UNIT-1: Compiler Design

Introduction to Compiler:

A *compiler* is a program that translates a source program written in some high-level programming language (such as C, C++, Java etc.) into machine code for some computer architecture (such as the Intel Pentium architecture, Sparc assembly language etc.). Since computer can understand only in binary numbers so a compiler is used to fill the gap otherwise it would have been difficult for a human to find info in the 0 and 1 form. The generated machine code can be later executed many times against different data each time.

Of course, one expects a compiler to do a faithful translation, i.e. the meaning of the translated text should be the same as the meaning of the source text.

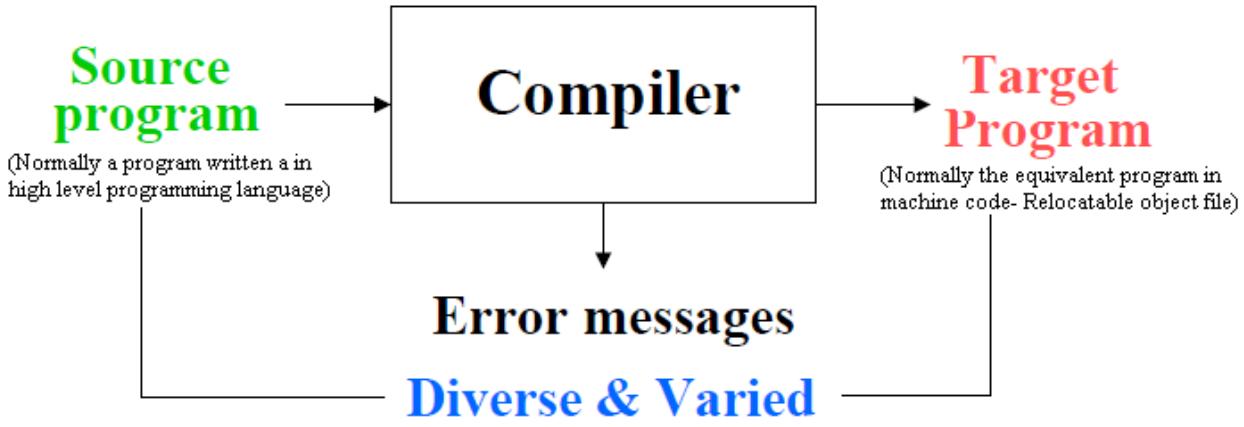


Figure1:

Source code is normally optimized for human readability:

1. Expressive: Matches our notion of languages(and application)
2. Redundant to help avoid programming errors.

Machine code is optimized for hardware:

1. Redundancy is reduced
2. Information about the intent is lost

There are two parts of compilation;

1. Analysis
2. Synthesis

The ***analysis part*** breaks up the source program into constant piece and creates an intermediate representation of the source program. e.g.: Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.

The ***synthesis part*** constructs the desired target program from the intermediate representation. e.g.: Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Applications of compilers

1. Traditionally, a compiler is thought of as translating a so-called "high level language" such as C or Modula2 into assembly language. Since assembly language cannot be directly executed, a further translation between assembly language and (relocatable) machine language is necessary. Such programs are usually called assemblers but it is clear that an assembler is just a special (easier) case of a compiler.
2. Sometimes, a compiler translates between high level languages. E.g. the first C++ implementations used a compiler called "cfront" which translated C++ code to C code. Such a compiler is often called a "cross compiler".
3. A compiler need not target a real assembly (or machine) language. E.g. Java compilers generate code for a virtual machine called the "Java Virtual Machine" (JVM). The JVM interpreter then interprets JVM instructions without any further translation.
4. Compilers (and interpreters) have wider applications than just translating programming languages. Conceivably any large and complex application might define its own "command language" which can be translated to a virtual machine associated with the application.
5. Compilation techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
6. Compilation techniques used in a parser can be used in a query processing system such as SQL.
7. Much software having a complex front-end may need techniques used in compiler design.
8. Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Compiler is part of program development environment; some other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc. The compiler (and all other tools) must support each other for easy program development.

Language processing system by using compiler:

Whenever language is processed, complete processing of language is not dependent on the compiler. So complete processing of language is dependent on Assembler, Linker, and Loader etc. Language processing system process shown in below;

