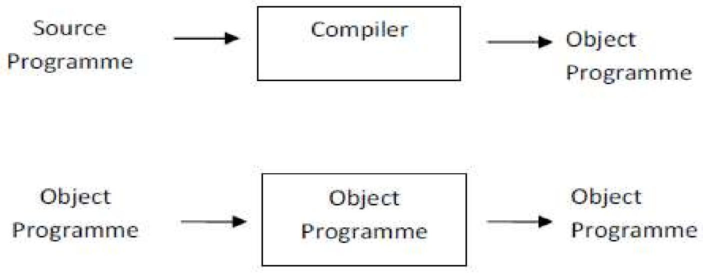
DR. SOTONWA

COMPILER CONSTRUCTION

REVIEW OF COMPILER, ASSEMBLER AND INTERPRETER

A translator is a programme that takes as input a programme written in one programming language (the source language) and produces as output a programme in another language (the object or target language). If the source language is a high-level language such as COBOL, PASCAL, etc. and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a *Compiler*.
Executing a programme written in a high-level programming language is basically a two-step process, as illustrated in Figure 1. The source programme must first be compiled, that is, translated into object programme. Then the resulting object programme is loaded into memory and executed.



Compilation and Execution

Certain other translators transform a programming language into a simplified language, called intermediate code, which can be directly executed using a programme called an interpreter. You can think of the intermediate code as the machine language of an abstract computer designed to execute the source code. There are other important types of translators, besides compilers. If the source language is assembly language and the target language is machine language, then the translator is called an *assembler*. The term *preprocessor* is used for translators that take programs in one high level language into equivalent programs in another high level language. For example, there many FORTRAN preprocessors that map 'structured' versions of FORTRAN into conventional FORTRAN.

Why Do We Need Translators?

We need translators to overcome the rigor of programming in machine language, which involves communicating directly with a computer in terms of bits, register, and primitive machine operations. A machine language program is a sequence of 0's and 1's, therefore, programming a complex algorithm in such a language is terribly tedious and prone to mistakes. Translators free the programmer from the hassles of programming in machine language.

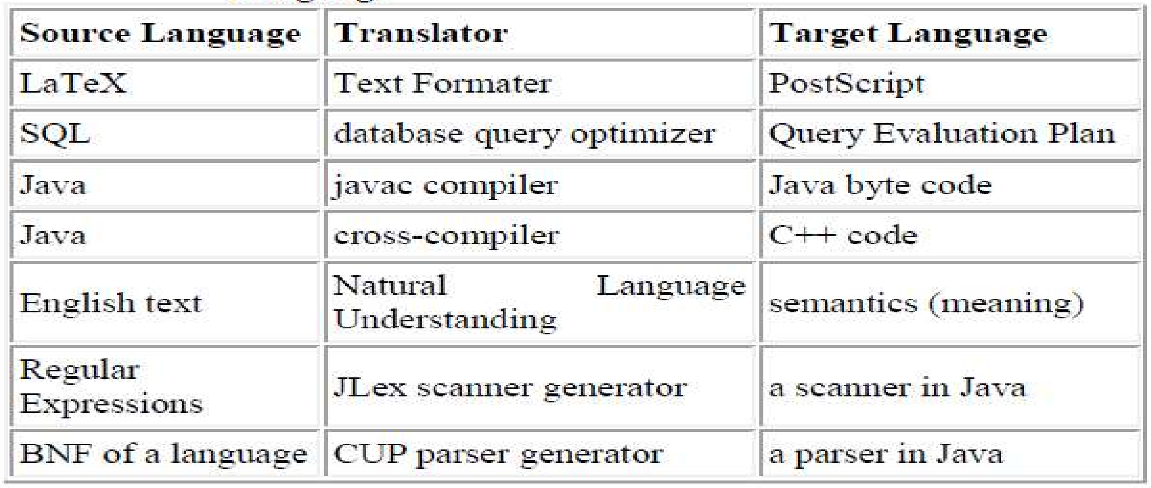
What is a Compiler?

A *compiler* is a program that translates a source program written in some high-level programming language (such as Java) into machine code for some computer architecture. The generated machine code can later be executed many times against different data each time. An *interpreter* reads an executable source programme written in a high level programming language as well as data for this programme, and it runs the program against the data to produce some results. One example is the UNIX shell interpreter, which runs operating system commands interactively.

You should note that both interpreters and compilers (like any other program) are written in some high-level programming language (which may be different from the language they accept) and they are translated into machine code. For example, a Java interpreter can be completely written in Pascal, or even Java. The interpreter source program is machine independent since it does not generate machine code. (Note the difference between *generate* and *translated into* machine code.) An interpreter is generally slower than a compiler because it processes and interprets each statement in a programme as many times as the number of the evaluations of this statement. For example, when a for loop is interpreted, the statements inside the for loop body will be analyzed and evaluated on every loop step. Some languages, such as Java and Lisp, come with both an interpreter and a compiler. Java source program (Java classes with .java extension) are translated by the java compiler into byte-code files (with .class extension). The Java interpreter, java, called the Java Virtual Machine (JVM), may actually interpret byte codes directly or may internally compile them to machine code and then execute that code.

