UNIT 3

Syllabus: *TM: Definition of TM, Structural representation of TM, Construction of TM*

Compiler: *Definition of Compiler, Phases of Compiler, Lexical Analysis, Input Buffering.*

Turing Machine

Topics to be Discussed:

→ Introduction

→ Configuration

→ Definition

→ REC & Recursive

→ Construction of TM

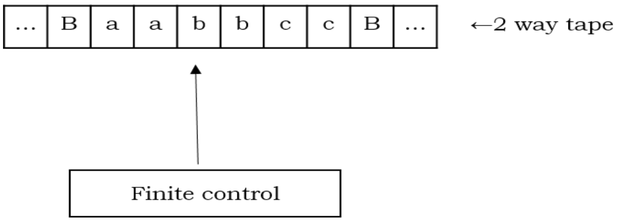
Introduction

→ Mathematical Model of General-Purpose Computer

→ Languages accepted by TM are called REL.

→ Consists of input tape of Infinite length, read-write head and a finite control.

Configuration



- Can also be said FA + [RW head] + [left | Right Direction]
- Can also be said Two stack FA.
- PDA = FA + 1 stack, 7 tuple system

Definitions

A TM is a 7-tuple system

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

DTM → $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \{ L, R \}$

NTM → $\delta: Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$

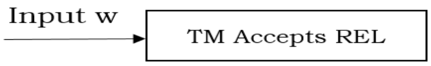
Variants of TM

- One-way Infinite Tape TM
- Two-way Infinite Tape TM
- Multitape Infinite Tape TM
- Multitape and Multihead
- Multidimensional TM
- Universal TM
- 2 stack PDA
- FA + 2 stack

==

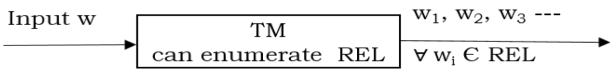
Functions of TM

1. Acceptor: (TM accepts REL) → transducer.



- If $w \in REC$, it always halts at final state.
- If $w \notin REL$, it may halt at non final state or it may never halt (Enters into loop).

2. Enumerator: It enumerates the language.

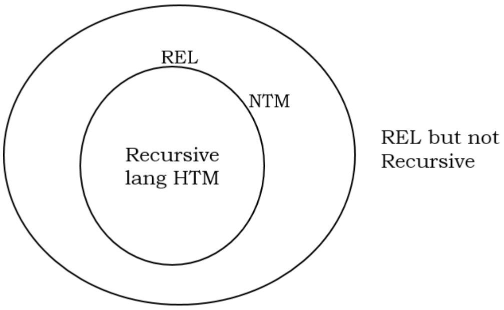


==

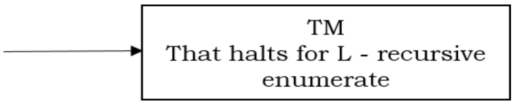
TM that halts at FS

Halting TM

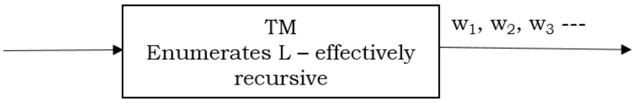
1. Final state + Halt → accepts $w \in L$ (TM)
2. Non-final + Halt → Not accept $w \notin L$ (TM)
3. ∞- loop or No Halt → can't conclude $w \in L$ (TM)
   $w \notin L$ (TM)



HTM:

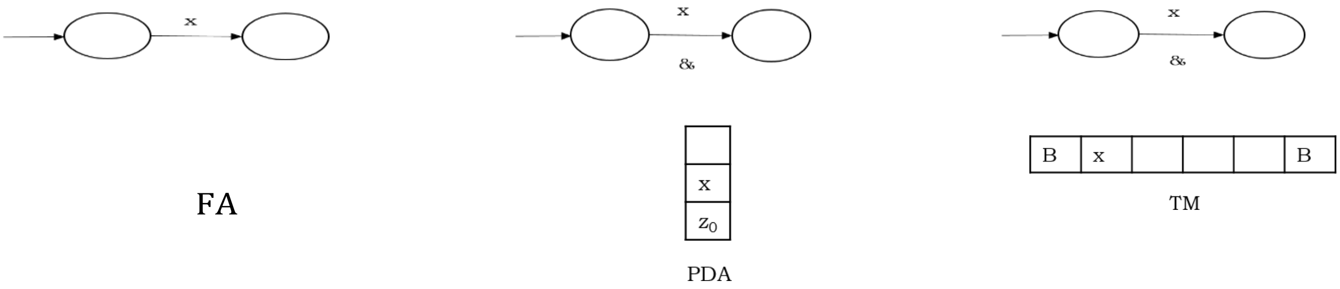


- if $w \in L$ halts at final state
- If $w \notin L$ halts at non final



Effective order lexicographical order.

