
(e) **sum = sum + unit ∗/∗accumulate sum ∗/ 1.2e-12 ;**

2.1 Specification of Tokens

2.1.1 Formal Languages

A formal language is one that can be specified precisely and is amenable for use with computers, whereas a natural language is one which is normally spoken by people.

2.1.1.1 Language Elements

We need to understand some fundamental definitions from discrete mathematics.

➢ A set is a collection of **unique** objects. In listing the elements of a set, we normally list each element only once, and the elements may be listed in any order. A **set** may **contain an infinite number of objects**. The set which contains **no elements** is still a set, and we call it the **empty set** and designate it either by **{ }** or by **φ**.

➢ A string **is a list of characters from a given alphabet**. The elements of a string need **not be unique**, and the order in which they are listed is important. For example, **"abc"** and **"cba"** are **different strings**, as are **"abb"** and **"ab"**. The string which consists of no characters is still a string (of characters from the given alphabet), and we call it the **null string** and designate it by **ε**. It is important to remember that if, for example, we are speaking of strings of zeros and ones (i.e. strings from the alphabet **{0, 1}**).
The following are examples of languages from the alphabet **{0, 1}**:

1. {0,10,1011}
2. {}
3. {ε,0,00,000,0000,00000,...}
4. The set of all strings of **zeroes** and ones having an even number of ones.

The first **two examples are finite sets** while the **last two examples are infinite**. The first two examples do not contain the null string, while the last two examples do.

2.1.1.2 Finite State Machines

We now encounter a problem in **specifying, precisely, the strings in an infinite** (or very large) language. If we describe the language in English, we lack the precision necessary to make it clear exactly which strings are in the language and which are not in the language. **One solution** to this problem is to use a **mathematical** or **hypothetical machine** called a **finite state machine**. This is a machine which we will describe in mathematical terms and whose operation should be perfectly clear, though we will not actually construct such a machine. The study of theoretical machines such as the finite state machine is called automata theory because “automaton” is just another word for “machine”. A finite state machine consists of:

1. A **finite set of states**, one of which is designated the starting state, and zero or more of which are designated accepting states. The starting state may also be an accepting state.

2. A **state transition** function which has two arguments – a **state** and an **input** symbol (from a given input alphabet) – and returns as result a state.

Here is how the machine works. The **input** is a string of symbols from the input alphabet. The machine is initially in the **starting state**. As each symbol is read from the input string, the machine proceeds to a new state as indicated by the transition function, which is a function of the input symbol and the current state of the machine. **When the entire input string has been read**, the machine is either in an **accepting** state or in a non-accepting state. If it is in an accepting state, then we say the input string has been accepted. Otherwise the input string has not been accepted, i.e. it has been rejected. The set of all input strings which would be accepted by the machine form a language, and in this way the finite state machine provides a precise specification of a language.

A string is a **finite sequence** of symbols, such as 0011. Sequence and word are synonyms for string. The length of the string x, usually denoted $| x |$ is the total number of the symbols in x. For example 01101 is a string of length 5. A special string in the empty string, which denote by ε. This string is of length zero.
If x and y are string, then the concatenation of x and y; written x.y or xy, is the string formed by the symbol of x followed by the symbol of y. For example if **x=abc** and **y=de**, where **a, b, c, d, e** are symbols, then xy=abcde. The condition of the empty string with any string is that string, more formally $\varepsilon x = x\varepsilon = x$.

We may think of condition as a "product". It thus makes sense to talk of exponentiation of string as representing an iterated project. For example, $x^1=x$, $x^2=xx$, $x^3=xxx$ and so on. In general, x^i is the string x repeated i times. As a useful convention, we take x^0 to be ε for any string x. Thus, ε, is the identity of concatenation.