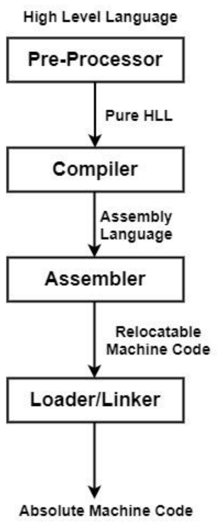
UNIT I SYLLABUS

Introduction: Languages Processors – Structure of Compiler – Applications of Compiler Technology – Programming Language basics.
Lexical Analysis: Role of Lexical Analyser - Input Buffering- Specifications and recognition of tokens – Lexical – Analyser generator Lex- Finite Automata – From regular expressions to Automata – Design of a Lexical Analyser Generator – Optimization of DFA Based pattern.

1.1 Language Processors

In a language processing system, the source code is first preprocessed. The modified source program is processed by the compiler to form the target assembly program which is then translated by the assembler to create relocatable object codes that are processed by linker and loader to create the target program. It is based on the input the translator takes and the output it produces, and a language translator can be defined as any of the following.

High-Level Language − If a program includes #define or #include directives, including #include or #define, it is known as HLL.



Pre-Processor − The pre-processor terminates all the #include directives by containing the files named file inclusion and all the #define directives using macro expansion. A pre-processor can implement the following functions −

- **Macro processing** − A preprocessor can enable a user to define macros that are shorthands for higher constructs.
- **File inclusion** − A preprocessor can include header files into the program text.
- **Rational preprocessor** − These preprocessors augment earlier languages with additional current flow-of-control and data structuring facilities.
- **Language Extensions** − These preprocessors try to insert capabilities to the language by specific amounts to construct in macro.

Pure HLL − It means that the program will not contain any # tags. These # tags are also known as preprocessor directives.

Assembler − Assembler is a program that takes as input an assembly language program and changes it into its similar machine language code.

Assembly Language − It is an intermediate state that is a sequence of machine instructions and some other beneficial record needed for implementation. It neither in the form of 0's and 1's.

Advantages of Assembly Language

- Reading is easier.
- Addresses are symbolic & programmers need not worry about addresses.
- It is mnemonics. An example we use ST instead of 01010000 for store instruction in assembly language.
- It is easy to find and correct errors.

Relocatable Machine Code − It means that you can load that machine code at any point in the computer and it can run. The address inside the code will be so that it will maintain the code movement.

Loader / Linker − This is a code that takes as input a relocatable program and compiles the library functions, relocatable object records, and creates its similar absolute machine program.

Linking enables us to create a single program from several documents of a relocatable machine program. These documents can have been resulting in different compilations, one or more can be library routines supported by the system available to any code that requires them.

The **loading** includes taking the relocatable machine program, changing the relocatable addresses, and locating the modified instructions and information in memory at the suitable area.

1. 2 Introduction to Compilers

Compiler

- The Compilers act as translators, transforming human-oriented programming language into Computer-oriented machine language.

$$\begin{tikzcd}\textbf{(Source Language)}\\\textbf{Programming Language} \arrow[r] & \boxed{\textbf{Compilers}} \arrow[r] & \textbf{(Target Language)}\\\textbf{Machine Language}\end{tikzcd}$$

- A Compiler is a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language.

Compiler – target codes

- Compilers may be **distinguished in two ways**
  1. By the kind of machine code they generate
  2. By the format of target code they generate

- **Kind of Machine Codes**
  Compilers may generate **any of three types of code** by which they can be differentiated:
  1. Pure machine code
  2. Augmented machine code
  3. Virtual machine code

  1. Pure Machine Code:
  - Compilers may generate code for a particular machine, not assuming any operating system or library routines.
  - This is "pure code" because it includes nothing beyond the instruction set.
  - Pure code can execute on bare hardware without dependence on any other software.

  2. Augmented Machine Code:
  - Commonly, compilers generate code for a machine architecture augmented with operating system routines and run-time language support routines.
  - To use such a program, a particular operating system must be used and a collection of run-time support routines (I/O, storage allocation, mathematical functions, etc.) must be available.

  3. Virtual Machine Code:
  - Generated code can consist entirely of virtual instructions (no native code).
  - This supports transportable code that can run on a variety of computers.
  - **Java**, with its JVM (Java Virtual Machine) is a great example of this approach.

- **Formats of Translated Programs**

  Compilers differ in the format of the target code they generate. Target formats may be categorized as
  1. Assembly language,
  2. Relocatable binary,
  3. Memory-image (or) Absolute binary

1. Assembly Language (Symbolic) Format:
- A text file containing assembler source code is produced.
- A number of code generation decisions (jump targets, long vs. short address forms, and so on) can be left for the assembler.
- Generating assembler code supports cross compilation and also simplifies debugging and understanding a compiler

2. Memory-Image (Absolute Binary) Form:
- Compiled code may be loaded into memory and immediately executed.
- This is faster than going through the intermediate step of link/editing.
- The ability to access library and precompiled routines may be limited.
- The program must be recompiled for each execution.

3. Relocatable Binary Format:
- Target code may be generated in a binary format with external references and local instruction and data addresses are not yet bound.
- Instead, addresses are assigned relative to the beginning of the module or relative to symbolically named locations.
- A linkage step adds support libraries and other separately compiled routines and produces an absolute binary program format that is executable.

