SYSTEM SOFTWARE and COMPILERS- 18CS61

Prof. Sampada K S, Assistant Professor
DEPT. OF CSE | RNSIT

MODULE-2

- Introduction: Language Processors, The structure of a compiler, The evaluation of programming languages, The science of building compiler, Applications of compiler technology.
- Lexical Analysis: The role of lexical analyzer, Input buffering, Specifications of token, recognition of tokens.

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

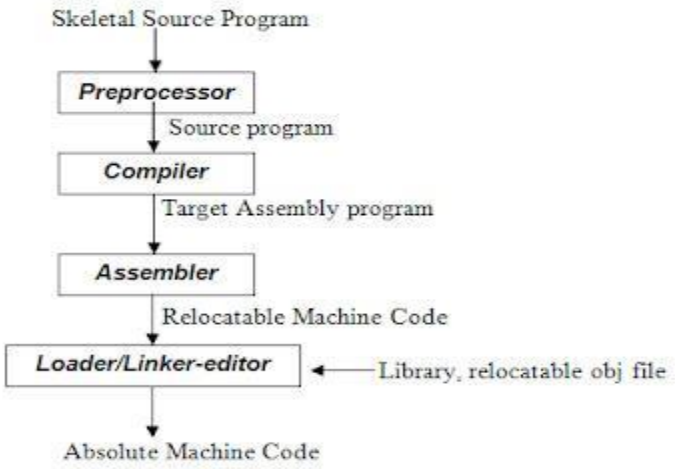
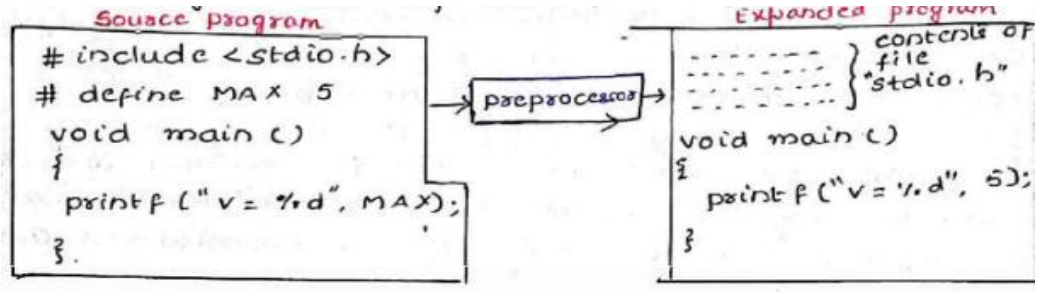


Fig 1.1 Language –processing System

Preprocessor

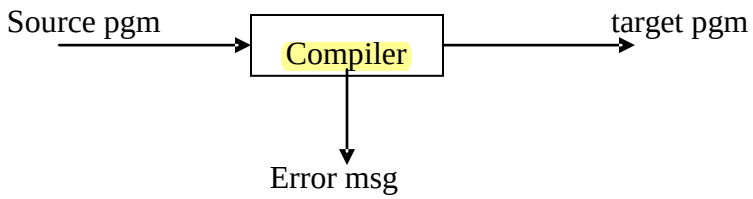
A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

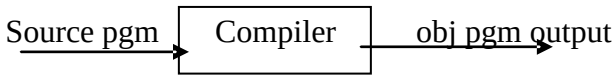


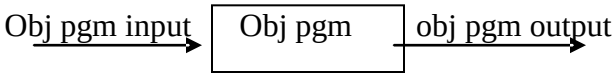
COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

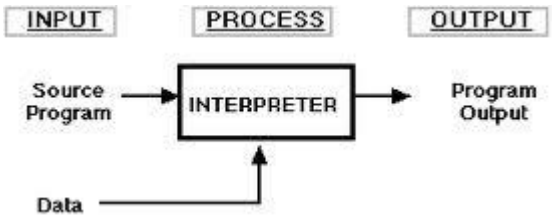




ASSEMBLER

programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER: An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA

also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for Interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is *slower*.
- *Memory* consumption is more

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

Compiler	Interpreter
Compiler scans the whole program in one go.	Translates program one statement at a time.
As it scans the code in one go, the errors (if any) are shown at the end together.	Considering it scans code one line at a time, errors are shown line by line.
Main advantage of compilers is it's execution time.	Due to interpreters being slow in executing the object code, it is preferred less.
It converts the source code into object code.	It does not convert source code into object code instead it scans it line by line
It does not require source code for later execution.	It requires source code for later execution.
C, C++, C# etc.	Python, Ruby, Perl, SNOBOL, MATLAB, etc.

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problem of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs or they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the hll program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

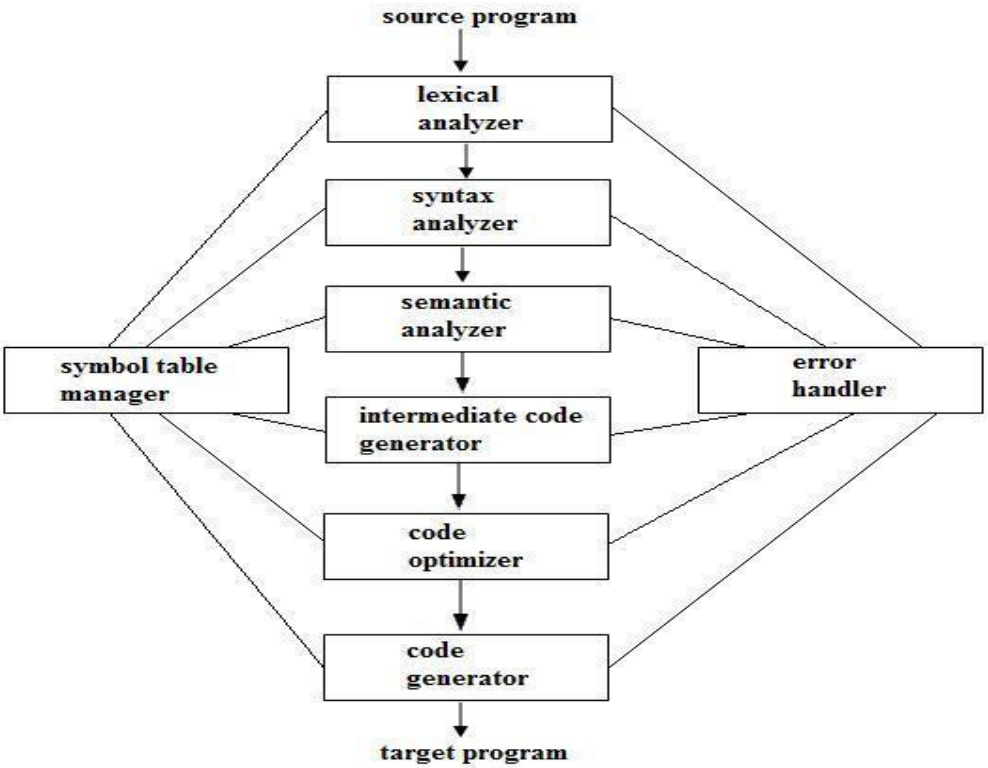


Fig 1.5 Phases of a compiler

Lexical Analysis:- Lexical Analyzer or *Scanners* reads the source program one character at a time, On reading character stream of source program, it groups them into meaningful sequences called "*Lexemes*". For each lexeme analyzer produces an output called **tokens.**

<token_name, attribute value>

Abstract symbol used in parser

points to entry in the symbol table for this token

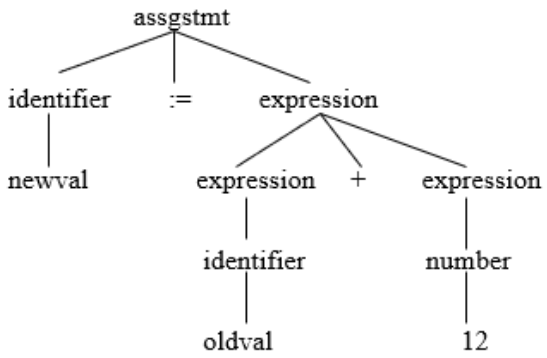
A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)

Ex newval := oldval + 12 => tokens:

newval	*identifier*
:=	*assignment operator*
oldval	*identifier*
+	*add operator*
12	*a number*

Syntax Analysis:-The second stage of translation is called Syntax analysis or *parsing*. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program.
A syntax analyzer is also called as a parser. A parse tree describes a syntactic structure.