Construction of TM: How to remember a symbol

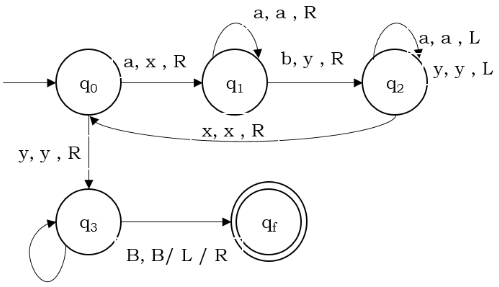


==

Practice Questions

1. Construct a TM for L = {$a^n b^n$ | n >= 1}

B	$\cancel{a}$	$\cancel{a}$	$\cancel{a}$	$\cancel{b}$	$\cancel{b}$	$\cancel{b}$	$\cancel{B}$

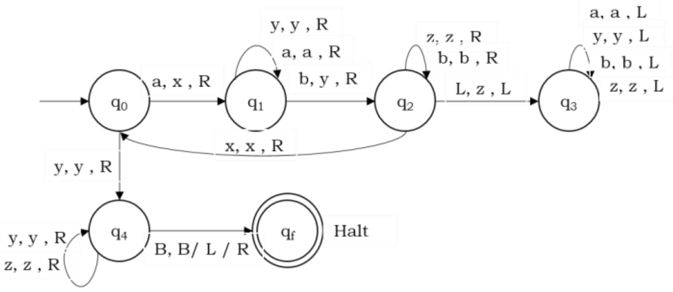


No of states required: 5

==

2. Construct a TM for L = {$a^n b^n c^n$ | n >= 1} → PDA does not accept.

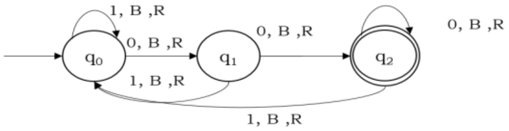
B	$\cancel{a}$	$\cancel{a}$	$\cancel{a}$	$\cancel{b}$	$\cancel{b}$	$\cancel{b}$	$\cancel{z}$	$\cancel{z}$	$\cancel{z}$	B



Number of of states required: 6.

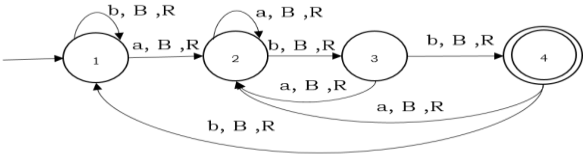
==

3. L = {Ending with 00}



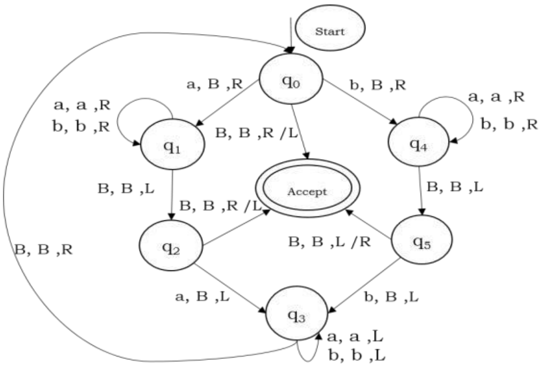
==

4. L = {Ending with abb}



==

5. TM that accepts palindromes of strings over the alphabet ∑ = {a, b}

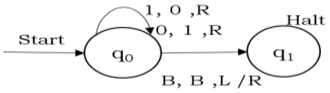


Q: # states for even palindrome ---------------------

Q: # states for odd palindrome ----------------------

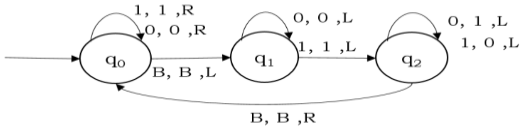
==

6. 1’s complement of a binary number.



==

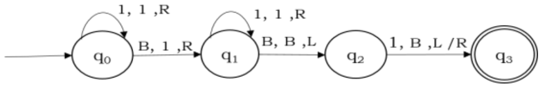
7. 2’s compliment of a binary number.