Compilers and interpreters are not the only examples of translators. In the table below are a few more:

Table 1: Table of Translators, Source Language and Target Language



Source Language	Translator	Target Language
LaTeX	Text Formater	PostScript
SQL	database query optimizer	Query Evaluation Plan
Java	javac compiler	Java byte code
Java	cross-compiler	C++ code
English text	Natural Language Understanding	semantics (meaning)
Regular Expressions	JLex scanner generator	a scanner in Java
BNF of a language	CUP parser generator	a parser in Java

The Challenge in Compiler Development
There are various challenges involved in developing compilers; some of these are itemized below:
Many variations:
a) many programming languages (e.g. FORTRAN, C++, Java)
b) many programming paradigms (e.g. object-oriented, functional, logic)
c) many computer architectures (e.g. MIPS, SPARC, Intel, alpha)
d) many operating systems (e.g. Linux, Solaris, Windows)
Qualities of a compiler: these concerns the qualities that are compiler must possess in other to be effective and useful. These are listed below in order of importance:
a) the compiler itself must be bug-free
b) it must generate correct machine code
c) the generated machine code must run fast
d) the compiler itself must run fast (compilation time must be proportional to programme size)
e) the compiler must be portable (i.e. modular, supporting separate compilation)
f) it must print good diagnostics and error messages
g) the generated code must work well with existing debuggers
h) must have consistent and predictable optimization.

Compiler Architecture
As earlier mentioned, a compiler can be viewed as a programme that accepts a source code (such as a Java programme) and generates machine code for some computer architecture. Suppose that you want to build compilers for *n* programming languages (e.g. FORTRAN, C, C++, Java, BASIC, etc.) and you want these compilers to run on *m* different architectures (e.g. MIPS, SPARC, Intel, alpha, etc.). If you do that naively, you need to write *n*m* compilers, one for each language architecture combination. The only grail of portability in compilers is to do the same thing by writing *n* + *m* program only. You can do this by using a universal *Intermediate Representation* (*IR*) and you make the compiler a two phase compiler. An IR is typically a tree-like data structure that captures the basic features of most computer architectures. One example of an IR tree node is a representation of a 3-address instruction, such as d $s1 + s2$ that gets two source addresses, $s1$ and $s2$, (i.e. two IR trees) and But the above ideal separation of compilation into two phases does not work very well for real programming languages and architectures.
Ideally, you must encode all knowledge about the source programming language in the front end, you must handle all machine architecture features in the back end, and you must design your IRs in such a way that all language and machine features are captured properly.

A typical real-world compiler usually has multiple phases. This increases the compiler's portability and simplifies retargeting. The front end consists of the following phases:

- **Scanning:** a scanner groups input characters into tokens
- **Parsing:** a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees* (ASTs)
- **Semantic analysis:** performs *type checking* (i.e. checking whether the variables, functions etc. in the source programme are used consistently with their definitions and with the language semantics) and translates ASTs into IRs
- **Optimization:** optimizes IRs. The back end consists of the following phases:
- **Instruction selection:** maps IRs into assembly code
- **Code optimization:** optimizes the assembly code using control flow and data-flow analyses, register allocation, etc.
- **Code emission:** generates machine code from assembly code. The generated machine code is written in an object file. This file is not executable since it may refer to external symbols (such as system calls). The operating system provides the following utilities to execute the code:
- **Linking:** A linker takes several object files and libraries as input and produces one executable object file. It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions/procedures and it resolves all external references to real addresses. The libraries include the operating system libraries, the language-specific libraries, and, maybe, user-created libraries.
- **Loading:** A loader loads an executable object file into memory, initializes the registers, heap, data, etc. and starts the execution of the programme.

Relocatable shared libraries allow effective memory use when many different applications share the same code produces one destination address, *d*. The first phase of this compilation scheme, called the front-end, maps the source code into IR, and the second phase, called the back-end, maps IR into machine code. That way, for each programming language you want to compile, you write one front-end only, and for each computer architecture, you write one back-end. So, totally you have *n* + *m* components.

THE STRUCTURE OF A COMPILER

We can identify four components

i) **Front-End: t**he front-end is responsible for the analysis of the structure and meaning of the source text. This end is usually the analysis part of the compiler. Here we have the syntactic analyzer, semantic analyzer, and lexical analyzer. This part has been automated.

ii) **Back-End:** The back-end is responsible for generating the target language. Here we have intermediate code optimizer, code generator and code optimizer. This part has been automated.

iii) **Tables of Information:** It includes the symbol-table and there are some other tables that provide information during compilation process.

iv) **Run-Time Library:** It is used for run-time system support.