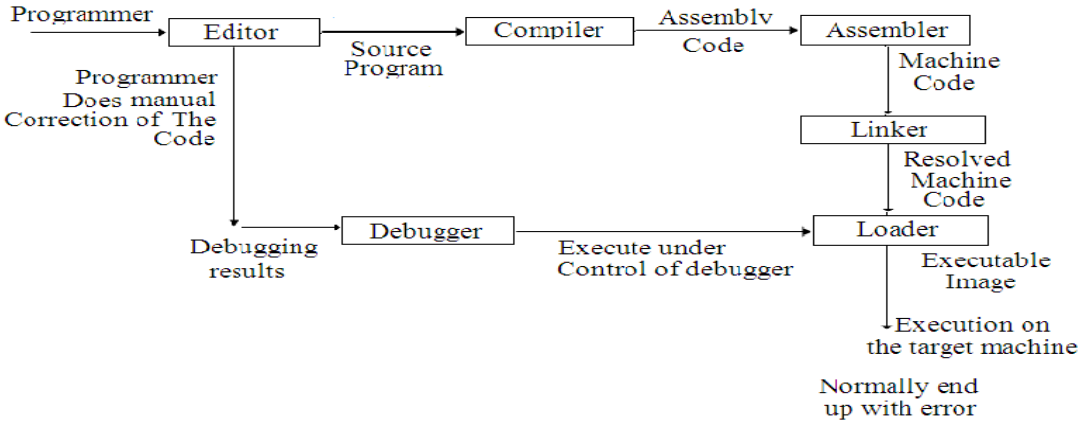


Figure:

Assemblers:

Assembly code is a mnemonic version of machine code in which names, rather than binary values, are used for machine instructions, and memory addresses.

Some processors have fairly regular operations and as a result assembly code for them can be fairly natural and not-too-hard to understand.

Linkers:

Linkers, also known as linkage editors combine the output of the assembler for several different compilations. The linker has another input, namely libraries, but to the linker the libraries look like other programs compiled and assembled. The two primary tasks of the linker are

1. Relocating relative addresses.
2. Resolving external references (such as the procedure xor() above).

Relocating relative addresses:

The assembler processes one file at a time. Thus the symbol table produced while processing file A is independent of the symbols defined in file B, and conversely. Thus, it is likely that the same address will be used for different symbols in each program. The technical term is that the (local) addresses in the symbol table for file A are *relative* to file A; they must be *relocated* by the linker. This is accomplished by adding the starting address of file A (which in turn is the sum of the lengths of all the files processed previously in this run) to the relative address.

Resolving external references:

Assume procedure f, in file A, and procedure g, in file B, are compiled (and assembled) separately. Assume also that f invokes g. Since the compiler and assembler do not see g when processing f, it appears impossible for procedure f to know where in memory to find g.

The solution is for the compiler to indicate in the output of the file A compilation that the address of g is needed. This is called a *use* of g. When processing file B, the compiler outputs the (relative) address of g. This is called the *definition* of g. The assembler passes this information to the linker.

The simplest linker technique is to again make two passes. During the first pass, the linker records in its external symbol table (a table of external symbols, not a symbol table that is stored externally) all the definitions encountered. During the second pass, every use can be resolved by access to the table.

Loaders:

After the linker has done its work, the resulting executable file can be loaded by the operating system into central memory. The details are OS dependent. With early single-user operating systems all programs would be loaded into a fixed address (say 0) and the loader simply copies the file to memory. Today it is much more complicated since (parts of) many programs reside in memory at the same time. Hence the compiler/assembler/linker cannot know the **real** location for an identifier. Indeed, this real location can change.

Interpreter:

Interpreter is a program that translates an instruction into a machine language and executes it before proceeding to the next instruction, a high level programming language translator that translates and runs the program at the same time. It translates one program statement into machine language, executes it, and then proceeds to the next statement. This differs from regular executable programs that are presented to the computer as binary-coded instructions. Interpreted programs remain in the source language the programmer wrote in, which is human readable text. Interpreted programs run slower than their compiler

counterparts. Whereas the compiler translates the entire program before it is run, interpreters translate a line at a time while the program is being run. However, it is very convenient to write an interpreted program, since a single line of code can be tested interactively. One example is the UNIX shell interpreter, which runs operating system commands interactively.

Difference between compiler and interpreter:

1. A complier converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
2. Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
3. List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
4. An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

Phases and Passes of Compiler:

Compiler operates in phases, each of which transforms the source program from one representation to another.