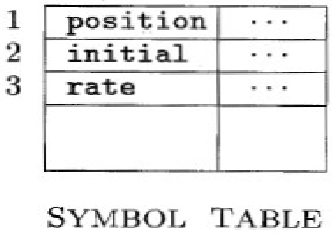


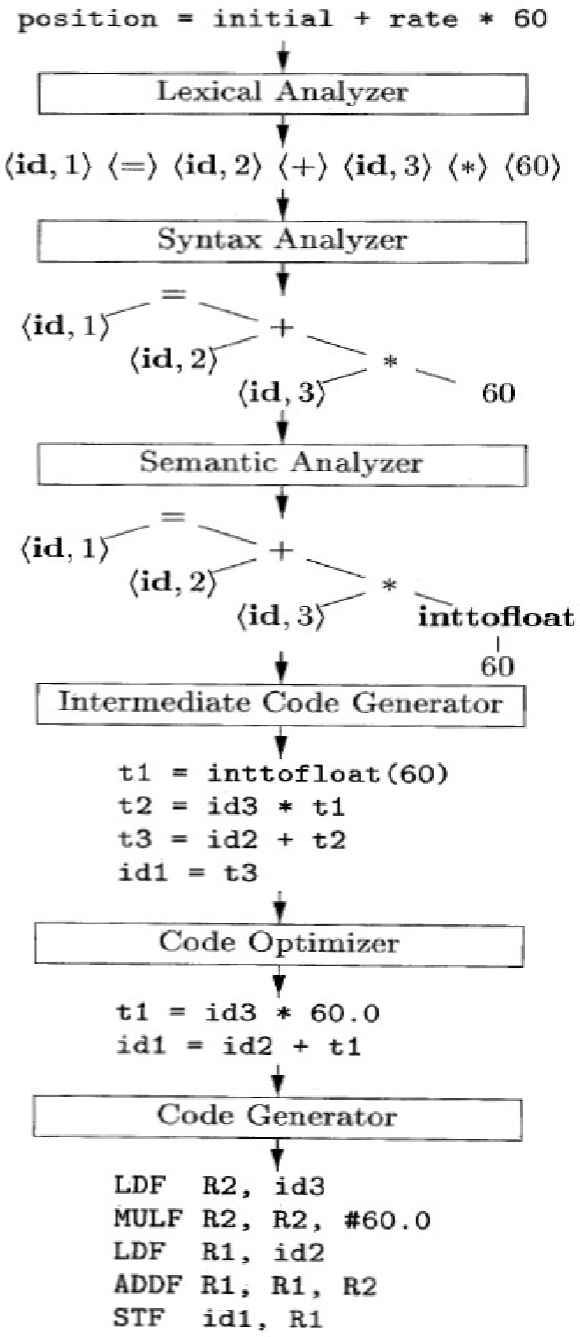
Semantic Analysis: Uses syntax tree and information in symbol table to check source program for semantic consistency with language definition. It gathers type information and saves it in either syntax tree or symbol table for use in Intermediate code generation.
Type checking- compiler checks whether each operator has the matching operands.
Coercions-language specification may permit some type of conversion.

Intermediate Code Generations:- An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.





The Evolution of Programming Languages

The move to higher-level languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

Impacts on compilers

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

The Science of Building a Compiler

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled.

Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modeling in compiler design and implementation

The study of compilers is mainly study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

The science of code optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

Compiler construction tools

Some of the tools that help the programmer to build the compiler very easily and efficiently are listed below:

Parser generators: They accept grammatical description of a programming language and produce syntax analyzers.

Scanner generators: They accept regular expression description of the tokens of a language and produce lexical analyzers.

Syntax-directed translation engines: They produce various routines for walking a parse tree and generating the intermediate code.

Code generator generators: Using the intermediate code generated, they produce the target language. The target language may be object program or assembly language program.

Data flow analysis engines: It gathers the information of how values are transmitted from one part of a program to other part of the program. Data flow analysis is key part of the code optimization

Compiler construction toolkits: They provide a set of routines for constructing various phases of the compiler.

Applications of Compiler Technology

Implementation of high-level programming languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Optimizations for computer architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: parallelism and memory hierarchies. Parallelism can be found at several levels: at the instruction level, where multiple operations are executed simultaneously and at the processor level, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism-All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Memory Hierarchies- A memory hierarchy consists of several levels of storage with different

speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Design of new computer architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

Program translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages.

Software productivity tools

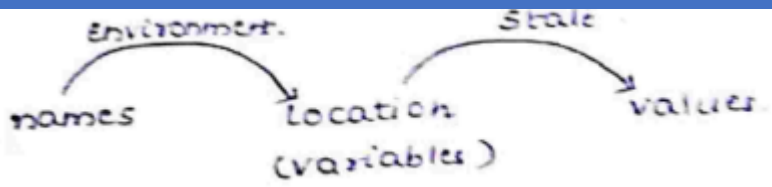
Programs are arguably the most complicated engineering artifacts ever produced; they consist of many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs. An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Dataflow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data- flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

Programming Language Basics

The static/dynamic distinction among the most important issues that we face when designing a compiler for a language is what decisions the compiler can make about a program. If a compiler language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.

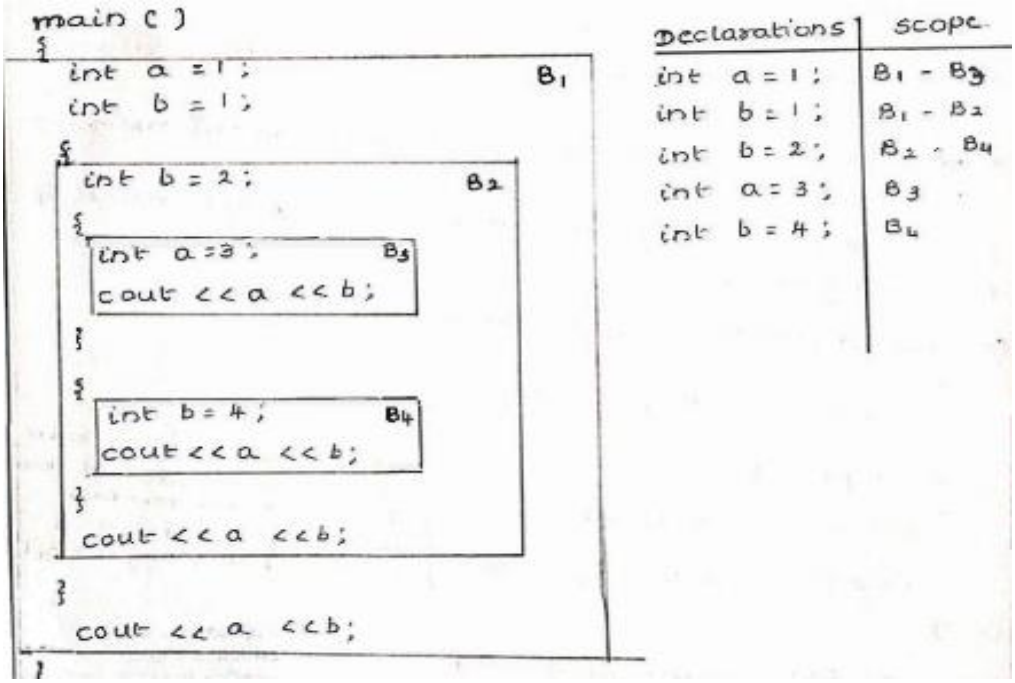
Environments and states

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as x = y+1 changes the value denoted by the name x. More specifically, the assignment changes the value in whatever location is denoted by x.



Static scope and block structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C + + , Java, and C # , also provide explicit control over scopes through the use of keywords like public, private, and protected. In this section we consider static-scope rules for a language with blocks, where a block is a grouping of declarations and statements. C uses braces {and} to delimit a block; the alternative use of b e g in and end for the same purpose dates back to Algol.



Explicit access control

Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition. In analogy with block structure, the scope of a member declaration x in a class C extends to any subclass C, except if C has a local declaration of the same name x.

Dynamic scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations.

Parameter passing mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this we consider how the actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by- reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

* call by value :-
- The actual parameter is evaluated [if an expression or copied (if a variable), the value is placed in the location belonging to corresponding formal parameter of called procedure.
- It has all computations involving formal parameters done by the called procedure is local to that procedure.