Languages for Writing Compiler

a. Machine language
b. Assembly language
c. High level language or high level language with bootstrapping facilities for flexibility and transporting.

Phases of a Compiler

A compiler takes as input a source programme and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub-processes called phases. A phase is a logically cohesive operation that takes as input one representation of the source programme and produces as output another representation.

.

OPERATIONS OF A COMPILER

- The lexical analysis (or scanning) }
- The syntax analysis }Front end

- Semantic analysis }
- Intermediate code optimization }
- Code generation } Back-end
- Code optimization }

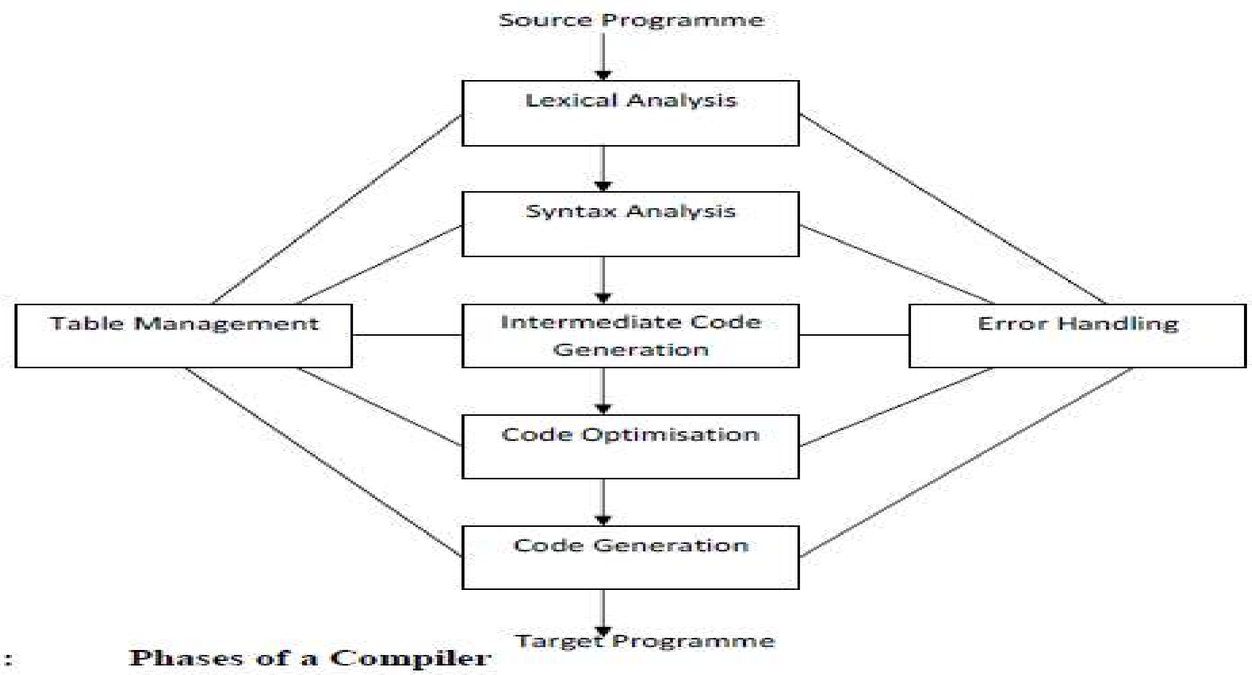
The Error Handler
This is invoked when a flaw in the source programme is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on flawed program, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routines interact with all phases of the compiler

Passes
In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases. The numbers of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine, rather than by any mathematical optimality criteria.

The structure of the source language has strong effect on the number of passes. Certain languages require at least two passes to generate code easily. For example, languages such as PL/I or ALGOL 68 allow the declaration of a name to occur after uses of that name. Code for expression containing such a name cannot be generated conveniently until the declaration has been seen.

Reducing the Number of Passes
Since each phase is a transformation on a stream of data representing an intermediate form of the source programme, you may wonder how several phases can be combined into one pass without the reading and writing of intermediate files. In some cases one pass produces its output with little or no memory of prior inputs. Lexical analysis is typical. In this situation, a small buffer serves as the interface between passes. In other cases, you may merge phases into one pass by means of a technique known as 'back patching'. In general terms, if the output of a phase cannot be determined without looking at the remainder of the phase's input, the phase can generate output with 'slots' which can be filled in later, after more of the input is read.