Structure of Compilers

A Compiler is a program that maps a source program into a semantically equivalent target program. There are two parts to that mapping:

1. Analysis part
2. Synthesis part

Analysis Part - Analysis of the Source Program

Synthesis Part - Synthesis of the Machine language Program

A. Analysis Part

The Analysis part collects information about the source program and performs the following,
- Breaks up the source program into constituent pieces called TOKENs.
- Imposes a grammatical structure on the pieces
- Makes the source program syntactically well semantically sound.
- Create the intermediate representation of the source program.

Analysis on the Source Program

The analyses helps to collect information about the source program are,

1. Linear or Lexical Analysis
2. Hierarchical or Syntax Analysis
3. Semantic Analysis

1. Linear (or) Lexical Analysis:

- The stream of characters making up the source program is read from left to right and grouped into Tokens.
- Linear analysis is also called lexical analysis or scanning.
- The blanks separating the character of tokens are eliminated.

2. Hierarchical (or) Syntax Analysis:

- The tokens of the source program are grouped into grammatical phrases that are used by the compiler to synthesize the output.
- Hierarchical analysis is also called Parsing or Syntax Analysis.
- The hierarchical structure of a program is usually expressed by recursive rules.

3. Semantic Analysis:

- It checks the source program for semantic errors and gathers type information for the code generation phase.

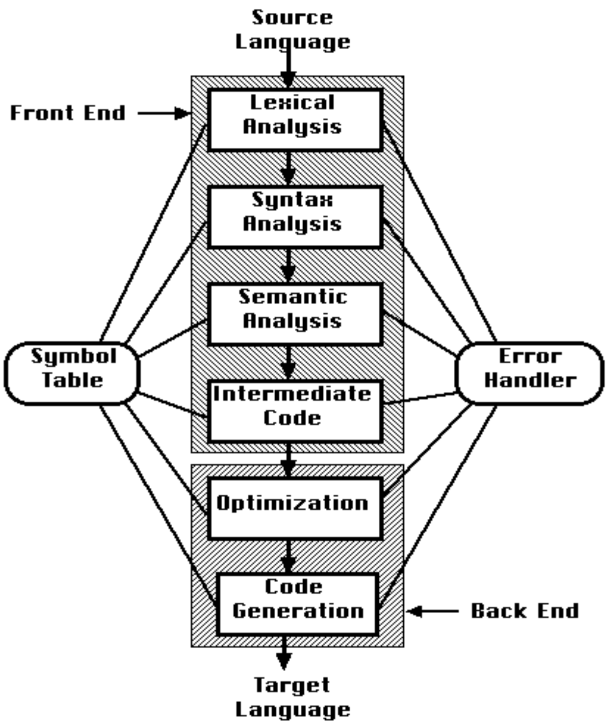
B. Synthesis Part:

The Synthesis part reads intermediate representation from analysis part and produces optional assembly language or constructs desired target language.

Phases of Compilers

- The compilation process operates in phases, each of which transforms one representation of the source program into another.
- The typical decomposition of phases of compilers are
    1. Lexical Analysis
    2. Syntax Analysis
    3. Semantic Analysis
    4. Intermediate Code Generation
    5. Code optimization
    6. Code Generation

  and two supporting phases includes
    1. Symbol Table Management
    2. Error Handling.



1. Lexical Analysis

- The Lexical Analyzer SCAN the source program one character at a time from left-to-right. It groups the stream of characters into meaningful sequences called TOKENs.
- It keeps track of line numbers to indicate errors.
- The blanks separating the tokens and comment line statements would be discarded.
- It makes an entry into the symbol table on recognizing identifiers used in the source program.

2. Syntax Analysis

- The Syntax analyzer or parser groups the token stream produced by the Lexical analyzer into grammatical phrases with collective meaning.
- It checks them against the Grammar of the Source language and produces the valid syntactical construct in the form of tree, called parse tree.

3. Semantic Analysis

- The Semantic analyzer uses the parse tree generated by the syntax analyzer and the information in the Symbol table to,
  - Check the source program for semantic consistency with the language definition,
  - Update the type information into the symbol table.

4. Intermediate Code Generation

- The ICG generates an explicit intermediate representation of the source program.
- The Intermediate representation can have a variety of forms as
  1. Syntax tree
  2. Postfix form
  3. Three Address Code (TAC) Statements

- The Intermediate representation should have two properties
    - It should be easy to produce from the source program
    - It should be easy to translate into equivalent target program.

5. Code Optimization

- The code optimizer attempts to improve the quality of Intermediate code, so that faster running target code will result.
- The Code optimizer considers the factors like algorithm, memory, running time of the target program by applying code improving transformations.
- The Code optimization is optional and can be performed,
    - **Before Code generation** – transformation applied over the intermediate representation of the source program, called Pre-Optimization or Machine Independent Optimization.
    - **After Code Generation** – transformation applied on the target code, called post optimization or machine dependent optimization.

6. Code Generation

- The code generator takes as input an optimized Intermediate representation of the source program and maps it into the target language.
- Each intermediate instruction is translated into a sequence of machine instructions that perform the same task by using some pattern matching algorithm.