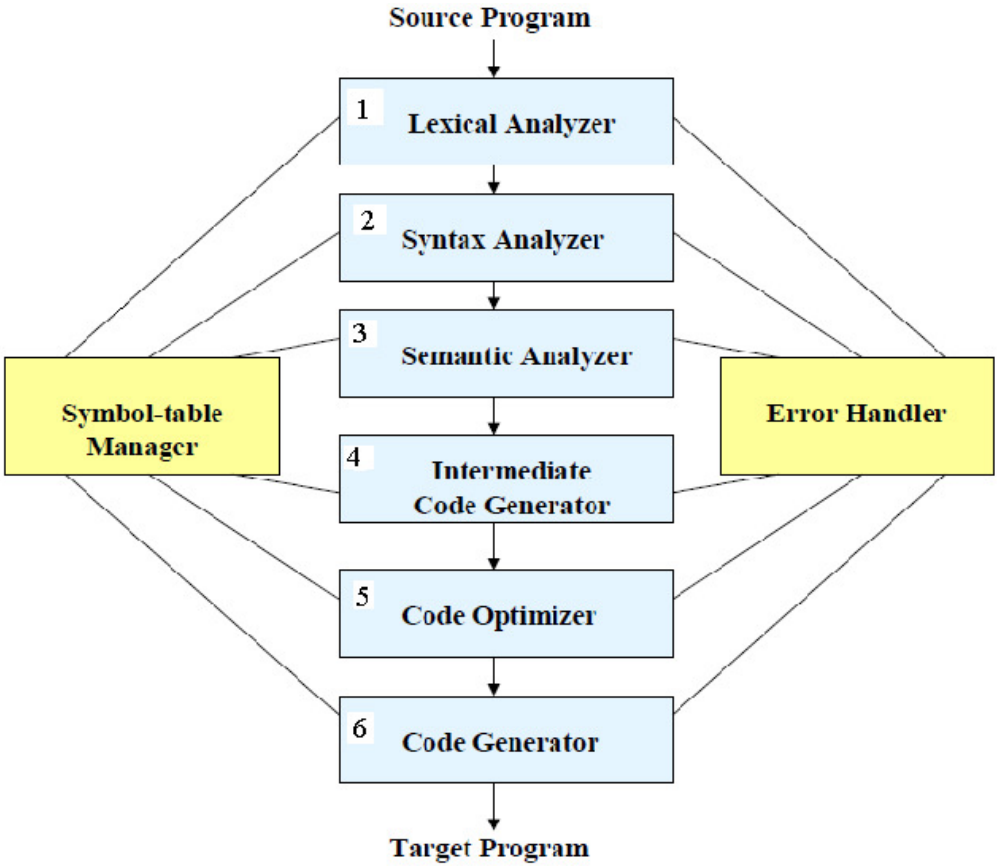


Figure: Phases of Compiler (Block diagram of compiler)

1. **Lexical Analyzer (or scanning):**

The first job of a compiler is to group sequences of raw characters into meaningful tokens. The lexical analyzer module is responsible for this job. The character stream input is grouped into meaningful units called **lexemes**, which are then mapped into **tokens**, the latter constituting the output of the lexical analyzer. In addition, a lexical analyzer will typically access the symbol table to store and/or retrieve information on certain source language concepts such as variables, functions, types. Lexical Analyzer ignores comments or white spaces. A token is a pair of <token-name, attribute-value>.

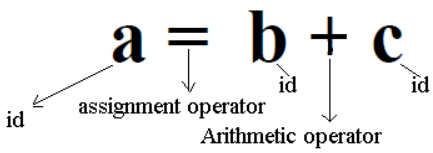


Figure:

For example **X3 := y + 3**

1. The lexeme X3 would be mapped to a token such as <id, 1>. The name id is short for identifier. The value 1 is the index of the entry for x3 in the symbol table produced by the compiler. This table is used gather information about the identifiers and to pass this information to subsequent phases.
2. The lexeme: = would be mapped to the token **< :=>**. In reality it is probably mapped to a pair, whose second component is ignored. The point is that there are many different identifiers so we need the second component, but there is only one assignment symbol: =.
3. The lexeme y is mapped to the token <id, 2>
4. The lexeme + is mapped to the token <+>.
5. The lexeme 3 is somewhat interesting and is discussed further in subsequent chapters. It is mapped to <number, something>, but what is the something. On the one hand there is only one 3 so we could just use the token <number, 3>. However, there can be a difference between how this should be printed (e.g., in an error message produced by subsequent phases) and how it should be stored (fixed vs. float vs. double). Perhaps the token should point to the symbol table where an entry for this kind of 3 is stored. Another possibility is to have a separate numbers table.
6. The lexeme; is mapped to the token < ;>.

2. **Syntax Analyzer:**

Once lexical analysis is finished, then parser takes over to check whether the sequence of tokens is grammatically correct, according to the rules that define the syntax of the source language.

Position: = id * rate + 60

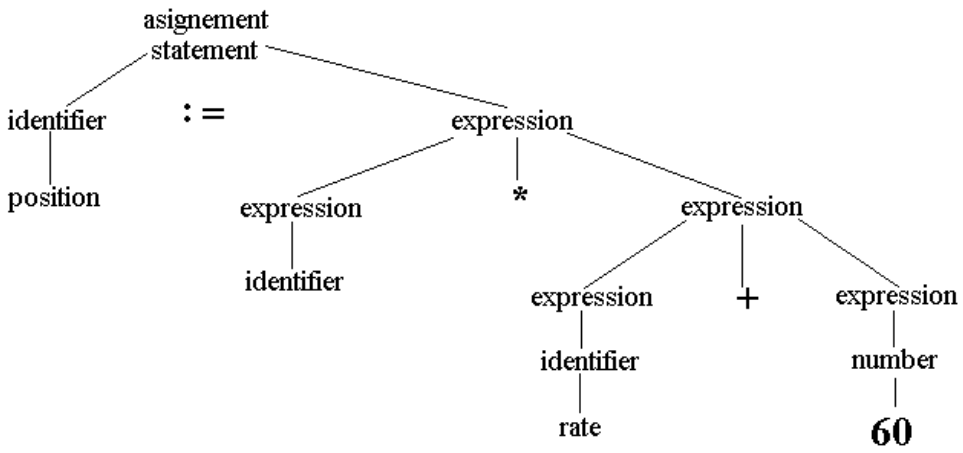


Figure:

3. Semantic Analyzer:

Having established that the source text is syntactically correct, the compiler may now perform additional checks such as determining the type of expressions and checking that all statements are correct with respect to the typing rules, that variables have been properly declared before they are used, that functions are called with the proper number of parameters etc. This phase is carried out using information from the parse tree and the symbol table.