The Lexical Analyzer: this is the first phase and it is also referred to as the *Scanner*. It separates characters of the source language into groups that logically belong together; these groups are called *tokens*. The usual tokens are keywords, such as DO or IF, identifiers such as X or NUM, operator symbol such as <= or +, and punctuation symbol such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the *syntax analyzer* or *parser*. The tokens in this stream can be represented by codes which we may regard as integers. Thus DO might be represented by 1, + by 2, and "identifier" by 3. In the case of a token like "identifier", a second quantity, telling which of those identifiers used by the programme is represented by this instance of token "identifier" is passed along with the integer code for "identifier". For Example, in the FORTRAN statement: IF (5 .EQ. MAX) GO TO 100 we find the following eight tokens: IF; (; 5; .EQ; MAX;); GOTO; 100.



: Phases of a Compiler

The Syntax Analyzer: This groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a syntactic structure called an *expression*. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together. The parser has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language. It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

The Intermediate Code Generator: This uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and a small number of operands. These instructions with one operator and a small number of operands. These instructions can be viewed as simple macros like the macro ADD2. the primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.
Code Optimization: This is an optional phase designed to improve the intermediate code so that the ultimate object programme runs faster and/or takes less space. Its output is another intermediate code programme that does the same job as the original, but perhaps in a way that saves time and/or space.
Code Generation: This is the final phase and it produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object program is one of the most difficult parts of a compiler design, both practically and theoretically.

The Role of the Lexical Analyzer: The lexical analyzer could be a separate pass, placing its output on an intermediate file from which the parser would then take its input. But, more commonly, the lexical analyser and the parser are together in the same pass; the lexical analyzer acts as a subroutine or co-routine, which is called by the parser whenever it needs a new token. This organization eliminates the need for the intermediate file. In this arrangement, the lexical analyser returns to the parser a representation for the token it has found. The representation is usually an integer code if the token is a simple constructs such as left parenthesis, comma, or colon; it is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant. The integer code gives the token type, the pointer points to the value of that token.

The Need for Lexical Analyser
The purpose of splitting analysis of the source programme into two phases, lexical analysis and syntactic analysis, is to simplify the overall design of the compiler. This is because it is easier to specify the structure of a token than

the syntactic structure of the source program. Therefore, a more specialized and more efficient recognizer can be constructed for tokens than for syntactic structures.

By including certain constructs in the lexical rather than the syntactic structure, we can greatly simplify the design of the syntax analyser. Lexical analyser also performs other functions such as keeping track of line numbers, producing an output listing if necessary, stripping out white space (such as redundant blanks and tabs), and deleting comments.

HAND IMPLEMENTATION OF LEXICAL ANALYSER

In lexical analysis, the source program character by character and converge them to tokens. A token is the smallest unit recognizable by the compiler. There are basically a few numbers of tokens that are recognized. Generally, we have four classes of tokens that are usually recognized and they are:

Keywords Identifies Constants Delimiters

Construction of Lexical Analyser

There are 2 general ways to construct lexical analyser:

• Hand implementation

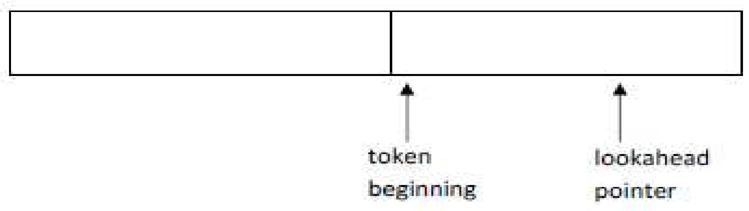
• Automatic generation of lexical analyser

Hand Implementation: There are two ways to use hand implementation:

• Input Buffer approach

• Transitional diagrams approach

Input Buffering: The lexical analyser scans the characters of the source programme one at a time to discover tokens. Often, however, many characters beyond the next token may have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyser to read its input from an input buffer. There are many schemes that can be used to buffer input but we shall discuss only one of these schemes here. Figure 1shows a buffer divided into two halves of, say, 100 characters. One pointer marks the beginning of the token being discovered. We view the position of each pointer as been between the character last read and the character next to be read. In practice, each buffering scheme adopts one convention; either a pointer is at the symbol last read or the symbol it is ready to read.



: **Input Buffer**

Transition Diagram (TD)

One way to begin the design of any programme is to describe the behaviour of the program by a flowchart. This approach is particularly useful when the programme is a lexical analyser, because the action taken is highly dependent on what character has been seen recently. Remembering previous characters by the position in a flowchart is a valuable too, so much so that a specialized kind of flowchart for lexical analyzers, called transition diagram, has evolved. In a TD, the boxes of the flowchart are drawn as circles and called states. The states are connected by arrows, called *edges*. The labels on the various edges leaving a state indicate the input characters that can appear after that state. In TD, we try to construct a TD for each token, and then link up.