7. Symbol Table Management

- The Symbol Table is a data structure containing a record for each identifier with fields for attributes for the same.
- It is an essential function of compiler to record the symbols used in the source program and collects the information about various attributes (like storage, type, scope) of each symbol.

8. Error Handling

- One of the most important functions of a compiler is the detection and reporting of errors in the source program.
- The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.
- Both of the symbol table-management and error-Handling routines interact with all phases of the compiler.

1.5. Grouping of Phases

- In an implementation, activities of several phases may be grouped into a pass that reads an input and writes an output file.

- The Front-end phases of Lexical analysis, Syntax analysis, Semantic analysis and Intermediate code generation might be grouped into one pass.
- There could be a back-end pass consisting of optional code optimization and code generation for a particular target machine.

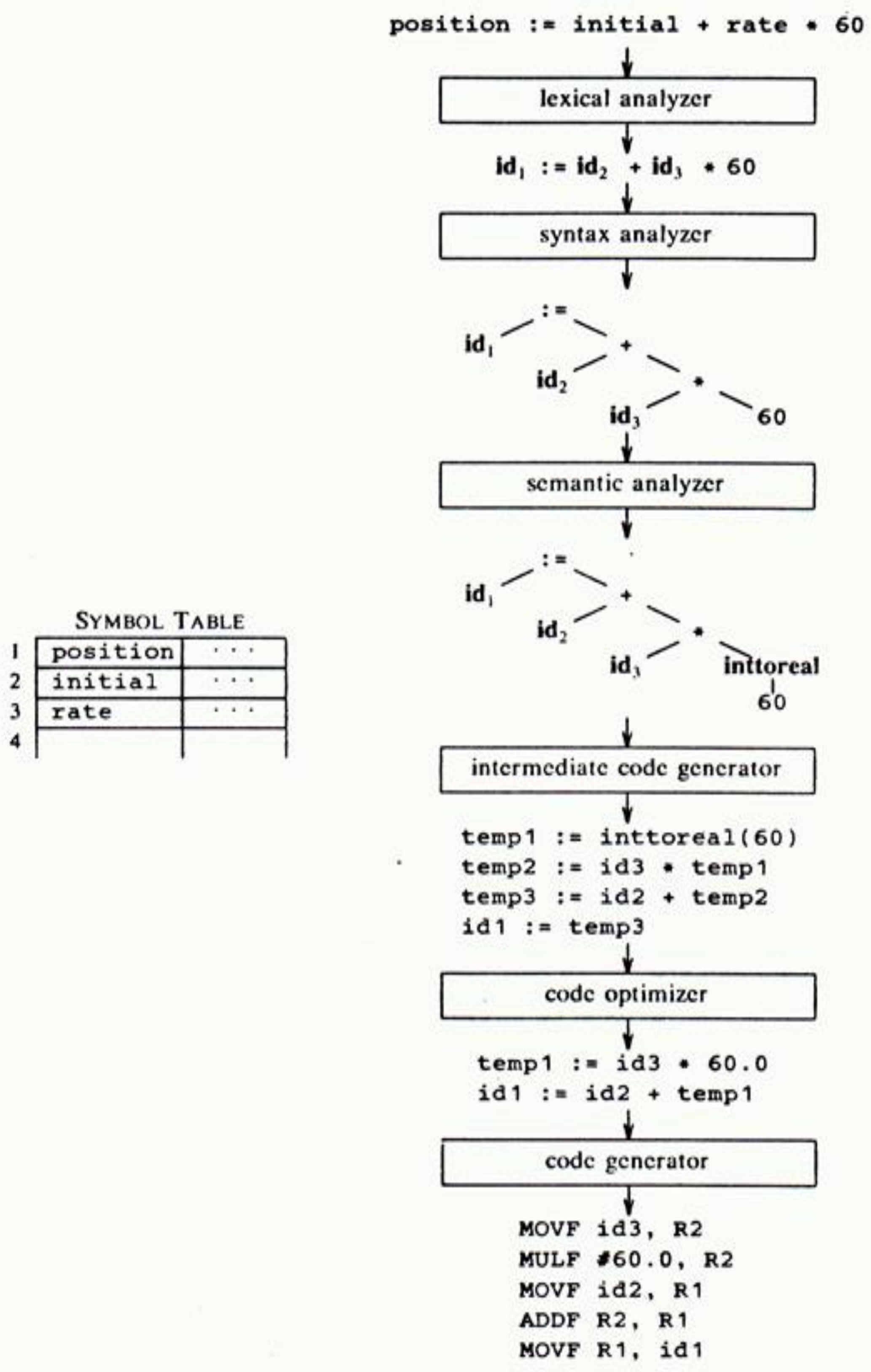
Benefits - Grouping of Compiler Phases

A carefully designed Intermediate representation allows the front-end for a particular language to interface with the back-end for different target machine and also different front-ends to interface with a single back-end.

- **Platform Integration -** Producing compilers for different target machines, by combining a front end with back-ends for different target machines.
  Example: JAVA technology.