E.g.:
```
int add (int a, int b)
{
  return (a+b);
}
main ()
{
  :
  c = add (10, 20);
  :
}
```

* call by reference:
- The address of actual parameter is passed to the callee as the value of the corresponding formal parameter.
- uses of formal parameter in the code of the callee are implemented by following this pointer to location indicated by caller.
- changes to the formal parameter thus appears as changes in actual parameters.
- If actual parameter is expression, it is evaluate before the call and its value stored in a location of its own. changes to formal parameter change value in this location, but can have no effect on data of caller.

```
int add ( int *a, int *b)
{
  return (a+b);
}
main ()
{
  int p=10; q=20;
  c = add (&p, &q);
}
```

Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

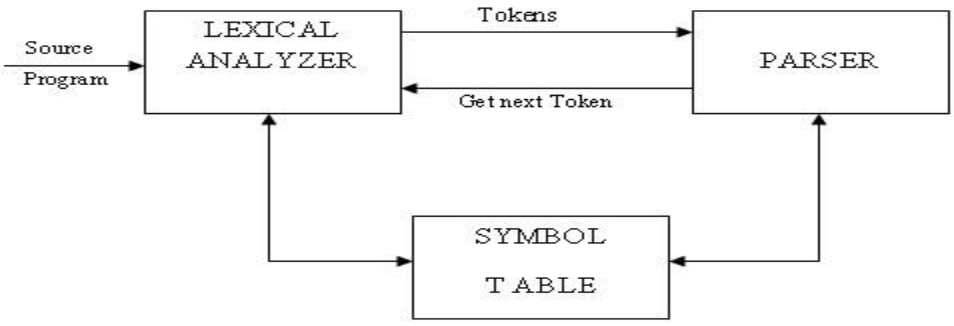
LEXICAL ANALYSIS

OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	**lexeme**	**pattern**
const	const	const
if	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognize a *lexeme* as a valid *token* for the lexer.

These errors are detected during the lexical analysis phase. Typical lexical errors are:

- Exceeding length of identifier or numeric constants.
- The appearance of illegal characters
- Unmatched string

Example 1 : **printf("Geeksforgeeks");$**

This is a lexical error since an illegal character $ appears at the end of statement.

Example 2 : **This is a comment */**

This is an lexical error since end of comment is present but beginning is not present

Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognized valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error recovery for lexical errors:

Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated

set of synchronizing tokens is found. Synchronizing tokens are delimiters such as; or }

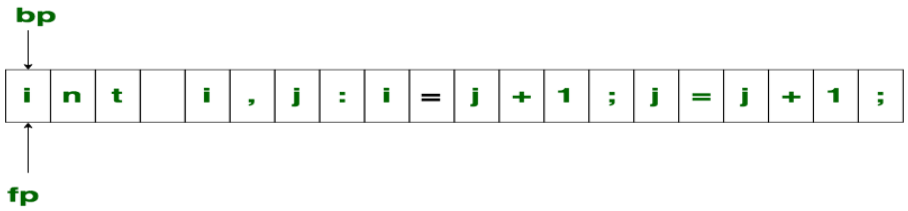
- The advantage is that it is easy to implement and guarantees not to go into an infinite loop
- The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

Error-recovery actions are:

i. Delete one character from the remaining input.
ii. Insert a missing character in to the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

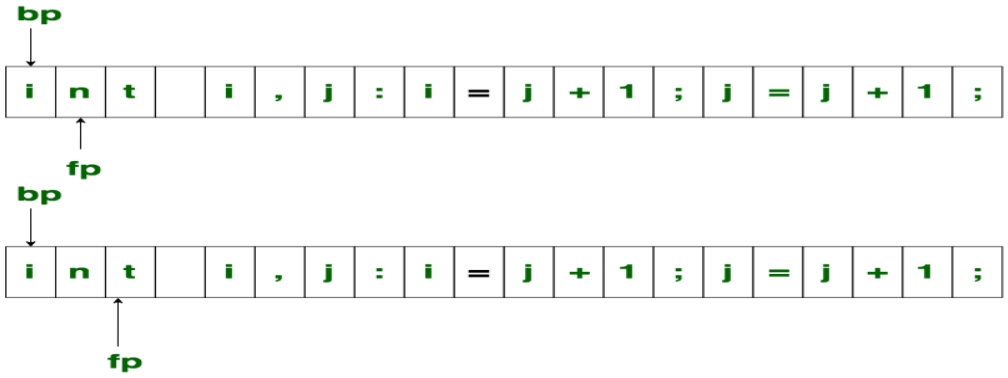
INPUT BUFFERING

Lexical Analysis has to access secondary memory each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers **lexeme begin** ptr(bp) and **forward** to keep track of the pointer of the input scanned. Initially both the pointers point to the first character of the input string as shown below



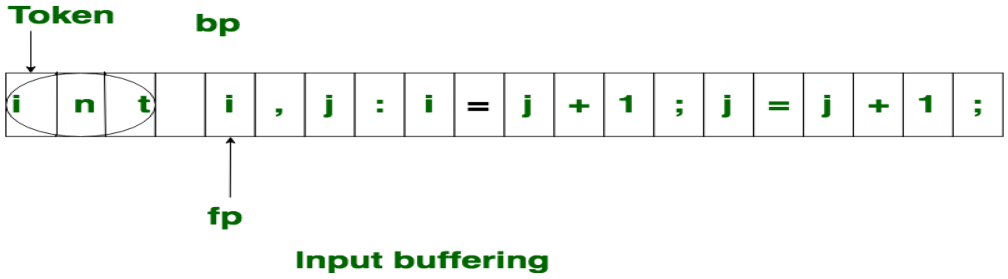
Initial Configuration

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified.



Input Buffering

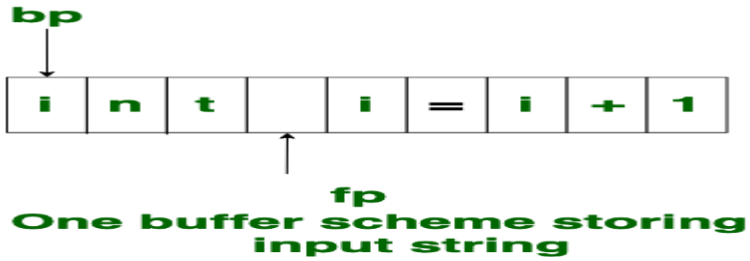
The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.
The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Input buffering

One Buffer Scheme:

In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.

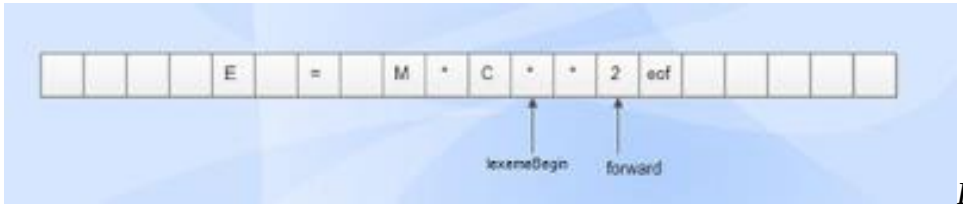


One buffer scheme storing input string

Two Buffer Scheme:

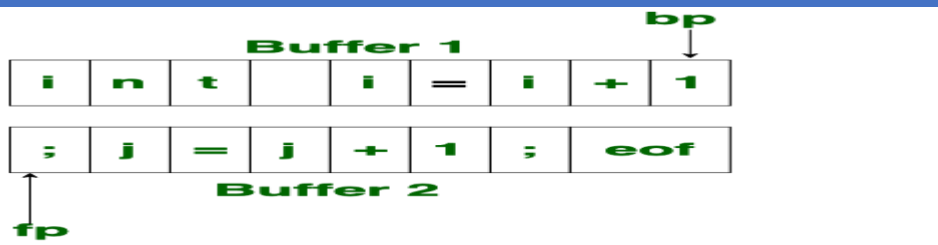
To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

A specialized buffering techniques0 used to reduce the amount of overhead, which is required to process an input character in moving characters.