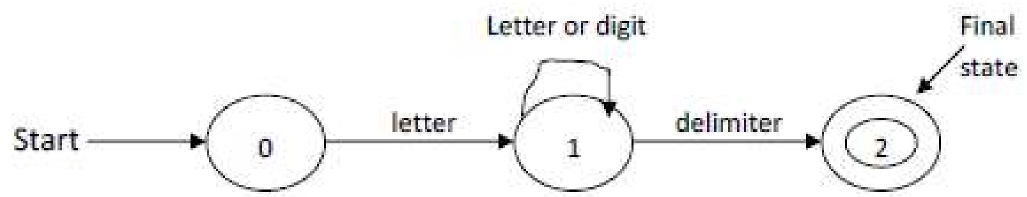


Fig. 2: Transition Diagram for Identifier

AUTOMATIC GENERATION OF LEXICAL ANALYSER

Automatic Generation of Lexical Analyser

There are tools that can generate lexical analyser for you. But before we begin the discussion of the design of a programme for generating lexical analyzers, you will first be introduced to a very useful notation, called regular expressions, suitable for describing tokens.

Language Theory Background

Symbol: (character, letter)
Alphabet: a finite nonempty set of characters. E.g. {0, 1}, ASCII, Unicode
String (sentence, word): a finite sequence of characters, possibly empty.
Language: a (countable) set of strings, possibly empty.

Operations on Strings

- **Concatenation:** concatenation of a string : for a string x of length m and another string y of length n, the concatenation of x and y is written as x y and it is the string obtained by appending y to string x e.g
  x = x1, x2, -------, xm
  y = y1, y2, ---------, yn
  xy = x1x2-----xmy1y2------yn
- **Exponentiation:** *x 0* is the empty string, *xi = xi-1x, for i > 0*
- **Kleene/Star Closure:** *L0* is { _ }, even when *L* is the empty set, *Li* = *Li*-1*L*, for *i* > 0. Note that *L** always contains the empty string. It is the set of all possible strings of all possible links defined over the alphabet *L* and its include the string o length 0 also known as empty string denoted by λ.
  $L^* = L0 \cup L1 \cup L2$ ---------- e.g. *L = {a, b}*

  Example 1: String = {a, aa, bb, aba, -----}
  L = {*λ, a, aa, bb, aba, ----} the different between language and Kleene closure is the λ sign for Kleene closure and if not then its ordinary language.
- **Positive Closure of a language:** this is written as L^+. it is the set of all possible string of all possible length defined over an alphabet *L* but exclude the string of length *0*. $L^+ = L^* - L0$.
  $L^+ = L0 \cup L1 \cup L2$ ---------- *L0*
  $L^+ = L0 \cup L1 \cup L2$ ----------

  Example 2:
  L = {a, aa, b, aba, -----}
  L^+ *= {a, aa, b, aba, -----}*

  Example 3: L = {0, 1, 11, 101, -----}
  L^* *= {λ , 0, 1, 11, 101, -----}*

  L^+ *= {0, 1, 11, 101, -----}*

Class Exercise: Give 5 samples things that can be generated from a Kleene closure and positive closure
L = {λ, a} w1, w2, w3, w4, ------------wn

Regular Expressions (REs)

Regular expressions are a very convenient form of representing (possibly infinite) sets of strings, called *regular sets*. For example, the RE (*a*| *b*)**aa* represents the infinite set {``*aa*",``*aaa*",``*baa*",``*abaa*", ... }, which is the set of all strings with characters *a* and *b* that end in *aa*. Many of today's programming languages use regular expressions to match patterns in strings. E.g., awk, flex, lex, java, javascript, perl, python.

Definition of a Regular Expression and the Language it Denotes

Basis

• _ is a regular expression that denotes { _ }.
• A single character *a* is a regular expression that denotes { *a* }.

Induction

Suppose *r* and *s* are regular expressions that denote the languages L(*r*) and L(*s*):
• (*r*)|(*s*) is a regular expression that denotes L(*r*) ∪ L(*s*)
• (*r*)(*s*) is a regular expression that denotes L(*r*)L(*s*)
• (*r*)* is a regular expression that denotes L(*r*)*
• (*r*) is a regular expression that denotes L(*r*).
We can drop redundant parenthesis by assuming:
• the Kleene star operator * has the highest precedence and is left associative
• concatenation has the next highest precedence and is left associative
• the union operator | has the lowest precedence and is left associative E.g., under these rules r|s*t is interpreted as (r)|((s)*(t)).

Extensions of Regular Expressions
• Positive closure: *r*+ = *rr**
• Zero or one instance: *r*? = _ | *r*
• Character classes:
a. [abc] = a | b | c
b. [0-9] = 0 | 1 | 2 | … | 9

FORMAL GRAMMAR

A formal grammar (sometimes called a grammar) is a set of rules of a specific kind, for forming strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar describes only the form of the strings and not the meaning or what can be done with them in any context. A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting must start. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer". Recognizer is a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognizers, formal language theory uses separate formalisms, known as ***Automata Theory***.