- **Language Integration**: Producing compilers for different source languages for one target machine by combining different front-ends with the back-end for that target machine. **Example: .NET technology.**

Translation of an Assignment Statement: position = initial + rate * 60



1	position	...
2	initial	...
3	rate	...
4		

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Compiler Construction Tools

- In addition to general-software development tools, more specialized tools have been created to help implement various phases of a compiler.
- They also have known as Compiler Compilers or Compiler Writing Systems.
  The commonly used compiler-construction tools include,

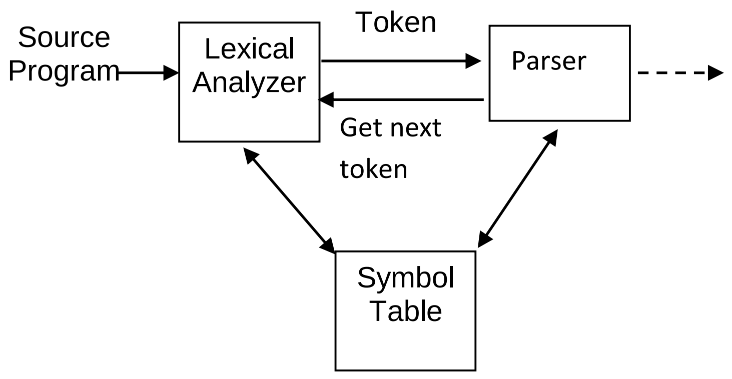
1. Scanner Generator
2. Parser Generator
3. SDT Engine
4. Data flow analysis Engine
5. Code Generator Generators
6. Compiler-Construction Toolkit

1. **Scanner Generators** – produce lexical analysers from a regular expression description of the tokens of a language. Example: LEX tool.
2. **Parser Generators** – automatically produce syntax analyzer from a grammatical description of a programming language.
3. **Syntax-Directed translation Engines** – produce collection of routines for traversing a parse tree and generating intermediate code.
4. **Code Generator Generators** – produce a code generation from a collection of rules for translating each operation of the intermediate language into machine language for target machine.
5. **Data Flow Analysis Engine** – facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data flow analysis is a key part of code optimization.
6. **Compiler Construction Toolkit** – provide an integrated set of routines for constructing various phases of a compiler.

2. LEXICAL ANALYSIS

2.1 The Role of the Lexical Analyser

- Lexical analyzer is the first phase of compiler.
- The lexical analyzer reads the input source program from left to right one character at a time and generates sequence of tokens.
- As the lexical analyzer scans the source program to recognize tokens, it is also called as scanner.



Functions of the lexical analyzer

- The Lexical analyzer is the only phase of the compiler that reads the source program, it may perform certain tasks besides identification of lexemes which includes,
  - Stripping out comments and whitespaces (blank, newline, tab and other characters that are used to separate tokens in the input.)
  - Correlating error messages generated by the compiler with the source program.
  - It may keep track of newline characters seen, so it can associate a line number with error message.
  - On recognizing any identifier, the information about an identifier – its lexeme and the location at which it is first found is entered in the Symbol table.

The lexical analyzers are divided into a cascade of two processes,

1. Scanning

- Scanning is the process of reading the source text from left to right one character at a time.
- Tasks include deletion of comments and compaction of consecutive white space characters into one. It do not require tokenization of the input

2. Lexical Analysis

- Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner

2.2 Lexical Analyzer - Design Issues

- There are a number of reasons why the analysis part of a compiler is normally separated into lexical analysis and parsing phases,

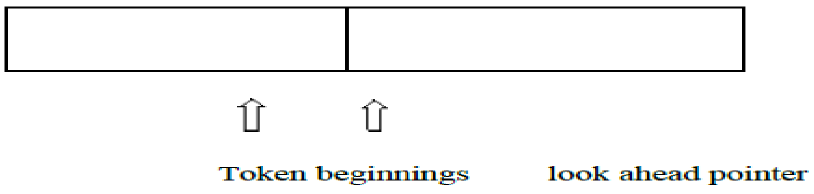
  - **Simplicity of design** – The separation of lexical and syntactic analyses allows the compiler designer to simplify at least one of these tasks.
  - **Compiler efficiency is improved** – a separate lexical analyzer allows the designer to apply specialized techniques that serve only the lexical task. Specialized Input buffering technique for reading input character can speed up the compiler significantly.
  - **Compiler portability is enhanced** – Input device specific peculiarities can be restricted to the lexical analyzer.

2.3 Lexical Errors & Recovery actions

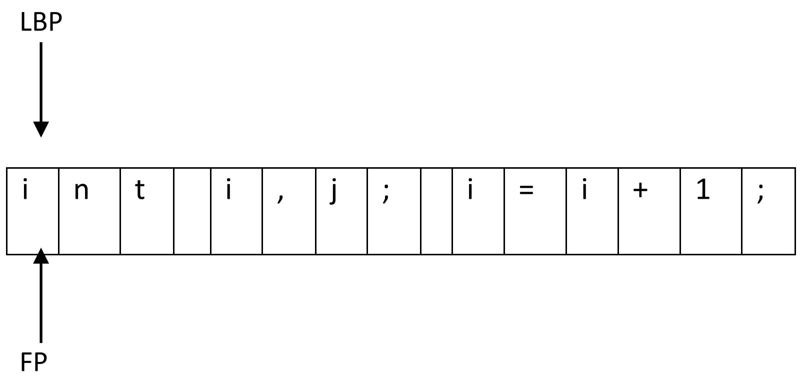
- The lexical errors are very simple in nature like misspelling of Identifiers, keywords of operators, etc.
- Possible Error recovery actions are,
  - ✓ Delete one character from the remaining input
  - ✓ Insert a missing character into the remaining input
  - ✓ Replace a character by another character
  - ✓ Transpose two adjacent characters