# 3 states

==

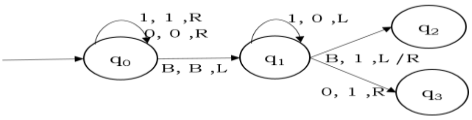
8. TM for unary adder.



# of states.

==

9. TM for the increment of a binary number.



1

B	B	1	0	1	B

2

B	B	1	0	0	B

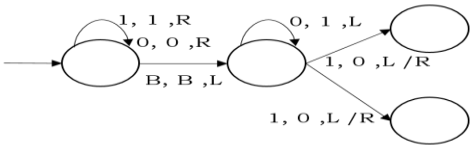
3

B		1	1	1	B

# States required.

==

10. TM for the decrement of a binary number.



1

B	1	0	0	0	0	B

2

B	1	1	1	1	1	B

3

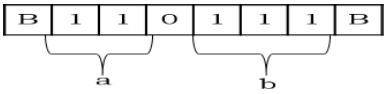
B	1	0	1	0	1	B

Q: The input string is 10001

Q: # states required to decrement the w = 1000

========================== ==

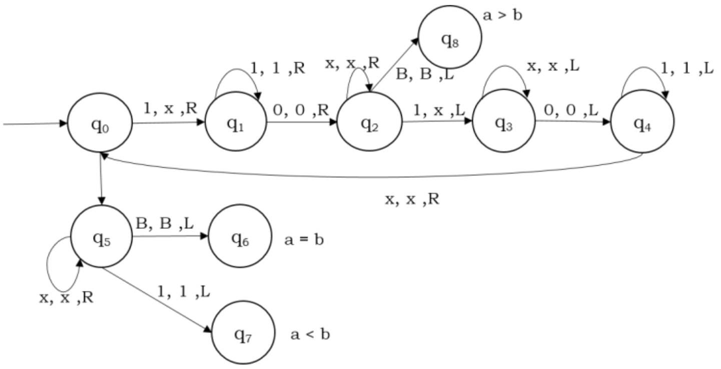
11. TM as a comparator.



a = b?

a > b?

a < b?



# States for w_1 c w_2: 7

==

12. If TM head is unidirectional, such TM is equivalent to FA such TM accepts Regular language.

13. If TM tape is read only then such TM is equivalent to FA.

14. If TM tape is read only and head is unidirectional, such TM is equivalent to RL

15. IF TM tape is bounded then such TM is called LBA such TM accepts CSL.

==

Compilers

Compiler is a program which translates a program written in one language (the source language) to an equivalent program in other language (the target language). The source language usually is a high-level language like Java, C, Fortran etc. whereas the target language is machine code that a computer's processor understands.

The source language is optimized for humans. It is more user-friendly, to some extent platform-independent. They are easier to read, write and maintain. Hence, it is easy to avoid errors. Ultimately, programs written in a high-level language must be translated into machine language by a compiler. The target machine language is efficient for hardware but lacks readability.

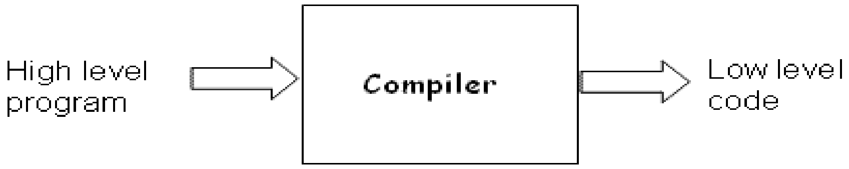
- Translates from one representation of the program to another.
- Typically, from high level source code to low level machine code or object code.
- Source code is normally optimized for human readability.
- Machine code is optimized for hardware.
- Redundancy is reduced.
- Information about the intent is lost.

Goals of translation

Good performance for generated code: The metric for the quality of the generated code is the ratio between the size of handwritten code and compiled machine code for same program. A better compiler is one which generates smaller code. For optimizing compilers this ratio will be lesser.

Good compile time performance: A handwritten machine code is more efficient than a compiled code in terms of the performance it produces.