The process of recognizing an utterance (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language is referred to as **Parsing**. Most languages have the meanings of their utterances structured according to their syntax – a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down into parts and look at its analyzed form (known as its parse tree in Computer Science). A grammar mainly consists of a set of rules for transforming strings. (If it consists of these rules only, it would be a semi-Thue system). To generate a string in the language, one begins with a string consisting of a single start symbol. The *production rules* are then applied in any order, until a string that contains neither the start symbol nor designated *nonterminal symbols* is produced. The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.
Example
Assuming the alphabet consists of *a* and *b*, the start symbol is *S* and we have the following production rules:
1. S→aSb
2. S→ ba

then we start with *S*, and can choose a rule to apply to it. If we choose rule 1, we obtain the string *aSb*. If we choose rule 1 again, we replace *S* with *aSb* and obtain the string *aaSbb*. If we now choose rule 2, we replace *S* with *ba* and obtain the string *aababb*, and are done. We can write this series of choices more briefly, using symbols: S→aSb →aaSbb →aababb. The language of the grammar is then the infinite set $\{a^n bab^n \mid n \geq 0\}$ = {ba, abab, aababb, aaababbb, ------}, where a^k is *a* repeated *k* times (and *n* in particular represents the number of times production rule 1 has been applied).

The Syntax of Grammars
In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s, a grammar *G* consists of the following components:
o a finite set *N* of *non-terminal symbols*, none of which appear in

- strings formed from *G*.
  - a finite set _ of *terminal symbols* that is disjoint from *N*.
  - a finite set *P* of *production rules*, each rule of the form
  - where * is the Kleene star operator and denotes set union. That is, each production rule maps from one string of symbols to another, where the first string (the "head") contains at least one non-terminal symbol. In the case that the second string (the "body") consists solely of the empty string – i.e. it contains no symbols at all, it may be denoted with a special notation (often _, *e* or _) in order to avoid confusion.

A distinguished symbol, that is, the *start symbol*. A grammar is formally defined as the tuple (*N*, _, *P*, *S*). Such a formal grammar is often called a **rewriting system** or a **phrase structure grammar** in the literature A grammar can be formally written as four tuple element i.e G = {N, ∑, S, P}
where N is the set of variable or non-terminal symbols
∑ is a set of terminal symbol known as the alphabet
G = (N, ∑, S, P)
N → {S, A} where S is the start symbol
∑ = {λ, a}

Example 1: Consider the grammar G where N = {S, B} = {a, b, c}, S is the start symbol and P consists of the following production rules:
`1. S→ aBSc
2. S→ abc
3. Ba → aB
4. Bb → bb
This grammar defines the language L(G) = $a^nb^nc^n$ | n ≥ 1} where a^n denotes a string of n consecutive a's. Thus the language is the set of strings that consist of one or more a's, followed by the same number of b's and then by the same number of c's.
Solution:
S $\rightarrow_{2}$ **abc**
S $\rightarrow_{1}$ **aBSc** $\rightarrow_{2}$ aB**abc**c $\rightarrow_{3}$ a**aB**bcc $\rightarrow_{4}$ aa**bb**cc
S $\rightarrow_{1}$ **aBSc** $\rightarrow_{1}$ aB**aBSc**c $\rightarrow_{2}$ aBaB**abc**cc $\rightarrow_{3}$ a**aB**Babaccc $\rightarrow_{3}$ aaB**aB**bccc $\rightarrow_{3}$ aa**aB**Bbccc $\rightarrow_{4}$ aaaB**bb**ccc $\rightarrow_{4}$ aaa**bb**bccc

Example 2: Consider G = ({S, A, B}, {a, b}, S, P) N = {S, A, B} produce 2 sample strings from the grammar and a^2b^4 is contained in the grammar?
1. P: S → AB
2. A → Aa
3. B → Bb
4. A → a
5. B → b

Solution:
S $\rightarrow_{1}$ **AB**
S $\rightarrow_{1}$ **AB** $\rightarrow_{2}$ **Aa**B $\rightarrow_{3}$ Aa**Bb** $\rightarrow_{3}$ Aa**Bb**b $\rightarrow_{3}$ Aa**Bb**bb $\rightarrow_{4}$ **a**aBbbb $\rightarrow_{5}$ aa**b**bbb

Example 3: Analyze the structure of this grammar G = {V, T, S, P} where V is used in place of N and T is used in place of ∑ and produce 4 sample strings from the grammar?
G = (V, T, S, P), V = (A, S, B), T = (λ, a, b)
1. P: S → ASB
2. A → a
3. B → b
4. S → λ