4. Intermediate code generation:

In this phase, the compiler translates the source text into a simple intermediate language. There are several possible choices for an intermediate language. Intermediate code generates a program for an abstract machine. Sometimes it is also called as times 3-address code which is like assembly language for a machine in which every memory location acts like register.

Example: **a = b + c * d / f**

Solution:

Intermediate code for above example

T1 = c * d

T2 = T1 / f

T3 = b + T2

A = T3

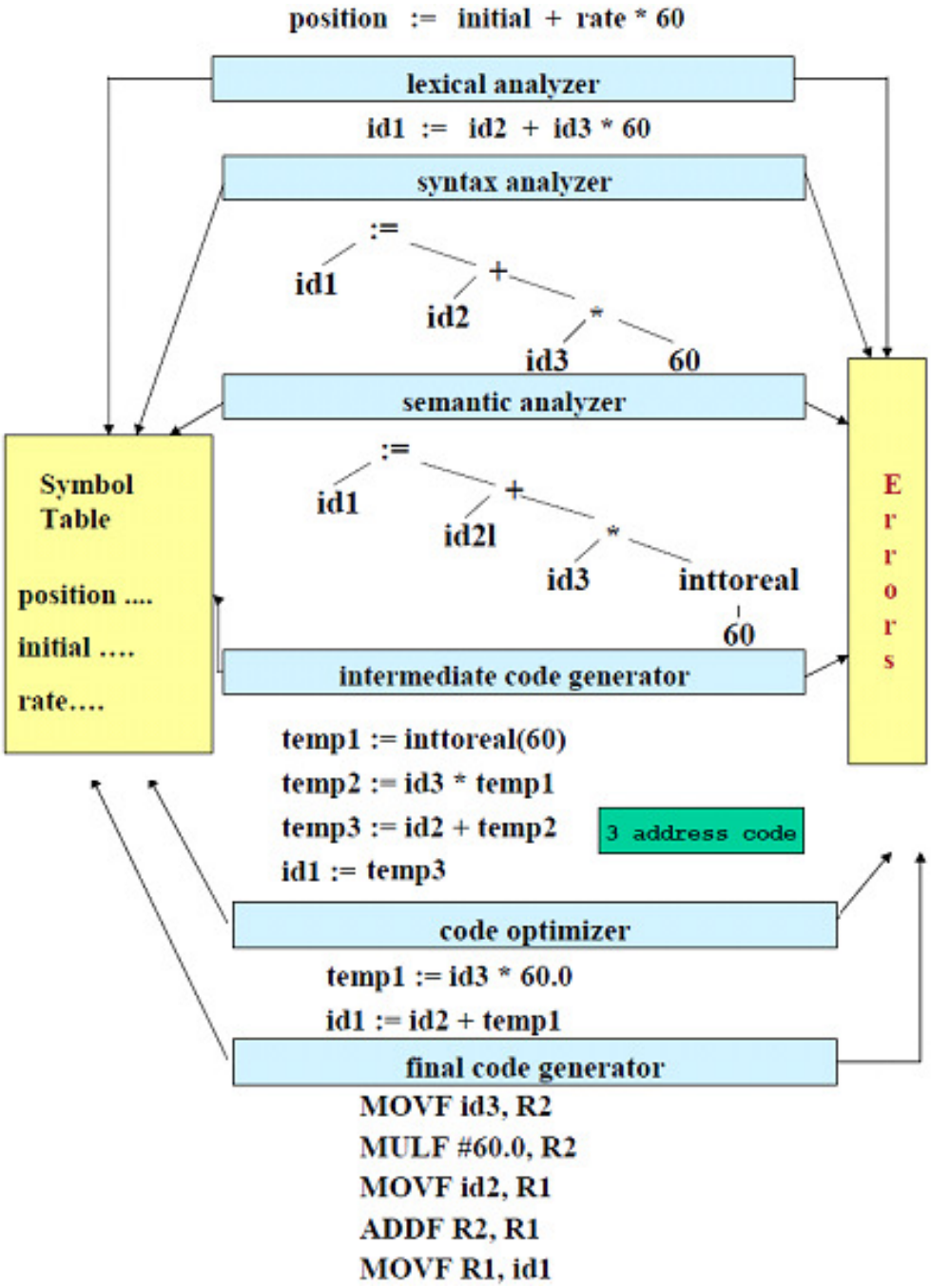
5. Code Optimization:

In this phase, the compiler tries several optimization methods to replace fragments of the intermediate code text with equivalent but faster (and usually also shorter) fragments.