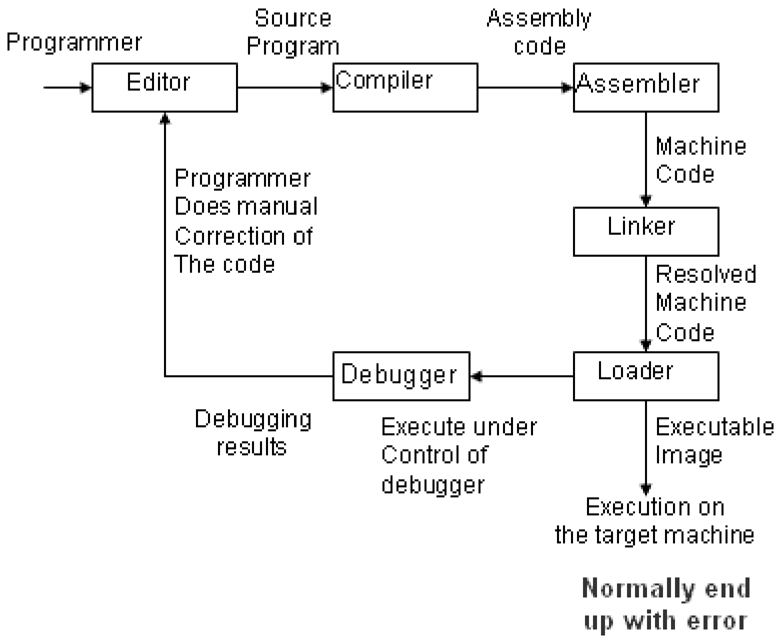
Correctness: A compiler's most important goal is correctness - all valid programs must compile correctly.



Pictorial Representation

- Compiler is part of program development environment
- The 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

All development systems are essentially a combination of many tools. For compiler, the other tools are debugger, assembler, linker, loader, profiler, editor etc. If these tools have support for each other than the program development becomes a lot easier.



This is how the various tools work in coordination to make programming easier and better. They all have a specific task to accomplish in the process, from writing code to compiling it and running / debugging it. If debugged then do manual correction in the code if needed after getting debugging results. It is the combined contribution of these tools that makes programming a lot easier and efficient.

Translation Procedure

- In order to translate a high-level code to a machine code, it needs to go step by step, with each step doing a particular task and passing its output for the next step in the form of another program representation.
- The steps can be parse tree generation, high level intermediate code generation, low level intermediate code generation and then the machine language conversion.
- As the translation proceeds the representation becomes more and more machine specific, increasingly dealing with registers, memory locations etc.
- Each step handles a reasonably simple, logical, and well-defined task.
- Intermediate representations should be a program manipulation of various kinds (type checking, optimization, code generation etc.).
- Representations become more machine specific and less language specific as the translation proceeds.

The first few steps

The first few steps of compilation like lexical, syntax and semantic analysis can be a natural language like. The first step in understanding a natural language will be to recognize characters, i.e. the upper and lower-case alphabets, punctuation marks, alphabets, digits, white spaces etc.

Similarly, the compiler has to recognize the characters used in a programming language. Similarly, the programming language have a dictionary as well as rules to construct words (numbers, identifiers etc).

- The first step is recognizing alphabets of a language. For example

- English text consists of lower and upper-case alphabets, digits, punctuations and white spaces
- Written programs consist of characters from the ASCII characters set.
- The next step to understand the sentence is recognizing words (lexical analysis)
- English language words can be found in dictionaries
- Programming languages have keywords etc. and rules for constructing words (identifiers, numbers etc.)

PHASES OF A COMPILER:

Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. Therefore a typical Compiler is having the following Phases:

1. Lexical Analyzer (Scanner),
2. Syntax Analyzer (Parser),
3. Semantic Analyzer,
4. Intermediate Code Generator(ICG),
5. Code Optimizer(CO) , and
6. Code Generator(CG).

In addition to these, it also has **Symbol table management**, and **Error handler** phases.
The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.
The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
The analysis part is often called the front end of the compiler; the synthesis part is the back end.
If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.
A typical decomposition of a compiler into phases is shown in Fig. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly .

PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely different representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

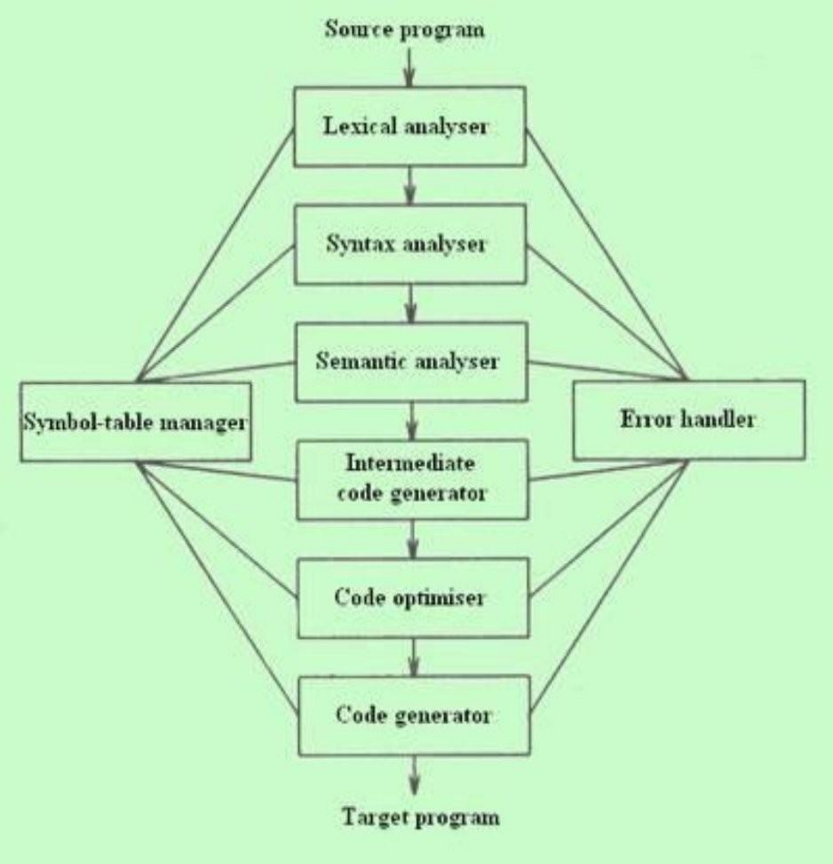


Figure : Phases of a Compiler

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

- Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .
- The character sequence forming a token is called a **lexeme** of thetoken.
- The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn‘texist.
- Also removes the Comments, and unnecessaryspaces.
  - The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves aretokens.
- The interior node of this tree represents the stream of tokens that logicallybelongs together.
- It means it checks the syntax of program elements.

SEMANTICANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct,it may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

- The Syntactically and Semantically correct structures are produced here in the form ofa Syntax tree or DAG or some other sequential representation likematrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- It should be easy to produce,and Easy to translate into the target program.Example intermediate code formsare:
- Three address codes,
- Polish notations,etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

→Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductionsetc.

- Sometimes the data structures used in representing the intermediate forms may also be changed.

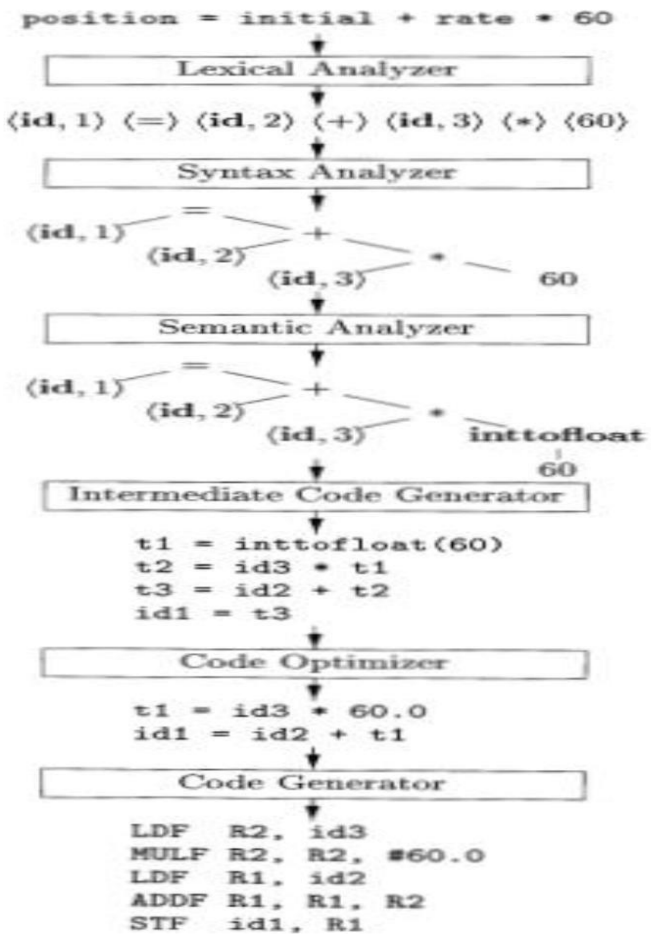
CODE GENERATOR:
This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