Solution
S $\rightarrow_{4}$ **λ**
S $\rightarrow_{1}$ **ASB** $\rightarrow_{2}$ **a**SB $\rightarrow_{3}$ aS**b** $\rightarrow_{4}$ ab
S $\rightarrow_{1}$ **ASB** $\rightarrow_{4}$ AB

$\mathbf{S} \to_1 \mathbf{ASB} \to_2 \mathbf{a}SB \to_4 aB$

Exercise: Obtain a gramma that generates a language $L(G) = \{a^n b^{n+1} : n \geq 0\}$

Types of Grammars and Automata

In the field of Computer Science, there are four basic types of grammars:

a. Type-0 grammars (unrestricted grammars) include all formal grammars.
b. Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages.
c. Type-2 grammars (context-free grammars) generate the context free languages.
d. Type-3 grammars (regular grammars) generate the regular languages.

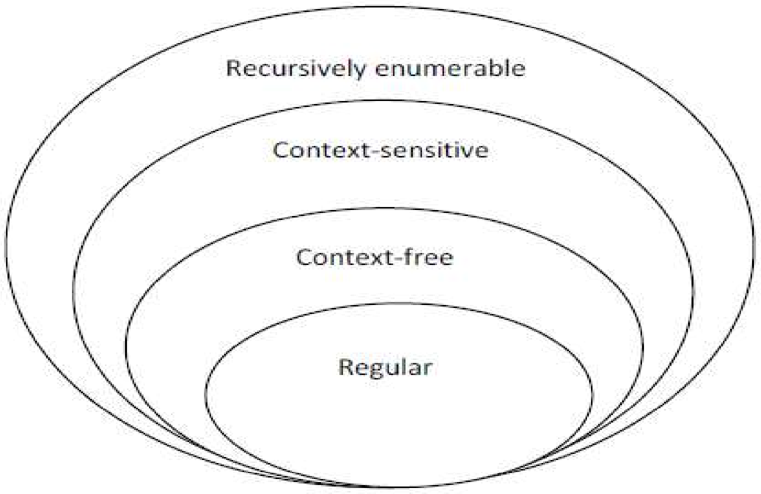
The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. Two important types are *context-free grammars* (Type 2) and *regular grammars* (Type 3). The languages that can be described with such a grammar are called *context-free languages* and *regular languages*, respectively. Although much less powerful than unrestricted grammars (Type 0), which can in fact express any language that can be accepted by a Turing machine, these two restricted types of grammars are most often used because parsers for them can be efficiently implemented.

Grammar Type	Grammar Accepted	Language Accepted	Automation
Type 0	Unrestricted grammar	Recursively enumerable language needs counters, registers, selection and memory to recognized the grammar	Turing machine $(a^n c^m b^n)^k$
Type 1	Context sensitive grammar	Context sensitive language needs counter, register and selection to recognized the grammar	Linearly bounded $a^n c^m b^n$
Type 2	Context free grammar	Context free language needs a counter and register to recognized the grammar	Push down automata $(ab)^n = a^n b^n$
Type 3	Regular expression	Regular expression, regular language only need counters to recognize grammar	Finite state automaton $(ab)^2 = ab$

Increase in strict production rules and decrease in expression of formal language

Chomsky Hierarchy

The Chomsky hierarchy (occasionally referred to as **Chomsky– Schützenberger hierarchy**) is a containment hierarchy of classes of formal grammars. This hierarchy of grammars was described by Noam Chomsky in 1956. It is also named after Marcel-Paul Schützenberger who played a crucial role in the development of the theory of formal languages. Note that the set of grammars corresponding to recursive languages is not a member of this hierarchy. Every regular language is context-free, every context-free language, not containing the empty string, is context-sensitive and every context sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages which are not context sensitive, context-sensitive languages which are not context-free and context-free languages which are not regular. The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.



FINITE AUTOMATA

Finite state system represents a model of a system with input and output. The input given to the system may be processed into various intermediate state then to a final state. The finite state system is very good designing tools for program such as text editor and lexical analyser. Compiler makes use lexical analyzer and text editor are Microsoft word. A finite automate is a five tuple machine define as FA = {ϴ, ∑, δ, q_0, F}.
ϴ: this is a set of non-empty finite state
∑: this is a finite set of symbol called the alphabet
δ: is the transition or mapping function where δ: ϴ x ∑ → ϴ
$q_{0:}$ is the initial state from where any input is processed and (q_0 Є ϴ
F: is a set of final state / states and F ≤ ϴ
A recognizer for a language *L* is a program that takes as input a string *x* and answers "yes" if *x* is a sentence of *L* and "no" otherwise. Clearly, the part of a lexical analyser that identifies the presence of a token on the input is a recognizer for the language defining that token. Variants of finite automata are commonly used to match regular expression patterns.