2.4 Input Buffering – Scanning

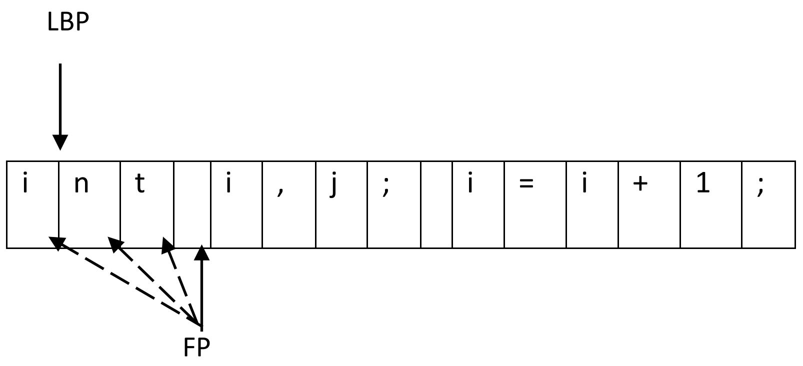
- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- Using a pair of buffers (instead of single buffer) cyclically and ending each buffer's contents with a sentinel character that warns of its ends are the two techniques that accelerate the process of buffering.
- The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers Lexeme_begin_pointer (LBP) and Forward_ptr (FP) to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string.



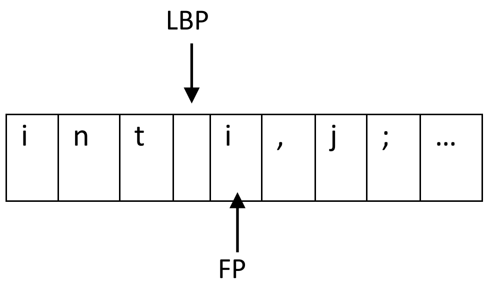
- **Example:** Input scanned from the source program is → int i,j; i=i+1;



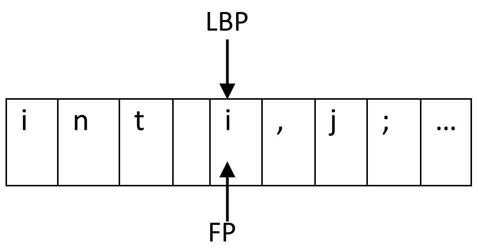
- The forward_ptr moves ahead to search for end of lexeme. As soon as the white space is encountered it indicates end of lexeme. In the above example as soon as the forward_ptr (FP) encounters a blank space the lexeme "int" is identified.



- At the moment the forward_ptr is located in the blank space. So the forward_ptr will ignore the blank space and it will move ahead. Then both the Lexeme_begin_ptr (LBP) and the forward_ptr (FP) will be set for the next string.



Forwarding the LBP ahead.



Setting of the two pointers

Organization of Dual Buffering Scheme:

- This scheme involves two buffers that are alternatively reloaded. Each buffer is of the same size N and N is usually the size of a disk.

Two pointers to the input are maintained

Pointer *Lexeme Begin*, marks the beginning of the current lexeme, whose extent we are attempting to determine

Pointer *Forward,* scans ahead until a pattern match is found.

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

Sentinels

If we use the scheme of Buffer pairs we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read.

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF.**

Note that **EOF** retains its use as a marker for the end of the entire input. Any **EOF** that appears other than at the end of a buffer means that the input is at an end.

2.5 Lexical 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.

LEXEME

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A lexeme is identified by the lexical analyzer as an instance of the token.

PATTERN

- A pattern is a description of the form the lexemes of a token may take.
- Example: Keywords – the pattern is just the sequence of characters that form the token.

Examples of Tokens

Token	lexeme	pattern
const	const	const
if	if	if
Relational Operator	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or
Identifier	pi , name, a, n	Letter followed by letter or digits
Number	3.14	any numeric constant
Literal	"core"	pattern

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent phases additional information about the particular lexeme that matched.
- The lexical analyzer collects information about tokens into their associated attributes.
- The token influence parsing decision; the attribute influence the translation of tokens.
- An ID token has usually only a single attribute- a *pointer* (*index*) to the symbol-table entry in which the information about the token is kept.
- The Lexical analyzer produces as output a token of the form,

< Token name, Attribute value >

Where

Token name – is an abstract symbol representing a kind of lexical unit.

Attribute value – an optional attribute value providing additional information to the parser.