6. Code generation:

The final phase of the compilation consists of the generation of target code from the intermediate code. When the target code corresponds to a register machine, a major problem is the efficient allocation of scarce but fast registers to variables.

Step by Step entire Process of Compiler phases:



Cross- Compiler and Bootstrapping Process:

A cross-compiler is a compiler that runs on one machine and produces object code for another machine. The cross-compiler is used to implement the compiler, which is characterized by three languages:

1. The Source Language,
2. The Target Language (object language),
3. Implementation Language.

If a compiler has been implemented in its own language, then this arrangement is called a "bootstrap" arrangement.

Implementing a Bootstrap compiler:

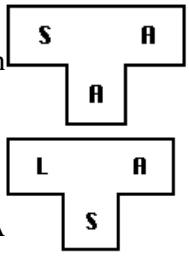
Notation: ${}^{S}C_{I}^{T}$ represents a compiler for Source *S*, Target *T*, implemented in language *I*. The *T-diagram* shown above is also used to depict the same compiler.

Compilers are of two kinds;

1. Native Compiler
2. Cross Compiler

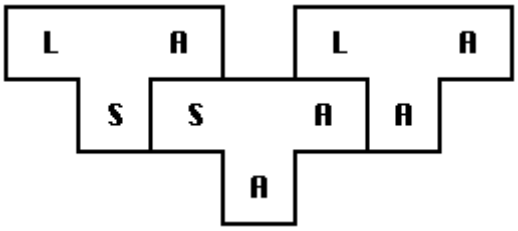
1. ***Native Compiler:*** Native compilers are written in the same langage as the target language. E.g. SMM is a compiler for the language S that is in a language that runs on machine M and generates output code that runs on machine M.

Steps involving to create a new language, L, for machine A (or Native Compiler):



1. Create ${}^{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)
2. Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.
3. Compile ${}^{L}C_{S}^{A}$ using ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$, a compiler for language L, which runs on machine A and produces code for machine A.

$${}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A}$$



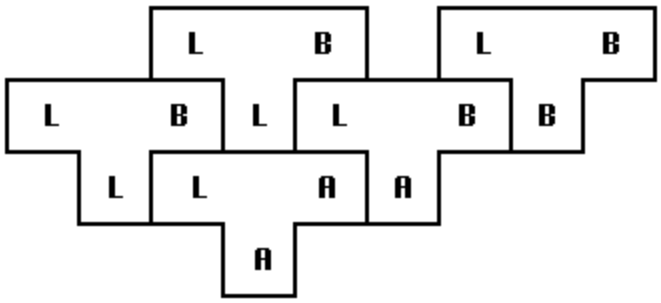
The process illustrated by the T-diagrams is called *bootstrapping* and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

2. ***Cross Compiler:*** Cross compilers are written in different language as the target language. E.g. SNM is a compiler for the language S that is in a language that runs on machine N and generates output code that runs on machine M.

Steps involving to produce a compiler for a different machine B (or cross compiler):

1. Convert ${}^{L}C_{S}^{A}$ into ${}^{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset of language L.
2. Compile ${}^{L}C_{L}^{B}$ to produce ${}^{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A and produces code for machine B.
3. Compile ${}^{L}C_{L}^{B}$ with the cross-compiler to produce ${}^{L}C_{B}^{B}$, a compiler for language L which runs on machine B.



Finite state machine:

A *finite automaton* is an abstract machine that serves as a recognizer for the strings that comprise a regular language. The idea is that we can feed an input string into a finite automaton, and it will answer "yes" or "no" depending on whether or not the input string belongs to the language that the automaton recognizes.

A finite auto-mata consists of a finite number of states and a finite number of transition, and these transitions are defined on certain, specific symbols called input symbols. A finite automate (M) is defined by using five (5) tuples that are;

$$M = (Q, \sum, \delta, q_0, F)$$

Where;

- **Q** is a finite non empty set of states,
- $\sum$ is a finite non empty set of input alphabet (symbols)
- **δ** is a transitions functions in the automata.
- i.e. ; $\delta : Q * \sum \rightarrow Q$
- q_0 is the initial state (i.e.: q_0 is a subset of Q)
- **F** is the set of final states (i.e.: F is a subset of Q)

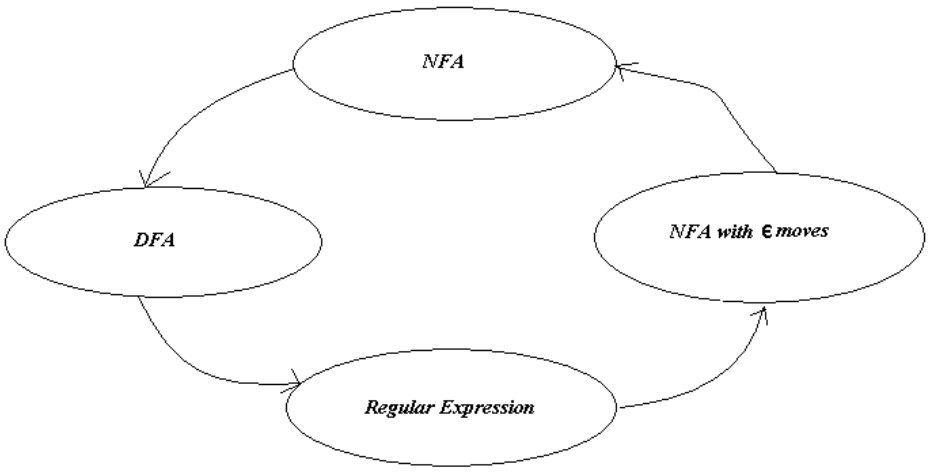


Figure:

Application of Finite state machine and regular expression in Lexical analysis:

Lexical analysis is the process of reading the source text of a program and converting that source code into a sequence of tokens. The approach of design a finite state machine by using regular expression is so useful to generates token form a given source text program.

Since the lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analysis is to;

1. Specify regular expressions for all of the kinds of tokens in the language. The disjunction of all of the regular expressions thus describes any possible token in the language.
2. Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA).
3. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.

To recognize identifiers, numerals, operators, etc., implement a DFA in code.

State is an integer variable, δ is a *switch* statement

Upon recognizing a lexeme returns its lexeme, lexical class and restart DFA with next character in source code.

Application of formal grammar in syntax analysis:

A formal *grammar* is a formal system that describes a language by specifying how any legal text can be derived from a distinguished symbol called the *axiom*, or *sentence symbol*. A syntax analyzer phase of a compiler has the responsibility to check the formation rules of the given grammar. A syntax analyzer phase does not describe the syntactically correctness of the token generated by the lexical analyzer. So to analysis a string is by the rules defined by the language is working by the formal grammar either CFG or BNF.

A **formal grammar** (sometimes simply called a **grammar**) is a set of formation rules for 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.

Introduction to Lex & Yacc:

A compiler or an interpreter for a programming language is often decomposed into two parts:

1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

Lex and *Yacc* can generate program fragments (either tokens or hierarchical structure) that solve the first task. The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (*Lex*).
2. Find the hierarchical structure of the program (*Yacc*)

LEX (LEX Compiler):

Lex is officially known as a "Lexical Analyser". Lex (Lexical analyser) helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine. The main job of Lex is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

For example, if we are writing a compiler for the C programming language,

1. The symbols `{ } ( ) ;` all have significance on their own.
2. The letter `'a'` usually appears as part of a keyword or variable name, and is not interesting on it's own. Instead, we are interested in the whole word.
3. Spaces and newlines are completely uninteresting, and we want to ignore them completely, *unless* they appear within quotes `"like this"`.

All of these things are handled by the Lexical Analyzer.

Structure of LEX program;

{ ---defination--- }

%%

{ ---rules--- }

%%

{ ---user subroutines--- }

Lexical analyzer generator:

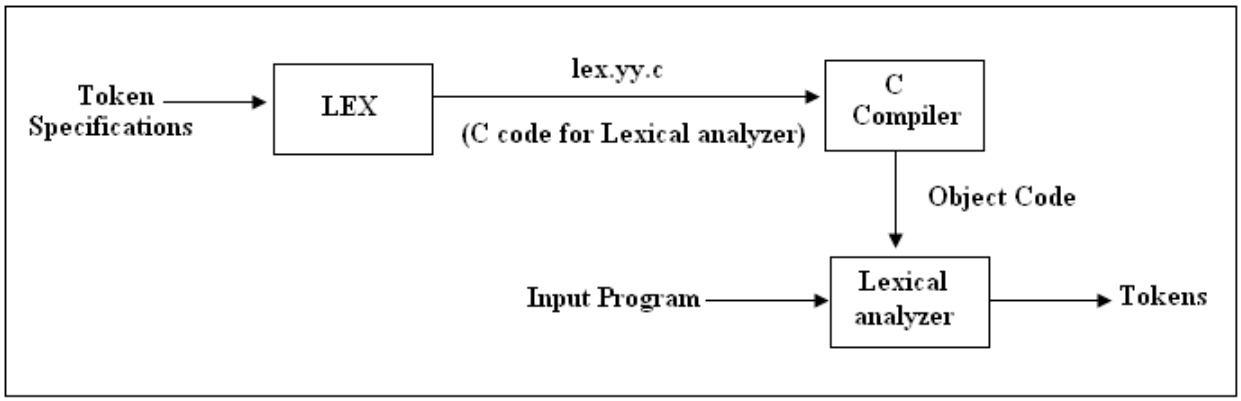


Figure:

LEX is one such lexical analyzer generator which produces C code, based on the token specifications. LEX tool has been widely used to specify lexical analyzers for a variety of languages. We refer to the tool as Lex Compiler, and to its input specification as the Lex language. Lex is generally used in the manner depicted in the above figure;

1. First, a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language.
2. Then, the lex.l is run through the Lex compiler to produce a C program lex.yy.c . The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of the lex.l, together with a standard routine that uses the table to recognize lexemes. The actions associated with the regular expressions in lex.l are pieces of C code and are carried over directly to lex.yy.c.
3. Finally, lex.yy.c is run through the C compiler to produce an object programa.out which is the lexical analyzer that transforms the input stream into a sequence of tokens.