Buffer pairs

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- Two pointers lexemeBegin and forward are maintained.
- Lexeme Begin points to the beginning of the current lexeme which is yet to be found.
- Forward scans ahead until a match for a pattern is found.
- Once a lexeme is found, lexeme begin is set to the character immediately after the lexeme which is just found and forward is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.



Two buffer scheme storing input string

```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else forward := forward + 1;
```

What is Sentinels ?

Sentinels is used to make a check, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

Therefore the ends of the buffer halves require two tests for each advance of the forward pointer. Test 1: For end of buffer. Test 2: To determine what character is read. The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

The sentinel is a special character that cannot be part of the source program. *(*eof character is used as sentinel).

Disadvantages

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

Advantages

- Most of the time, It performs only one test to see whether forward pointer points to an eof.
- Only when it reaches the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eofs, the average number of tests per input character is very close to 1.

Observe from the above algorithm that instead of having two tests as in buffer pair technique, there is only one test i.e., testing the eof marker.

```
Switch (*forward++) {
  case eof:
        if (forward is at end of first buffer) {
              reload second buffer;
              forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer) {
              reload first buffer;\
              forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input *
              terminate lexical analysis;
        break;
  cases for the other characters;
}
```

SPECIFICATION OF TOKENS

1. In theory of compilation regular expressions are used to formalize the specification of tokens
2. Regular expressions are means for specifying regular languages
3. Example: *Letter_(letter_ | digit)**
4. Each regular expression is a pattern specifying the form of strings

REGULAR EXPRESSIONS

1. Ɛ is a regular expression, L(Ɛ) = {Ɛ}
2. If a is a symbol in ∑then a is a regular expression, L(a) = {a}
3. (r) | (s) is a regular expression denoting the language L(r) L(s)
4. (r)(s) is a regular expression denoting the language L(r)L(s)
5. (r)* is a regular expression denoting (L(r))*
6. (r) is a regular expression denoting L(r)

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R2|R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-

expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each 'a' in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter	-	A | B | ……| Z | a | b |……| z|
Digits	-	0 | 1 | 2 | …. | 9
Id	-	letter (letter / digit)*

RECOGNITION OF TOKENS:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. The two functions that are used while designing the lexical analyzer are:

 retract(): This function is invoked only if we want to unread the last character read. It is identified by having a an edge labeled other to the final state with * marked to it. This function un reads the last character read

 install_ID(): Once the identifier is identified, the function install_ID() is called. This function checks whether the identifier is already there in the symbol table. If it is not there, it is entered into the symbol table and returns the pointer to that entry. If it is already there, it returns the pointer to that entry.

```
Stmt if expr then  stmt
     |  If expr then else stmt
     | e
```

```
Expr  term relop term
      | term
Term  id
      |number
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit  -->[0,9]
digits-->digit+
number   -->digit(.digit)?(e.[+-]?digits)?
letter-->[A-Z,a-z]
id     -->letter(letter/digit)*
if     --> if
then   -->then
else   -->else
relop  --> </>/<=/>=/==/< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws -->(blank/tab/newline)$^{+}$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

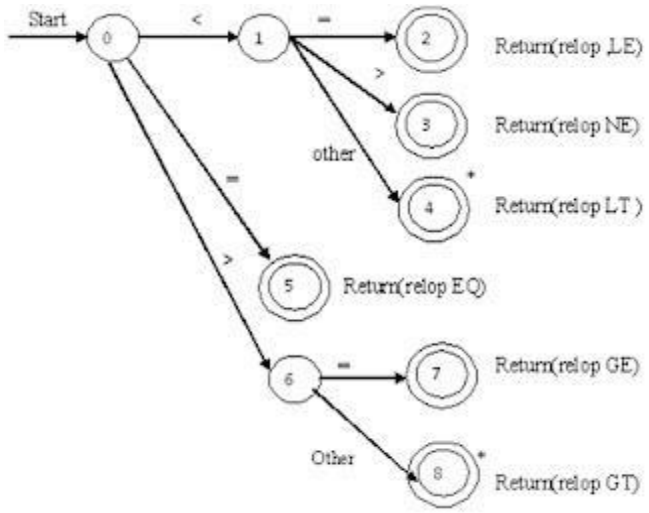
Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme

Begin and forward pointers we always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



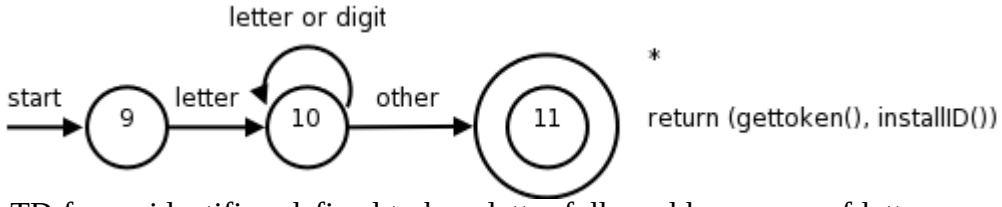
As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.

Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
        TOKEN retToken = new (RELOP)
        while (1)
        {/*repeat character processing until a return or failure occurs*/
        switch(state)
        {
          case 0: c= nextchar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail();  /* lexeme is not a relop */
                 break;
```

```
        case 1: …
        …
        case 8: retract();
                retToken.attribute = GT;
                return(retToken);
      }
```

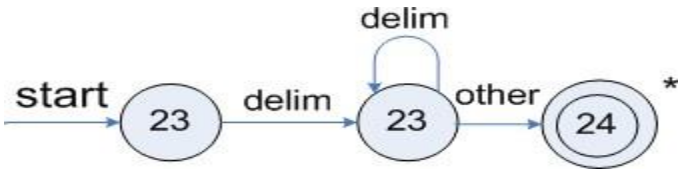
- **Transition diagrams for identifier**



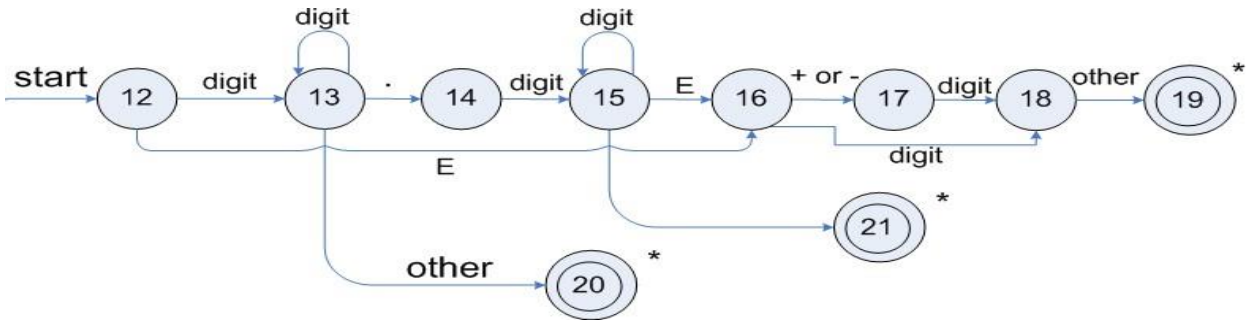
The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

```
state = 0;
for (;;)
{
switch (state)
{
     case 0:
          ch = getchar()
          if (ch == letter) state = 10;
          else state = 3; // Identify the next token
          break;
     case 1:
          ch = getchar()
          if (ch == letter or ch == digit) state = 11;
          else state = 2;
          break;
     case 2: retract(); // undo the last character read
          return ( ID, install_ID() )
     case 3: /* Identify the next token */
}
}
```

- **Transition diagram for whitespace**



Transition diagram for unsigned numbers



Review Question:

1) What is a language processor? What are the various types of language processors?
2) What is a compiler? What are the activities performed by the compiler?
3) What is an interpreter? What are the differences between a compiler and interpreter
4) What is an assembler? What are the activities performed by the compiler?
5) What is hybrid compiler?
6) Explain language processing system
7) Discuss the analysis of source program with respect to compilation process
8) What are the different phases of the compiler? or "Explain the block diagram of the compiler construction method
9) Show the output of each phase of the compiler for the assignment statement: sum = initial + value * 10
10) What is the difference between a phase and pass?
11) What are compiler construction tools?
12) What is lexical analysis? What is the role of lexical analyzer?
13) Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase?
14) What is the meaning of patterns, lexemes and tokens?
15) Identify lexemes and tokens in the following statement: a = b * d;
16) Identify lexemes and tokens in the following statement: printf("Simple Interest = %f\n", si);
17) What is the need for returning attributes for tokens along with token name?"
18) Give the token names and attribute values for the following statement: E = M * C **2. where ** in FORTRAN language is used as exponent operator.
19) Give the token names and attribute values for the following statement: si = p * t * r / 100
20) Why input buffering is required? What is input buffering?
21) What is the disadvantage of input buffering with buffer pairs? Explain the use of sentinels in recognizing the tokens.
22) Design a lexical analyzer to identify an identifier
23) Design a lexical analyzer to identify the relation operators such as <, <=, >, >=, == and !=
24) Design a lexical analyzer to identify the keywords begin, else, end. Sketch the program segment to implement it showing the first two states and one final state
25) Design a lexical analyzer to recognize unsigned number. Sketch the program segment to implement it showing the first two states and one final state

MODULE-4

Lex and Yacc –The Simplest Lex Program, Grammars, Parser-Lexer Communication, A YACC Parser, The Rules Section, Running LEX and YACC, LEX and Hand- Written Lexers, Using LEX - Regular Expression, Examples of Regular Expressions, A Word Counting Program

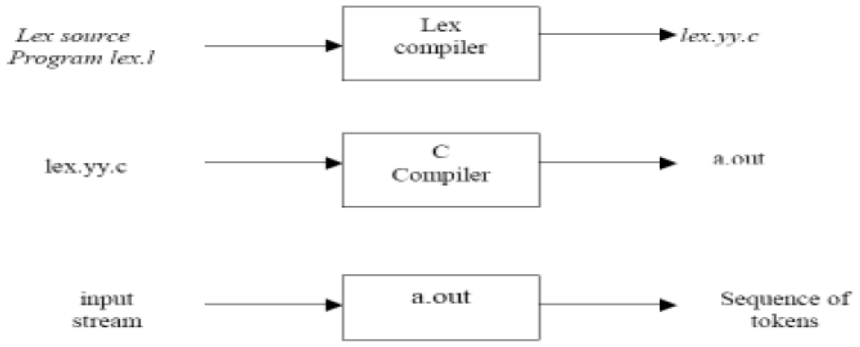
Using YACC – Grammars, Recursive Rules, Shift/Reduce Parsing, What YACC Cannot Parse, A YACC Parser - The Definition Section, The Rules Section, The LEXER, Compiling and Running a Simple Parser, Arithmetic Expressions and Ambiguity.

Lex is a tool for building lexical analyzers or lexers i.e. yylex(). A lexer takes an arbitrary inputstream and tokenizes it., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it can be "end product".

Lex and Yacc help us to write programs that transform structured input.Applications

- Simple text search program
- Compiler

When we write a lex specification, we create a set of patterns which lex matches against the input. Each time one of the patterns matches, the lex program invokes C Code that you provide which does something with the matched text. In this way a lex program divides the input into strings which we call **token**s. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called **yylex().** Your program calls yylex() to run the lexer.



Lex generates a deterministic finite automaton from the regular expressions in the source program. The automaton is interpreted, rather than compiled, to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input.

Structure of a Lex file

Lex file is divided into three sections, separated by lines that contain only two percent signs, as follows:

Definitions

%%

Rules

%%

User subroutines

red : required
blue : optional

- The ***definition*** section includes

declarations of variables, *int i=0, c=0;*

manifest constants (A manifest constant is an identifier that is declared to represent a constant) *# define PIE 3.14*),

regular definitions. **Definitions**: **NAME** <*Expression*>

Example: DIG [0-9]

The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

Literal Block: C code bracketed by lines *"%{"* and *"%}"*

- The ***rules*** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

p1 {action 1}
p2 {action 2}
p3 {action 3}
… …

where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

Choosing between different possible matches:

When more than one pattern can match the input, lex chooses as follows:

1. The longest match is preferred.

Longest Match Rule: When the lexical analyzer read the source-code, it scans the code letter by letter; and when a whitespace, operator symbol, or special symbols occurs, it decides that a word is completed.

2. Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

- The ***C code*** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

The Simplest Lex Program

This lex program copies its standard input to its standard output:

```
%%
.|\n   ECHO;
%%
```

It acts very much like the UNIX cat command run with no arguments.
Lex automatically generates the actual C program code needed to handle reading the input file and sometimes, as in this case, writing the output as well.

Recognizing Words with Lex

Let's build a simple program that recognizes different types of English words.

```
%{
/*
 * this sample demonstrates (very) simple recognition:
 * a verb/not a verb.
 */
%}
%%
[\t ]+                    /* ignore whitespace */ ;
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did  |
will |
would |
should |
can  |
could |
has  |
have |
had |
```

```
go        { printf("%s: is a verb\n", yytext); }
[a-zA-Z]+ { printf("%s: is not a verb\n", yytext); }

.|\n      { ECHO; /* normal default anyway */ }
%%

main()
{
      yylex() ;
}
```

LEX REGULAR EXPRESSIONS

A LEX regular expression is a word made of :

- text characters (letters of the alphabet, digits, ...)
- operators : " \ { } [ ] ^ $ < > ? . * + | () /

Regular expressions in lex are composed of *metacharacters* (Table 2-1). Pattern matching examples are shown in Table 2-2.
Moreover An operator can be used as a text character if it is precedes with the escape operator (\ backslash).
The quotation **(“ ”)** marks indicate that whatever is contained between a pair of quotes is to be taken as text characters. For instance

xyz"++" matches the string xyz++.

Table 2-1: Pattern Matching Primitives

Metacharacter	*Matches*
.	any character except newline
\n	newline
*****	zero or more copies of preceding expression
+	one or more copies of preceding expression
?	zero or one copy of preceeding expression
^	beginning of line as the first character of a regular expression
$	end of line as the last character of a regular expression
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal “a+b” (C escapes still work)
[]	character class matches any character specified using the operator pair [ ]. Within a character class, normal operators lose their meaning. The operators allowed in a character class are the hyphen (“-”), (“\”) and circumflex (“^”). (a) the escape character \ as above,

	(b) the hyphen character - the represents a range of characters (c) the circumflex character ^ as first character negates the expression [^abc] NOTabc
X{m,n}	Matches m to n repetitions of X

Table 2-2: Pattern Matching Examples

Expression	***Matches***
abc	abc
abc*	ab, abc, abcc, abccc, …
abc+	abc, abcc, abccc, …
a(bc)+	abc, abcbc, abcbcbc, …
a(bc)?	a, abc
[abc]	a, b, c
[a-z]	any letter, athrough z
[a\-z]	a, -, z
[-az]	-, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	a, ^, b
[a|b]	a, |, b
a|b	a or b

Lex variables and functions commonly used:

Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	Is called by lex when input is exhausted return 1 if done, 0 if not done
FILE *yyout	output file defaults to stdout..
FILE *yyin	input file defaults to stdin
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string
yylineno	Line number

Lex specification for decimal numbers

```
%%
[\n\t   ]   ;
  -?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)   {printf
  ("number\n");}
.    ECHO;
%%
main( )
{
 yylex( );
}
```

where \. denotes a literal period.

CONTEXT SENSITIVITY. LEX provides some support for contextual grammatical rules. *Left context* is handled by means of *start conditions.* **Start state**, a method of capturing context sensitive information within the lexer. Tagging rules with start states tells the lexer only to recognize the rules when the start state is in effect. It provides a mechanism for conditionally activating rules.

Used to activate rules conditionally.

- Any rule prefixed with will be activated only when the scanner is in start condition S.
- Declaring a start condition S:
- in the definition section: %x S – "%x" specifies "exclusive start conditions" – flex also supports "inclusive start conditions" ("%s"), see man pages.

Putting the scanner into start condition S:

action: BEGIN(S)

Example: [^"]* { …match string body… } – [^"] matches any character other than " –

The rule is activated only if the scanner is in the start condition STRING.

INITIAL refers to the original state where no start conditions are active. matches all start conditions. Start conditions let us explicitly simulate finite state machines.

This lets us get around the "longest match" problem for C style comments.