- Memory locations are selected for each variable used, and assignment of variables to registers is done.
- Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

Fig:Translation of assignment statement for **Position=initial+rate*60**



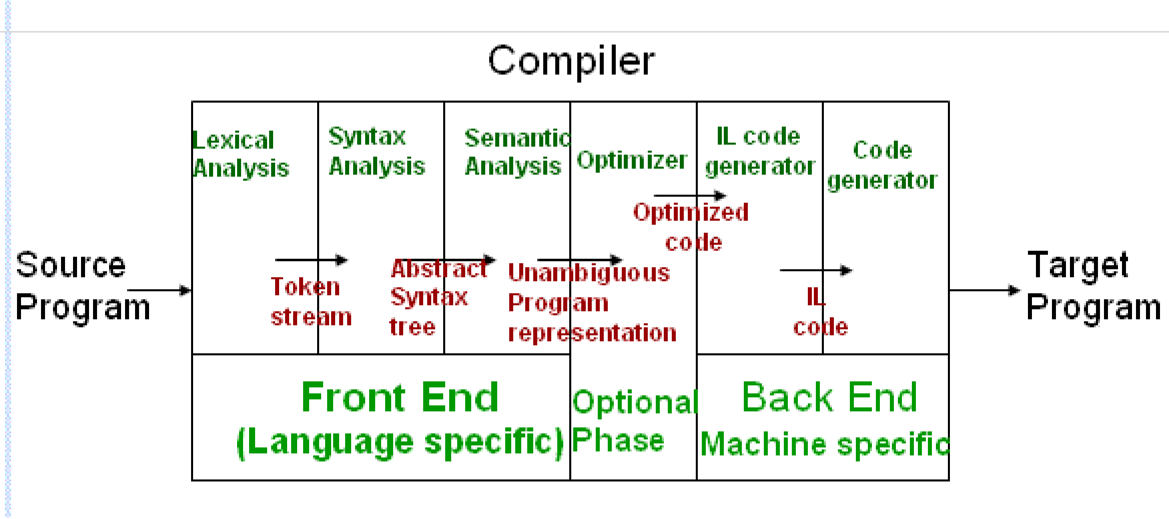
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