Working process of LEX;

1. Input to the generator
2. List of regular expressions in priority order (Regular expressions describe the languages that can be recognized by finite automata.)
3. Associated actions for each of regular expression(generates kind of token and other book keeping information)
4. Translate each token regular expression into a non deterministic finite automaton (NFA).
5. Convert the NFA into an equivalent DFA.
6. Minimize the DFA to reduce number of states.
7. Emit code driven by the DFA tables.
8. Output of the generator
9. Repots lexical errors (unexpected characters), if any

We assume that we have a specification of lexical analyzers in the form of regular expression and the corresponding action parameters. Action parameter is the program segments that are to be executed

whenever a lexeme matched by regular expressions is found in the input. So, the input to the generator is a list of regular expressions in a priority order and associated actions for each of the regular expressions. These actions generate the kind of token and other book keeping information. Our problem is to construct a recognizer that looks for lexems in the input buffer. If more than one pattern matches, the recognizer is to choose the longest lexeme matched. If there is two or more pattern that matches the longest lexeme, the first listed matching pattern is chosen. So, the output of the generator is a program that reads input character stream and breaks that into tokens. It also reports in case there is lexical error i.e. either unexpected characters occur or an input string doesn't match any of the regular expressions.

YACC:

YACC is officially known as a "parser". YACC stands for "Yet Another Compiler Compiler". The main job of YACC is to analyse the structure of the input stream, and operate of the "big picture". A YACC is usefull to builds a bottom up (shift reduce) syntax analyser from a context free grammar.

Structure of YACC:

%{ ---declaration (Optional)
%}
%%
Rules (Compulsory)
%%

A YACC generate a file to C compiler is named as 'y.tab.c'.

Backus-Naur Form:

BNF is an acronym for "Backus-Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language.

BNF Notation:

1. BNF is basically just a variant of a context-free grammar (CFG), with the symbol ``::=" used in place of ``→" to mean ``is defined as". Additionally, non-terminals are usually written in angle-brackets and terminals in quotes.

For example;

<block> ::= "BEGIN" <opt-stats> "END".

<opt-stats> ::= <stats-list> | ε.

<stats-list> ::= <statement> | <statement> ";" <stats-list>.

2. An extension of this notation, known as **extended BNF** (EBNF), allows us to use regular-expression-like constructs on the right-hand-side of the rules: for any sequence of symbols α, we

write "[α]" to say that α is optional, and "{ α }" to denote 0 or more repetitions of α (i.e. the Kleene closure of α).

Thus we could re-write the above rules in EBNF as:

<block> ::= "BEGIN" [<stats-list>] "END".

<stats-list> ::= {<statement> ";"} <statement>.

Using EBNF has the advantage of simplifying the grammar by reducing the need to use recursive rules.

We could describe the complete syntax of a programming language using EBNF (or equivalently a CFG), but it tends to make things easier if we break it up into two sections:

1. The lexical syntax, which defines which patterns of characters make up which words. This can be described using regular expressions.
2. The context-free syntax, which defines which sequences of words which make up phrases from the programming language; we describe this using a context-free grammar.

Thus the first task in writing a parser for a language is breaking up its syntax into the ``regular" and ``context-free" part. The choice is arbitrary but a simple guideline is that:

1. Characters from the input should not appear in the context-free syntax; these should be tokenized by the lexical phase.

As well as modularizing our task, this approach will facilitate debugging in parser generators such as *yacc*.

Classification of chomsky normal form:

Noam Chomsky defined four classes of languages:

Type of Grammar	*Restrictions on G = (N,T,P,S)*	*Advantages and Disadvantages*
Phrase Structure (Type 0) (Turing Machine)	No further restrictions	Can describe any language but mathematically proveable that no general recognition algorithms exist.
Context Sensitive (Type 1) (Linear Bound Automata)	Rules only take the form $x \rightarrow y$ where $\|x\| \leq \|y\|$	All programming language features can be described with these grammars, and general recognition algorithms exist but they are inefficient.
Context Free (Type 2) (CFG, BNF)	Rules only take the form $A \rightarrow y$ where $A \in N$ and $y \in (N \cup T)^*$	Efficient general recognition methods exist and this type is widely used in practice. However the matching of use of variables and their declaration can't be expressed in this grammar. Neither can type matching be expressed.
Regular Grammars (Type 3) (Finite Automata)	Rules only take the form $A \rightarrow \varepsilon$ or $A \rightarrow Ba$ where $A, B \in N$ and $a \in T$. The start set must satisfy $S \subseteq$	Sequential recognition methods exist with either left to right or right to left scan. These grammars can only describe language features having a sequential structure, e.g., numbers or identifiers. Nested tree-

	N.	like structures can't be described, e.g., block structure, expressions, if then else statements, etc.

Context free grammar:

A context free grammar consists of terminals, non-terminals, a start symbol and productions i.e. (CFG) G = (T, N, S, P).