Define start conditions in the *definitions* section of the specification file by using a line in the following form: **`%Start  name1 name2`**

where name1 and name2 define names that represent conditions. There is no limit to the number of conditions, and they can appear in any order. You can also shorten the word Start to s or S.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in <> (less than, greater than) symbols at the beginning of the rule. The following

example defines a rule, expression, that the **lex** program recognizes only when the **lex** program is in start condition name1**: <name1> expression**

To put the **lex** program in a particular start condition, execute the action statement in the action part of a rule; for instance, BEGIN in the following line: **BEGIN name1;**

This statement changes the start condition to name1.

To resume the normal state, enter:

BEGIN 0;

Or

BEGIN INITIAL;

where INITIAL is defined to be 0 by the **lex** program. **BEGIN 0**; resets the **lex** program to its initial condition.

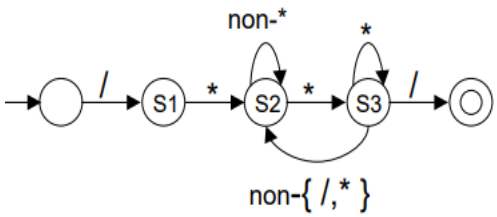
The **lex** program also supports exclusive start conditions specified with %**x** (percent sign, lowercase x) or %**X** (percent sign, uppercase X) operator followed by a list of exclusive start names in the same format as regular start conditions. Exclusive start conditions differ from regular start conditions in that rules that do not begin with a start condition are not active when the lexical analyzer is in an exclusive start state. For example:

```
%s      one
%x      two
%%
abc     {printf("matched ");ECHO;BEGIN one;}
<one>def        printf("matched ");ECHO;BEGIN two;}
<two>ghi        {printf("matched ");ECHO;BEGIN INITIAL;}
```

In start state one in the preceding example, both abc and def can be matched. In start state two, only ghi can be matched.

Example: Program to remove comment lines from c (using exclusive start)

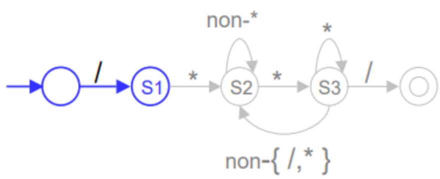
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;    /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*"         ;    /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

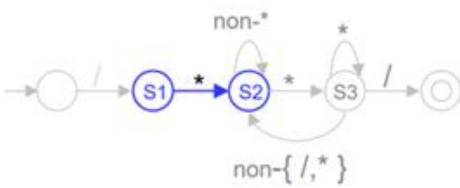
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;    /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;    /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

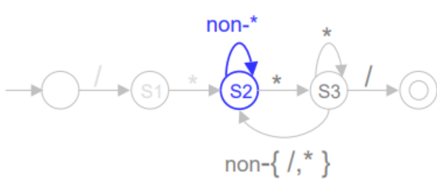
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;    /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;    /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

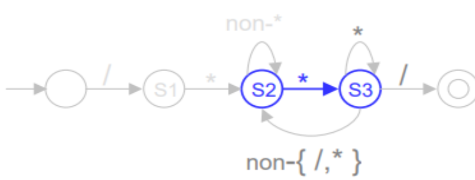
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;    /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;    /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

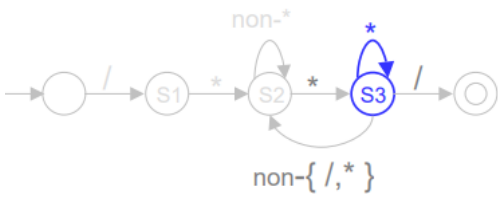
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;    /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;    /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

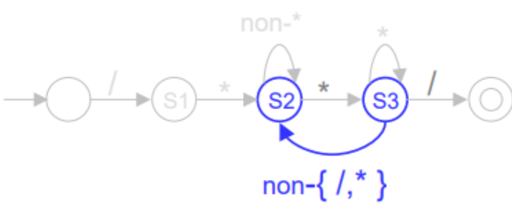
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;   /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;   /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

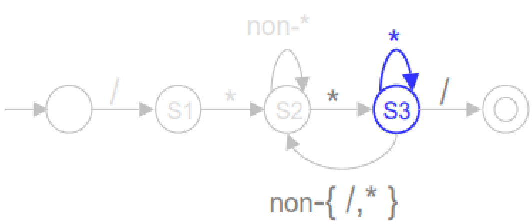
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;   /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;   /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"          BEGIN(S1);
<S1>"*"      BEGIN(S2);
<S2>[^*]     ;   /* stay in S2 */
<S2>"*"      BEGIN(S3);
<S3>"*"      ;   /* stay in S3 */
<S3>[^*/]    BEGIN(S2);
<S3>"/"      BEGIN(INITIAL);
```

Example: Program to remove comment lines from c (using inclusive start)

```
% {
int cements, code, whitespace;
% }
%s COMMENT
%%
^[ \tl*"/*" { BEGIN COMMENT; /* enter comnent eating state */ }
^[ \tl*"/*".*"*/" [ \tl*\n { commnents++; /* self-contained
commnent*/}
<COMMENT> "*/"[  \t}*\n {BEGIN 0; commnents++; }
<COMMENT> "*/"          {BEGIN 0; }
<COMMENT> \n            {comments++;}
<COMMENT>.\n            {comments++;}
^[  \t]*\n               {whitespace++;}

.+"/*".*"*/".*\n        { code++; }
.*"/*".*"*/".+\n        { code++; }
. +" /*" .*\n            { code++; BEGIN COMMENT; }
. \n                     { code++; }
.                       ;/* ignore everything */
% %
main ( )
{
YYlex( ;
printf("code: %d, comnents %d, whitespace %d, code, comments,
whiteSpace);
}
```

We added the rules ""<COMMENT>\n" and "<COMMENT>.\n" to handle the case of a blank line in a comment, as well as any text within a comment. Forcing them to match an end-of-line character means they won't match something like

/ this is the beginning of a comment*

*and this is the end */* int counter;

as two lines of comments. Instead, this will count as one line of comment and one line of code.

LEX Programs

1. Program to copy its standard input to its standard output.

```
%%
%%

(or)

%%
.|\n ECHO;
%%
```

2. Program to count the number of lines in its standard input.

```
%{
      #include<stdio.h>
       int count=0;
%}

%%
\n    count++;
.     ;
%%

void main( )
{
      yylex( );
      printf("The total number of  lines = %d\n", count);
}
```

3. /*lex program to count number of words*/

```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}

/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting
                                 number of words*/

"\n" {printf("%d\n", i); i = 0;}
```

```
%%
int yywrap(void){}

int main()
{
    // The function that starts the analysis
    yylex();

    return 0;
}
```

4. /*lex code to count words that are less than 10 and greater than 5 */

```
%{
int len=0, counter=0;
%}

%%
[a-zA-Z]+ { len=strlen(yytext);
             if(len<10 && len>5)
             {counter++;} }
%%

int yywrap (void )
{
   return 1;
}

int main()
{
printf("Enter the string:");
yylex();
printf("\n %d", counter);
return 0;
}
```

5. /* Lex program to Identify and Count Positive and Negative Numbers */

```
%{
int positive_no = 0, negative_no = 0;
%}
 /* Rules for identifying and counting positive and negative numbers*/

%%
^[-][0-9]+ {negative_no++;
             printf("negative number = %s\n",
                  yytext);} // negative number
```

```
[0-9]+ {positive_no++;
        printf("positive number = %s\n",
                  yytext);} // positive number
%%

/*** use code section ***/

int yywrap(){}
int main()
{

yylex();
printf ("number of positive numbers = %d,"
        "number of negative numbers = %d\n",
                  positive_no, negative_no);

return 0;
}
```

6. LEX program to count the number of vowels and consonants in a given string

```
%{
 int vow_count=0;
 int const_count =0;
%}

%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
%%
int yywrap(){}
int main()
{
 printf("Enter the string of vowels and consonents:");
 yylex();
 printf("Number of vowels are: %d\n", vow_count);
 printf("Number of consonants are: %d\n", const_count);
 return 0;
}
```

Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans through theinput recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of **yylex()**.

Not all tokens are of interest to the parser—in most programming languages the parser doesn't want to hear about comments and whitespace, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. We solve this problem by lettingyacc define the token codes.

Grammar

```
%token A B

%%

str: S

 ;

S: A S B

 |

 ;

%%
```

The tokens in our grammar are: A and B . Yacc defines each of these as a small integer using apreprocessor *#define*. Here are the definitions it used in this example:

```
# define A
251# define
B 252
```

Token code zero is always returned for the logical end of the input. Yacc doesn't define a symbol for it, but you can yourself if you want.

Yacc can optionally write a C header file containing all of the token definitions. You include thisfile, called *y.tab.h* on UNIX systems, in the lexer *and use the preprocessor symbols in your lexeraction code.*

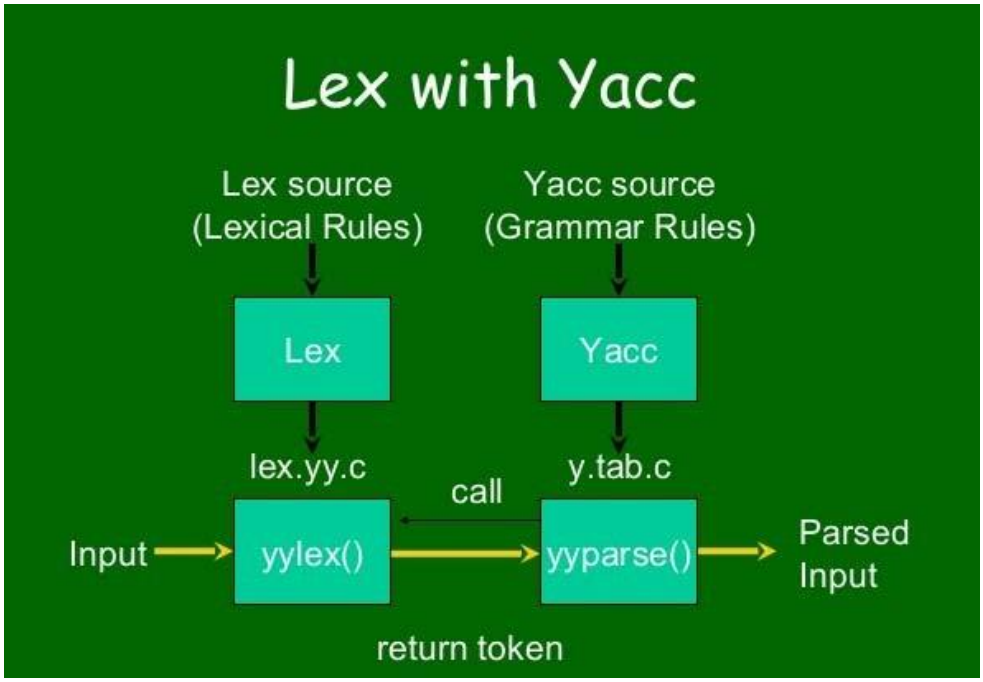
YACC

Yacc (for "**yet another compiler compiler**.") is the standard parser(**yyparse()**) generator for the Unix operating system. An open source program, **yacc** generates code for the parser in the C programming language.

Lex divides the input stream into pieces (tokens) and then yacc takes these pieces and groups them together logically.

Parser Lexer Communication

When you use a lex scanner and a yacc parser together, the parser(i.e.yyparse( )) is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans



through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of **yylex()**.

Not all tokens are of interest to the parser—in most programming languages the parser doesn't want to hear about comments and whitespace, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. We solve this problem by letting yacc define the token codes.

Grammar

A grammar is a series of *rules* that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar which is used to build a calculator.

statement → *NAME* = *expression*

expression → *NUMBER* + *NUMBER* | *NUMBER* − *NUMBER*

The vertical bar, "|", means there are two possibilities for the same symbol, i.e., an *expression*can be either an addition or a subtraction. The symbol to the left of the → is known as the *left- hand side* of the rule, often abbreviated LHS, and the symbols to the right are the *right-hand side*, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar isjust a short hand for this. Symbols that actually appear in the input and are returned by the lexer are *terminal* symbols or *tokens*, while those that appear on the left-hand side of some rule are *non-terminal* symbols or non-terminals. Terminal and non-terminal symbols must be different; it is an error to write a rule with a token on the left side.

A Yacc Parser

A yacc grammar has the same three-part structure as a lex specification. (Lex copied its structure from yacc.) The first section, the definition section, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the rules for the parser, and the third section is C code copied verbatim into the generated C program.

The Definition Section

The definition section includes declarations of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can also include a literal block, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
%token NAME NUMBER
```

We can use single quoted characters as tokens without declaring them, so we don't need to declare "=", "+", or "–".

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%

rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The Rules Section

The rules section simply consists of a list of grammar rules in much the same format as we used above. Since ASCII keyboards don't have a → key, we use a colon between the left- and right- hand sides of a rule, and we put a semicolon at the end of each rule:

```
%token NAME NUMBER
%%
statement: NAME '=' expression
|          expression
;
expression: NUMBER '+' NUMBER
|           NUMBER '–' NUMBER
;
```

Unlike lex, yacc pays no attention to line boundaries in the rules section, and you will find that a lot of whitespace makes grammars easier to read. We've added one new rule to the parser: a statement can be a plain expression as well as an assignment. If the user enters a plain expression, we'll print out its result.

The symbol on the left-hand side of the first rule in the grammar is normally the start symbol, though we can use a %**start** declaration in the definition section to override that.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side.

In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar

rules

A : B C D ;
A : E F ;
A : G ;

can be given to Yacc as

```
A : B C D
  | E F
  | G
  ;
```

- It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.
- If a nonterminal symbol matches the empty string **(ε)**, this can be indicated in the obvious way:
- **empty : ;**
- Names representing tokens must be declared; this is most simply done by writing
- **%token name1, name2 . . .**

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

- Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section.
- It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the % start keyword: **%start symbol**
- The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.
- It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end- of-file'' or ``end-of-record''.

SYMBOLS AND ACTIONS:

A literal consists of a character enclosed in single quotes ``'". As in C, the backslash ``\" is an escape character within literals, and all the C escapes are recognized. Thus

```
%token NAME NUMBER
%%
statement: NAME '=' expression
         | expression { printf("= %d\n", $1);}
;
expression: expression '+' NUMBER { $$ = $1 + $3; }
    | expression '-' NUMBER { $$ = $1 - $3; }
    | NUMBER { $$ = $1; }
;
```

The rules that build an expression compute the appropriate values, and the rule that recognizes an expression as a statement prints out the result. In the expression building rules, the first and second numbers' values are $1 and $3, respectively. The operator's value would be $2, although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value $1 to $$.

Actions:

- With each grammar rule, the user may associate actions to be Yacc: Yet Another Compiler-Compiler performed each time the rule is recognized in the input process.
- These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.
- An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

A : '(' B ')'
{ hello(1, "abc"); }
and
XXX : YYY ZZZ
{ printf("a message\n");flag = 25; }

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol “dollar sign" “$" is used as a signal toYacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to somevalue. For example, an action that does nothing but return the value 1 is { $$ = 1; }

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus,if the rule is

A : B C D ;

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

Expr : '(' expr ')' ;

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

Expr : '(' expr ')' { $$ = $2 ; }

By default, the value of a rule is the value of the first element in it ($1). Thus,grammar rules of the form

A : B ;

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes,it is desirable to get control before a rule is fully parsed. Yacc permits an actionto be written in the middle of a rule as well as at the end.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

The User subroutine Section

This section can contain any valid C Code. By default it contains main( ) function which call yyparse( ) function. The yyparse( ) function returns Zero when Stack contains only Start symbol at the end of input otherwise it will return a NON Zero value. If yyparse( ) receives a token which it doesn't know it calls yyerror( ) function.

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer- valued function called yylex. The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
extern int yylval;int c;
. . .
c = getchar();
. . .
switch( c ) {
                    . . .
                case  '0':
                case '1':
. . .
case '9':
yylval = c-'0'; return( DIGIT );
. . .