Lexeme	Token Name	Attribute Value
Any white space	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to symbol table entry
Any number	number	pointer to symbol table entry
<	relop	LT

Let C = A + B; is any statement in the source program, the tokens are

<ID, Pointer to ST Entry for C>
<Assign_OP>
<ID, Pointer to ST Entry for A>
<Arith_OP>
<ID, Pointer to ST Entry for B>

2.6 Specification of Tokens

- Regular Expressions are an important notation for specifying lexeme patterns.
- Regular expression (R.E.) is a mathematical symbolism which is used to describe the representation of strings by using defining rules.
- The REs are built recursively out of smaller regular expressions using the rules and properties.
- Some of the algebraic properties that hold for arbitrary regular expression r, s and t

$r \mid s = s \mid r$	$\rightarrow$ | is Commutative
$r \mid (s \mid t) = (r \mid s)t$	$\rightarrow$ | is Associative
$r(st) = (rs)t$	$\rightarrow$ Concatenation is associative
$r(st) = rs \mid rt;\ (s \mid t)r = sr \mid st$	$\rightarrow$ Concatenation is distributive over |
$\epsilon r = r\epsilon = r$	$\rightarrow$ ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	$\rightarrow$ ϵ is guaranteed in a closure
$r^{**} = r^*$	$\rightarrow$ * is idempotent.

Strings and Languages

- An alphabet denotes any finite set of symbols,
  - {0,1}: binary alphabet
  - ASCII code: computer alphabet
- A string over some alphabet is a finite sequence of symbols drawn from the alphabet.
- A language denotes a set of strings over some fixed alphabet.
- The string exponentiation operation is defined as $s^0 = \varepsilon$ (empty string);
  $s^i = s^{i-1}s$, for $i>0$ (string concatenation)

2.7 Specification of tokens using RE

(1) Pattern for Identifier Token

letter $\rightarrow$ [A-Za-z]
digit $\rightarrow$ [0-9]
id $\rightarrow$ letter (letter|digit)*

(2) Pattern for Integer Constant (signed/unsigned) Token

digit $\rightarrow$ [0-9]
num $\rightarrow$ $(+ \mid - \mid \epsilon)\ \text{digit}^+$

(3) Pattern for Real constant Token

digit $\rightarrow$ [0-9]
num $\rightarrow$ $\text{digit}^+.\text{digit}^+\ (\text{E}[+-]\text{digit}^+)$

(4) Pattern for keywords Token

if → if
main →main
int → int
else → else
void → void

(5) Pattern for Operators Token

rel_op → < | <= |> |>= | == | <>
arith_op → + | - | * | / |%

(6) Pattern for delimiters (whitespace) Token

ws → (blank | tab | newline)$^+$

- R.E. is only one statement rule.
- If the used statement is used to define the basic symbols of the R.E. then this type of statement is called Regular Definition such as the case in Letter and the case in Digit.

2.8 Recognition of Token

- The Token recognition is the process of takes the patterns for all the needed tokens and examines the input string and finds a lexeme matching one of the patterns.
- As an intermediate step in construction of lexical analyzer, first convert patterns into stylized flow charts, called "**transition diagrams**".
- Transition diagrams have 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 to another and are labeled by a symbol or set of symbols.

(1) Recognizer for Identifier Token

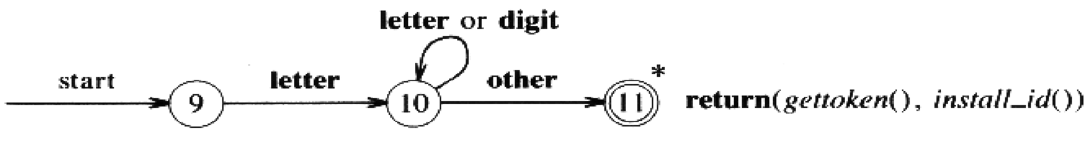
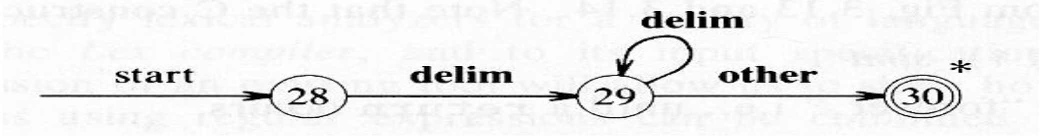


Fig. 3.13. Transition diagram for identifiers and keywords.

(2) Recognizer for delimiters Token



(3) Recognizer for Operators Token (relational operator)

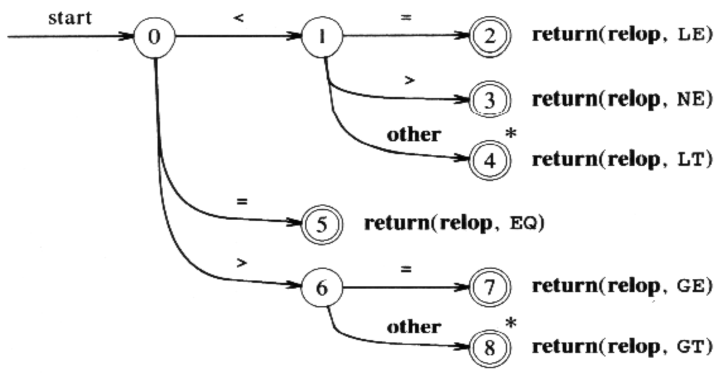
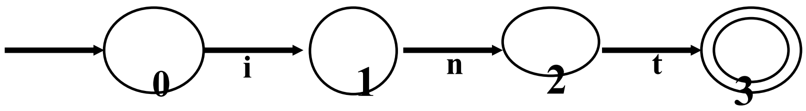


Fig. 3.12. Transition diagram for relational operators.