The input source program is **Position=initial+rate*60**

Compiler structure

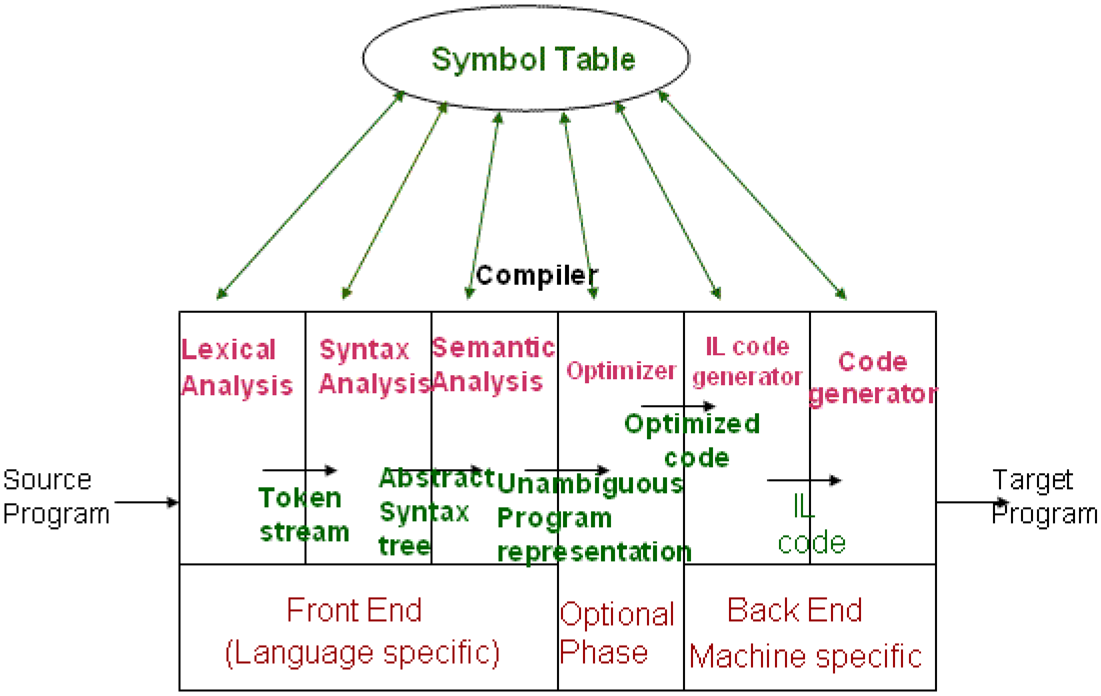


These are the various stages in the process of generation of the target code from the source code by the compiler. These stages can be broadly classified into the Front End (Language Specific) and the Back End (Machine specific) parts of the compilation.
Information required about the program variables during compilation

For the lexicons, additional information with its name may be needed. Information about whether it is a keyword/identifier, its data type, value, scope, etc might be needed to be known during the latter phases of compilation. However, all this information is not available in a straight away.

This information has to be found and stored somewhere. We store it in a data structure called Symbol Table. Thus, each phase of the compiler can access data from the symbol table & write data to it. The method of retrieval of data is that with each lexicon a symbol table entry is associated. A pointer to this symbol in the table can be used to retrieve more information about the lexicon.

Final Compiler structure



This diagram elaborates that each stage can access the Symbol Table. All the relevant information about the variables, classes, functions etc. are stored in it.

It is also known as Analysis-Synthesis model of compilation.

- Front end phases are known as analysis phases.

- Back end phases are known as synthesis phases.

Each phase has a well-defined work.

Each phase handles a logical activity in the process of compilation

The front-end phases are Lexical, Syntax and Semantic analyses. These form the "analysis phase" as you can well see these all do some kind of analysis.

The Back-End phases are called the "synthesis phase" as they synthesize the intermediate and the target language and hence the program from the representation created by the Front-End phases.

Advantages of the model.

- ❖ Compiler is retargetable
- ❖ Source and machine independent code optimization is possible.
- ❖ Optimization phase can be inserted after the front and back end phases have been developed and deployed.
- ❖ It is also known as Analysis-Synthesis model of compilation

Issues in Compiler Design

- Compilation appears to be very simple, but there are many pitfalls

- How are erroneous programs handled?
- Design of programming languages has a big impact on the complexity of the Compiler M*N vs. M+N problem
- Compilers are required for all the languages and all the machines
- For M languages and N machines we need to develop M*N compilers
- However, there is lot of repetition of work because of similar activities in the front ends and back ends

Specifications and Compiler Generator

How to write specifications of the source language and the target machine?

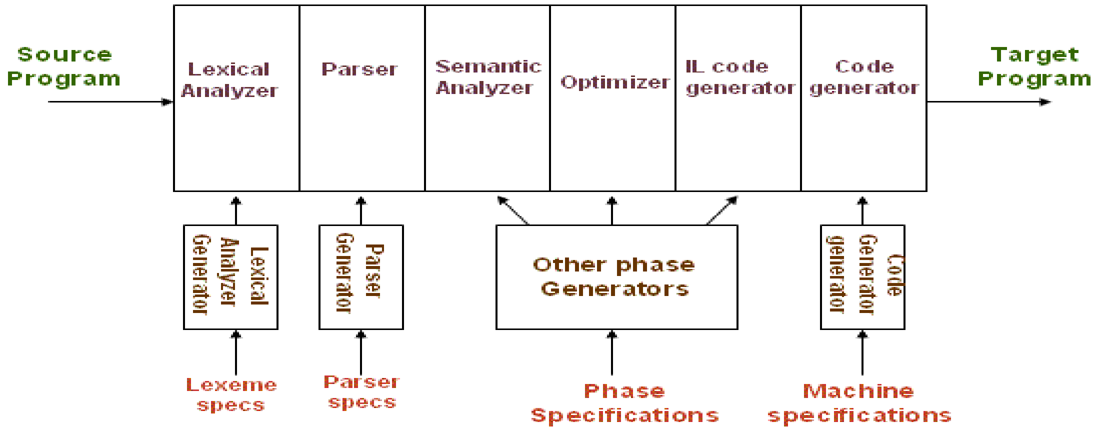
- Language is broken into sub components like lexemes, structure, semantics etc.
- Each component can be specified separately.