Nondeterministic Finite Automaton (NFA)

A nondeterministic finite automaton (NFA) consists of:
- A finite set of states *S*
- An input alphabet consisting of a finite set of symbols _
- A transition function _ that maps *S* × (_ ∪ {_}) to subsets of *S*.

This transition function can be represented by a transition graph in which the nodes are labelled by states and there is a directed edge labelled *a* from node *w* to node *v* if _(*w*, *a*) contains *v*
- An initial state *s*0 in *S*
- *F*, a subset of *S*, called the final (or accepting) states. An NFA accepts an input string *x* if there is a path in the transition graph from the initial state to a final state that spells out *x*. The language defined by an NFA is the set of strings accepted by the NFA.

Deterministic Finite Automata (DFAs)

A deterministic finite automaton (DFA) is an NFA in which:
- There are no _ moves, and
- For each state *s* and input symbol *a* there is exactly one transition out of *s* labelled *a*. Therefore, a DFA represents a finite state machine that recognizes a RE. For example, the following DFA:

Equivalence of Regular Expressions and Finite Automata

Regular expressions and finite automata define the same class of languages, namely the regular sets. In Computer Science Theory we showed that:

- Every regular expression can be converted into an equivalent NFA using the McNaughton-Yamada-Thompson algorithm.
- Every NFA can be converted into an equivalent DFA using the subset construction.
- Every finite automaton can be converted into a regular expression using Kleene's algorithm.

Converting a Regular Expression to an NFA

The task of a scanner generator, such as JLex, is to generate the transition tables or to synthesize the scanner programme given a scanner specification (in the form of a set of REs). So it needs to convert Res into a single DFA. This is accomplished in two steps: first it converts REs into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

An NFA, as earlier stated, is similar to a DFA but it also permits multiple transitions over the same character and transitions over _. In the case of multiple transitions from a state over the same character, when we are at this state and we read this character, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The transition does not consume any input characters, so you may jump to another state for free. Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blow-up in the number of states.

NFA for each RE Characteristics

As it can been shown inductively, the above rules construct NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE *AB*, we construct the NFAs for *A* and *B*, which are represented as two boxes with one start state and one final state for each box. Then the NFA for *AB* is constructed by connecting the final state of *A* to the start state of *B* using an empty transition

Converting a Regular Expression into a Deterministic Finite Automaton

To convert a RE into a DFA, you first convert the RE into an NFA as discussed. The next step is to convert the NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set {5, 6, 8}. This indicates that arriving to the state labelled {5, 6, 8} in the DFA is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input
(Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states).
First, we need to handle transitions that lead to other states for free (without consuming any input). These are the transitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more transitions.

The Scanner

A *scanner* groups input characters into tokens. For example, if the input is:
x = x*(b+1);

then the scanner generates the following sequence of tokens
id(x)
=
id(x)

*
(
id(b)
+
num(1)
)
;
where id(x) indicates the identifier with name x (a programme variable in this case) and num(1) indicates the integer 1. Each time the parser needs a token, it sends a request to the scanner. Then, the scanner reads as many characters from the input stream as it is necessary to construct a single token. The scanner may report an error during scanning (e.g. when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the scanner is suspended and returns the token to the parser. The parser will repeatedly call the scanner to read all the tokens from the input stream or until an error is detected (such as a syntax error).
Tokens are typically represented by numbers. For example, the token * may be assigned number 35. Some tokens require some extra information. For example, an identifier is a token (so it is represented by some number) but it is also associated with a string that holds the identifier name. For example, the token id(x) is associated with the string, "x". Similarly, the token num(1) is associated with the number, 1.
Tokens are specified by patterns, called *regular expressions*. For example, the regular expression [a-z][a-zA-Z0-9]* recognises all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.
A typical scanner:
• recognises the *keywords* of the language (these are the reserved words that have a special meaning in the language, such as the word class in Java);
• recognises special characters, such as (and), or groups of special characters, such as := and ==;
• recognises identifiers, integers, reals, decimals, strings, etc;
• ignores whitespaces (tabs and blanks) and comments;
• recognises and processes special directives (such as the #include "file" directive in C) and macros.
A key issue is speed. One can always write a naive scanner that groups the input characters into lexical words (a lexical word can be either a sequence of alphanumeric characters without whitespaces or special characters, or just one special character), and then tries to associate a token (i.e. number, keyword, identifier, etc.) to this lexical word by performing a number of string comparisons. This becomes very expensive when there are many keywords and/or many special lexical patterns in the language. In this unit you will learn how to build efficient scanners using regular expressions and finite automata. There are automated tools called *scanner generators*, such as *flex* for C and *JLex* for Java, which construct a fast scanner automatically according to specifications (regular expressions). You will first learn how to specify a scanner using regular expressions, then the underlying theory that scanner generators use to compile regular expressions into efficient programmes (which are basically finite state machines), and then you will learn how to use a scanner generator for Java, called JLex.