(4) Recognizer for keyword token - "int"



(5) Recognizer for Literals Token

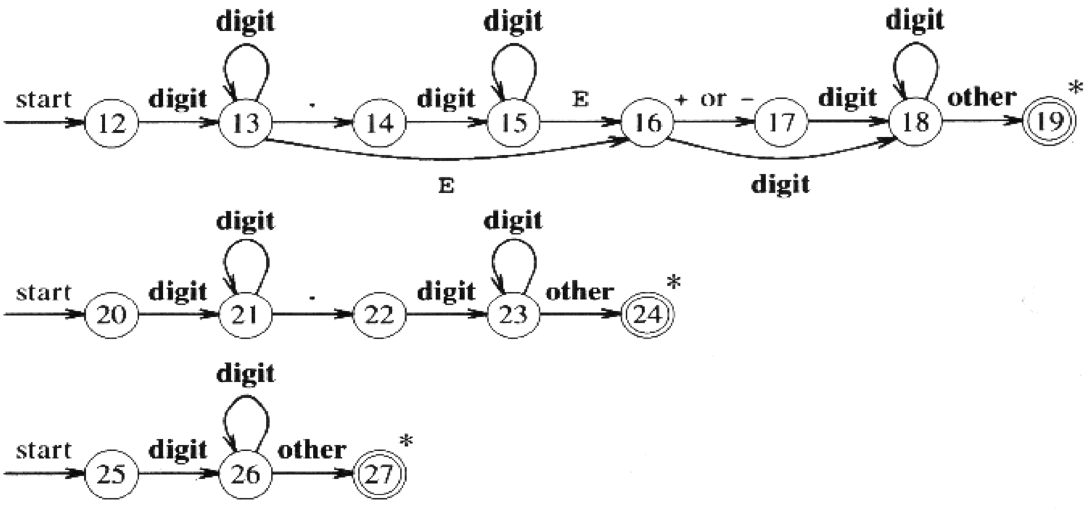


Fig. 3.14. Transition diagrams for unsigned numbers in Pascal.

<u>2.9 Finite Automata (FA)</u>

- FA also called Finite State Machine (FSM) ia an abstract model of a computing entity.
- FA can decide whether to accept or reject a string. Every regular expression can be represented as a FA and vice versa
- There are two types of FAs:
    - **Non-deterministic (NFA):** Has more than one alternative action for the same input symbol.
    - **Deterministic (DFA):** Has at most one action for a given input symbol.

- Regular expression is a declarative way to describe the token. It describes what a token is, but not how to recognize the token, whereas the FA is used to describe how the token is recognized.

NFA (Non-deterministic Finite Automaton)

An NFA is a 5-tuple (S, Σ, δ, S0, F)

S: a set of states;

Σ: the symbols of the input alphabet;

δ : a set of transition functions; move(state, symbol) → a set of states

S0: s0 ∈S, the start state;

F: F ⊆ S, a set of final or accepting states.

DFA (Deterministic Finite Automaton)

- A special case of NFA where the transition function maps the pair (state, symbol) to one state.
  - When represented by transition diagram, for each state *S* and symbol *a,* there is at most one edge labeled *a* leaving *S;*
  - When represented transition table, each entry in the table is a single state.
  - There are no ε-transition

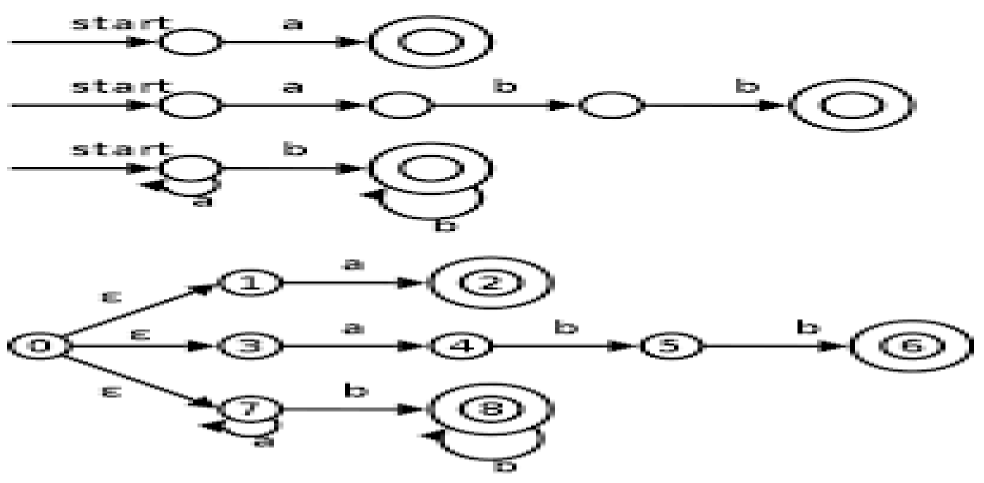
Conversion from NFA to DFA: Algorithm & Example: Please refer Class-work Notebook.