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. Non-terminals are syntactic variables that denote sets of strings that help define language generated by the grammar. They impose a hierarchical structure on the language.
3. In a grammar one non-terminal is distinguished as the start symbol, and the sets of string it denotes is the language denoted by the grammar.
4. The productions of a grammar specify the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal followed by an arrow (→) followed by a string of non-terminals and terminals.

E.g.:

E → EAE | (E) | -E | id
A→ + | - | * | / |

Where E, A are the non-terminals while id, +, *, -, /, (,) are the terminals.

Derivations:

It's a way to view the process by which a grammar defines a language. The central idea here is that a production is treated as a rewriting rule in which non terminal on left is replaced by the string on right side of the production.

For eg: grammar for arithmetic expressions, with non terminal E representing an expression.

E→ E+E | E*E | (E) | -E | **id**

We can replace a single E by -E. We can describe this action by writing

E→ -E

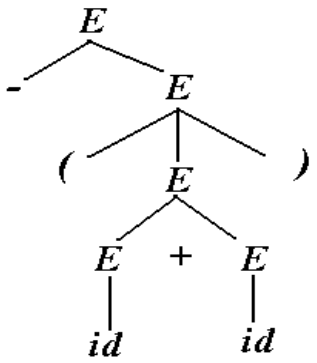
Which is read "E derives -E". We can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements.

For example E → -E → -(E) → -(**id**)

We call such a sequence of replacements a derivation of -(id)from E. This provides poof that a particular instance of an expression is the string -(**id**).

Parse trees and Derivation

A parse tree may be viewed as a a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labelled by some non-terminal A, and the children of the node are labelled from left to right by the symbols in the right side of the production by which A was replaced in the derivation .The leaves of the parse tree are labelled by non-terminals or terminals and read from left to right, they constitute a sentential form called the yield or frontier of the tree. For example the parse tree for -(**id +id**) implied is shown below,



Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous, Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence. Carefully writing the grammar can eliminate ambiguity.

Elimination of Left Recursion:

A grammar is left recursion if it has a non-terminal A such that there is a derivation A → Aa for some string a. Top-Down parsing method cannot handle left-recursion grammars, so a transformation that eliminates left recursion is needed. A left-recursive pair of productions A → A a could be replaced by the non-left-recursive productions

$$A \rightarrow bA'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as

$$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \ldots\ldots\ldots A\alpha m \mid \beta 1 \mid \beta 2 \mid \ldots\ldots\ldots \mid \beta n$$

Where no 'b_i' begins with an A. Then, we replace the A-production by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots\ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots\ldots \alpha_m A' \mid e$$

The non-terminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all immediate left recursion from the A and A' productions, but it does not eliminate left recursion involving derivation of two or more steps. For example, consider the grammar

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid e$$

The non-terminal S is left-recursive because S gives Aa gives Sda, but it is not immediately left recursive. Algorithm below will systematically eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles or ε productions
Cycles can be systematically eliminated from a grammar as can - ε productions.
Algorithm Eliminating left recursion.

Input: Grammar with no cycles or ε-productions.
Output: An equivalent grammar has no left recursion.
Method: Apply the algorithm to G. Note that the resulting non-left-recursive grammar may have ε-productions.

1. Arrange the non-terminals in some order A1, A2,.....,An.

2. **for** i := 1 to *n* **do begin**
   for j: = 1 to i - 1 **do begin**
   replace each production of the form $A_i \rightarrow A_{j\gamma}$
   by the productions $\mathbf{A_i \rightarrow \delta_{1\gamma}| \delta_{2\gamma}.....| \delta_{k\gamma}}$
   where $A_j \rightarrow \mathbf{\delta_1}| \mathbf{\delta_2}|.....| \mathbf{\delta_k}$ are all current A_j productions.
   end
   eliminate the immediate left recursion among Ai productions
   end

Left Factoring

Left factoring is a grammar transformation that is useful producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non terminal A, we may rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice. If $A \rightarrow \alpha\beta_1|\alpha\beta_2$ are two A-productions and the input begins with a non empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. We may defer decision by expanding A to αA'. Then after seeing the input derived from a we expand A' to $\beta_1$or to β_2

$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1| \beta_2$

Algorithm: left factoring a grammar

Input: Grammar G
Output: An equivalent left factored grammar.
Method: For each non terminal A find the longest prefix a common to two or more of its alternatives. If a!= E, i.e., there is a non trivial common prefix, replace all the A productions

$A \rightarrow \alpha\beta_1| \alpha\beta_2|.....| \alpha\beta_n|\gamma$, where g represents all alternatives that do not begin with a by
$A \rightarrow \alpha A'|\gamma$
$A' \rightarrow \beta_1| \beta_2.....|\beta_n$
Here A' is new non-terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.