}
. . .
```

- The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

- This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser;

How the Parser Works :

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex. however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

. reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34. Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer

s

erve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

Arithmetic Expressions and Ambiguity

Let's make the arithmetic expressions more general and realistic, extending the *expression* rules to handle multiplication and division, unary negation, and parenthesized expressions:

```
expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                  {     if($3 == 0)
                              yyerror("divide by zero");
                        else
                              $$ = $1 / $3;
                  }
      |     '-' expression          { $$ = -$2; }
      |     '(' expression ')'      { $$ = $2; }
      |     NUMBER                  { $$ = $1; }
      ;
```

The action for division checks for division by zero, since in many implementations of C a zero divide will crash the program. It calls **yyerror()**, the standard yacc error routine, to report the error.

But this grammar has a problem: it is extremely **ambiguous**. For example, the input 2+3*4 might mean (2+3)*4 or 2+(3*4), and the input 3−4−5−6 might mean 3−(4−(5−6)) or (3−4)−(5−6) or any of a lot of other possibilities.

If you compile this grammar as it stands, yacc will tell you that there are 16 shift/reduce conflicts, states where it cannot tell whether it should shift the token on the stack or reduce a rule first.

Types of Conflicts

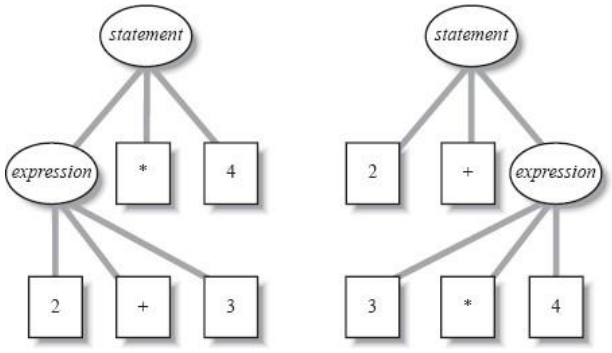
There are two kinds of conflicts that can occur when yacc tries to create a parser: “shift/reduce” and “reduce/reduce.”

Shift/Reduce Conflicts

A *shift/reduce* conflict occurs when there are two possible parses for an input string, and one of the parses completes a rule (the reduce option) and one doesn’t (the shift option). For example, this grammar has shift/reduce conflict:

```
%%
Statement: e
           ;
  e: e '+' e
   | e '-' e
   | e '*' e
   | e '/' e
  | ID
  | NUM
  ;
%%
```

For the input string “2+3*4” there are two possible parses,"(2+3)*4” or “2+(3*4)” So Yacc will tell you that there is an *shift/reduce* conflict.



We can avoid *shift/reduce* conflict by telling the yacc about the precedence and associativity of the operators. Precedence Controls which operator to execute first in an expression. Associativity controls the grouping of operators at the same precedence level.

Reduce/Reduce Conflicts

A *reduce/reduce* conflict occurs when the same token could complete two different rules. For example:

```
%
%
Prog        :   proga | progb
            ;
  Proga     :       'X'
            ;
  Progb     :       'X'
            ;
%%
```

An "X" could either be a **proga** or a **progb.** So Yacc will tell you that there is an reduce/reduce conflicts.

Compiling and Running a Simple Parser

For Compiling YACC Program:

- Write lex program in a file file.l and yacc in a file file.y
- Open Terminal and Navigate to the Directory where you have saved the files.
- type lex file.l
- type yacc file.y
- type cc lex.yy.c y.tab.h -ll
- type ./a.out

Example Programs

1. **Write a YACC program which accept strings a^nb**

LEX part

```
%{
#include "y.tab.h"
%}
%%
a return A;
```

```
b return B;
. return yytext[0];
\n return yytext[0];
%%

Yacc part
%{
/* Yacc program to recognize the grammar a^nb */
%}
%token A B
%%
str: s '\n'
s : x B
;
x : x A | A
;
%%
main()
{
printf(" Type the String ? \n");
if(!yyparse())
printf(" Valid String\n ");
}
int yyerror()
{
printf(" Invalid String.\n");
exit(0); }
```

2. **YACC program which accept strings that starts and ends with 0 or 1**

```
%{
/* Definition section */
extern int yylval;
%}

/* Rule Section */
%%
```

```
0 {yylval = 0; return ZERO;}

1 {yylval = 1; return ONE;}

.|\n {yylval = 2; return 0;}

%%
```

YACC part

```
%{
/* Definition section */
#include<stdio.h>
#include <stdlib.h>
void yyerror(const char *str)
{
printf("\nSequence Rejected\n");
}

%}
%token ZERO ONE

/* Rule Section */
%%
r : s {printf("\nSequence Accepted\n\n");}
;
s : n
| ZERO a
| ONE b
;
a : n a
| ZERO
;
b : n b
| ONE
```

```
;
n : ZERO
| ONE
;
%%

#include"lex.yy.c"
//driver code
int main()
{
 printf("\nEnter Sequence of Zeros and Ones : ");
 yyparse();
 printf("\n");
 return 0;
}
```

3. **Write a YACC program to check whether given string is Palindrome or not.**

```
%{
 /* Definition section */
 #include <stdio.h>
 #include <stdlib.h>
 #include "y.tab.h"
%}

/* %option noyywrap */

/* Rule Section */
%%

[a-zA-Z]+ {yylval.f = yytext; return STR;}
[-+()*/] {return yytext[0];}
[ \t\n] {;}

%%
```

```
int yywrap()
{
return -1;
}

YACC part
%{
 /* Definition section */
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 extern int yylex();
 void yyerror(char *msg);
 int flag;
 int i;
 int k =0;
%}
%union {
 char* f;
}
%token <f> STR
%type <f> E

/* Rule Section */
%%
S : E {
        flag = 0;
        k = strlen($1) - 1;
        if(k%2==0){

        for (i = 0; i <= k/2; i++) {
        if ($1[i] == $1[k-i]) {
                } else {
                flag = 1;
```

```
                }
        }
        if (flag == 1) printf("Not palindrome\n");
        else printf("palindrome\n");
        printf("%s\n", $1);

        }else{
        for (i = 0; i < k/2; i++) {
        if ($1[i] == $1[k-i]) {
        } else {
                flag = 1;
                }
                }
        if (flag == 1) printf("Not palindrome\n");
        else printf("palindrome\n");
        printf("%s\n", $1);



        }
 }
;
E : STR {$$ = $1;}
;
%%
void yyerror(char *msg)
{
 fprintf(stderr, "%s\n", msg);
 exit(1);
}
//driver code
int main()
{
 yyparse();
 return 0;
}
```

4. **Write YACC program for Binary to Decimal Conversion.**

```
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}

[ \t] {;}
\n return 0;
. return yytext[0];
%%

int yywrap()
{
return 1;
}
```

YACC

```
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *s);
%}
%token ZERO ONE

/* Rule Section */
%%
```

```
N: L {printf("\n%d", $$);}
L: L B {$$=$1*2+$2;}
| B {$$=$1;}
B:ZERO {$$=$1;}
|ONE {$$=$1;};
%%
//driver code
int main()
{
while(yyparse());
}
yyerror(char *s)
{
fprintf(stdout, "\n%s", s);
}
```

5. **Write a YACC program to evaluate a given arithmetic expression consisting of '+', '-', '*', '/' including brackets.**

```
%{
   /* Definition section*/
   #include "y.tab.h"
   extern yylval;
}%

%%
[0-9]+    {yylval = atoi(yytext);
           return NUMBER;}
 [a-zA-Z]+    { return ID; }
[ \t]+        ;  /*For skipping whitespaces*/

\n           { return 0; }
.           { return yytext[0]; }
 %%
```

```
%{
   /* Definition section */
  #include <stdio.h>
%}

%token NUMBER ID
// setting the precedence
// and associativity of operators
%left '+' '-'
%left '*' '/'

/* Rule Section */
%%
E : T       {
              printf("Result = %d\n", $$);
              return 0;
            }

T :
   T '+' T { $$ = $1 + $3; }
   | T '-' T { $$ = $1 - $3; }
   | T '*' T { $$ = $1 * $3; }
   | T '/' T { $$ = $1 / $3; }
   | '-' NUMBER { $$ = -$2; }
   | '-' ID { $$ = -$2; }
   | '(' T ')' { $$ = $2; }
   | NUMBER { $$ = $1; }
   | ID { $$ = $1; };
% %

int main() {
   printf("Enter the expression\n");
   yyparse();
}
```

```
/* For printing error messages */
int yyerror(char* s) {
   printf("\nExpression is invalid\n");
}
```

Review Questions of Module-4

1. Give the specification of yacc program? Give an example? (8)
2. What is grammar? How does yacc parse a tree? (5)
3. How do you compile a yacc file? (5)
4. Explain the ambiguity occurring in an grammar with an example? (6)
5. Explain shift/reduce and reduce/reduce parsing ? (8)
6. Write a yacc program to test the validity of an arthimetic expressions? (8)
7. Write a yacc program to accept strings of the form anbn , n>0? (8)