<u>2.10 Design of Lexical Analyzer Generator</u>

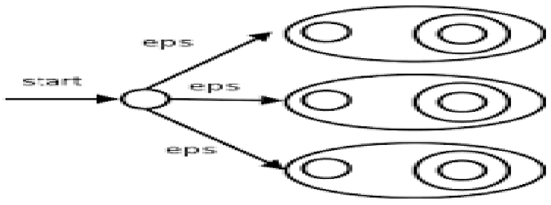
For the LEX program

a	**{action A1 for pattern p1}**
abb	**{action A2 for pattern p2}**
a*b+	**{action A3 for pattern p3}**

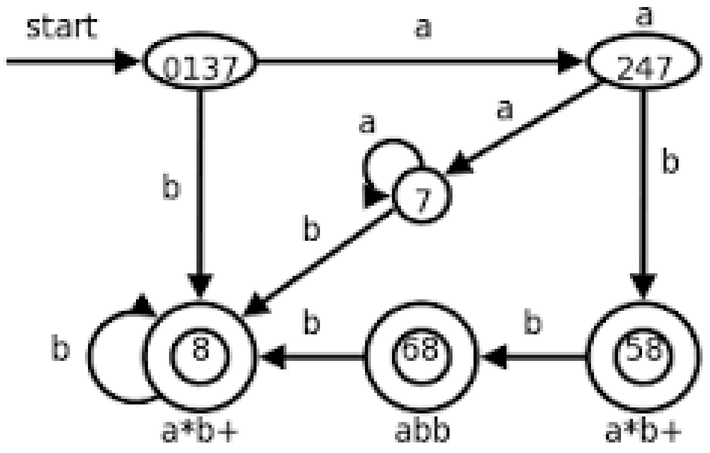
Step 1: Construct the automaton by taking each regular –expression pattern in the LEX program and converting it to an NFA.



Step 2: Combine all the NFA's into one by introducing a new start state with ϵ – transitions to each start states of the NFA$_i$ for pattern P$_i$.

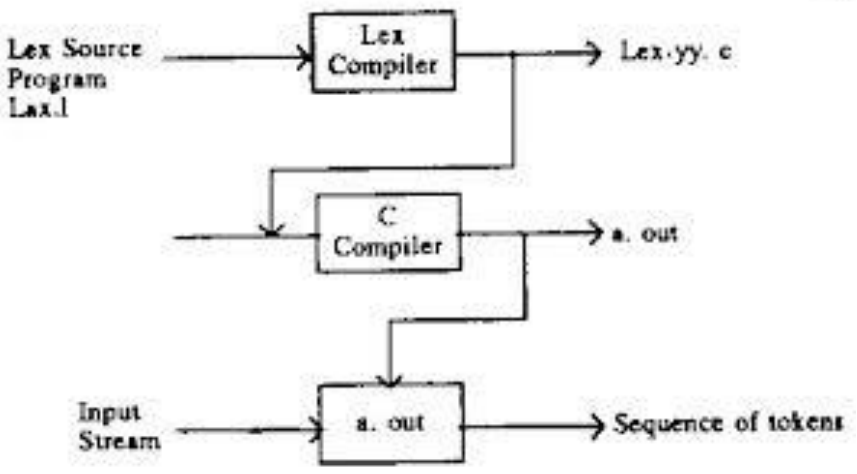


Step 3: Convert the NFA for all the patterns into an equivalent DFA, using Subset Construction method.



2.11 LEX

- LEX is a tool that allows one to specify a lexical analyzer by specifying regular expression to describe patterns for tokens.
- The notation for LEX tool is referred to as the Lex Language and the tool itself is the LEX compiler.
- The LEX Compiler transforms the input patterns into a transition diagram and generates code in a file called lex.yy.c that simulates this transition diagram.



2.12 LEX specifications

A Lex program (the .l file) consists of three parts:

declarations

%%
translation rules
%%

auxiliary procedures

1. The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.
2. The translation rules of a Lex program are statements of the form

pattern 1 {action 1}
pattern 2 {action 2}
pattern 3 {action 3}
… …
… …

where each pattern 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.
3. The third section holds whatever auxiliary procedures are needed by the actions. Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

What a LA will do upon recognizing a comments in the Source program?

A lexical analyzer, also known as a lexer or scanner, is responsible for converting a sequence of characters from the source code into a sequence of tokens. When the lexer encounters comments in the source program, its typical actions include:

1. **Identifying Comments**: The lexer identifies the start and end of the comment based on the syntax of the programming language (e.g., // for single-line comments or /* ... */ for multi-line comments in languages like C).
2. **Skipping Comments**: Once a comment is identified, the lexer usually skips over the entire comment, effectively ignoring it. This means the content of the comment is not converted into tokens or passed on to subsequent phases of the compiler.
3. **Maintaining Position Information**: While skipping comments, the lexer still keeps track of the position within the source code (line numbers, column numbers) to ensure accurate error reporting and debugging information.
4. **Optional Storage for Documentation**: In some cases, especially with documentation comments (e.g., Javadoc comments in Java, Doxygen comments in C++), the lexer might pass these comments to a separate documentation generator tool, but they are still not considered as part of the executable code.

To summarize, the main role of the lexical analyzer upon recognizing comments is to identify and skip them, ensuring that they do not affect the generation of tokens for the actual source code.