For example, an identifier may be specified as

- A string of characters that has at least one alphabet
- starts with an alphabet followed by alphanumeric
- letter (letter|digit)*

There are ways to break down the source code into different components like lexemes, structure, semantics etc. Each component can be specified separately. Similarly, there are rules for semantic as well as syntax analysis. We have some specifications to describe the target machine.

Tool based Compiler Development



Tools for each stage of compiler design have been designed that take in the specifications of the stage and output the compiler fragment of that stage. For example , Lex is a popular tool for lexical analysis,

YACC is a popular tool for syntactic analysis. Similarly, tools have been designed for each of these stages that take in specifications required for that phase e.g., the code generator tool takes in machine specifications and outputs the final compiler code.

This design of having separate tools for each stage of compiler development has many advantages that have been described on the next slide.

Compiler Construction Tools:

The compiler writer, like any software developer, can probably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on.

In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler. These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.

The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Some commonly used compiler-construction tools include

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.

2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.

4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

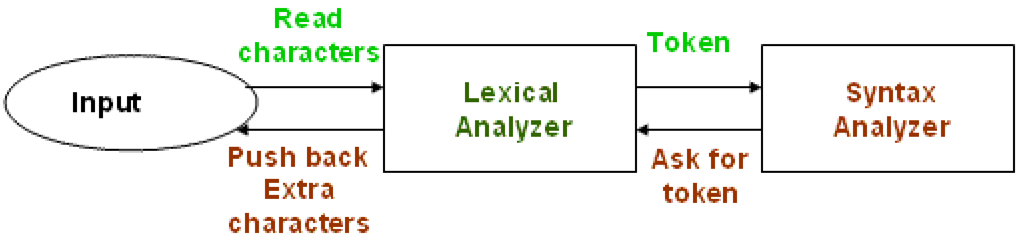
5. Data- flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data- flow analysis is a key part of code optimization.

6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Lexical Analysis

- The first phase of the compiler is lexical analysis.
- The lexical analyzer breaks a sentence into a sequence of words or tokens and ignores white spaces and comments.
- It generates a stream of tokens from the input.
- This is modelled through regular expressions and the structure is recognized through finite state automata.
- If the token is not valid i.e., does not fall into any of the identifiable groups, then the lexical analyser reports an error.
- Lexical analysis involves recognizing the tokens in the source program and reporting errors, if any.
- Token: A token is a syntactic category. Sentences consist of a string of tokens. For example constants, identifier, special symbols, keyword, operators etc are tokens.
- Lexeme: Sequence of characters in a token is a lexeme. For example, 100.01, counter, const, "How are you?" etc are lexemes.

Interface to other phases

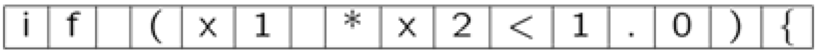


The lexical analyser reads characters from the input and passes the tokens to the syntax analyser whenever it is asked for a token. For many source languages, there are occasions when the lexical analyser needs to look ahead several characters beyond the current lexeme for a pattern before a match can be announced.

For example, > and >= cannot be distinguished merely on the basis of the first character >. Hence there is a need to maintain a buffer of the input for look ahead and push back.

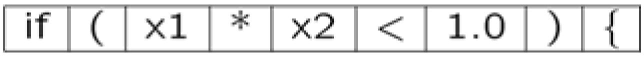
We keep the input in a buffer and move pointers over the input. Sometimes, we may also need to push back extra characters due to this lookahead character.

Recognize tokens and ignore white spaces, comments



i	f		(	x	1		*	x	2	<	1	.	0	)	{

Generates token stream



if	(	x1	*	x2	<	1.0	)	{

Input Buffering

- The lexical analyser scans the input string from left to right, one character at a time.
- It uses two pointers begin_ptr (bp) and forward_ptr (fp) to keep track of the portion of the input string scanned.
- Initially both the pointers point the first character of the input string.

bp
↓

i	n	t		a	,	b	;	a	=	a	+	5	;		b	=	b	*	3	;

↑
fp

- The forward ptr moves by searching the end of lexeme.
- When it finds the blank space, it indicates end of lexeme.
- In the above example fp finds a blank space then the lexeme int is identified.

bp
↓

i	n	t		a	,	b	;	a	=	a	+	5	;		b	=	b	*	3	;

↑
fp

- Then both the bp and fp are set at next token.
- The fp will be moved ahead at white space.
- When fp encounters white space, it ignores and moves ahead.

bp
↓

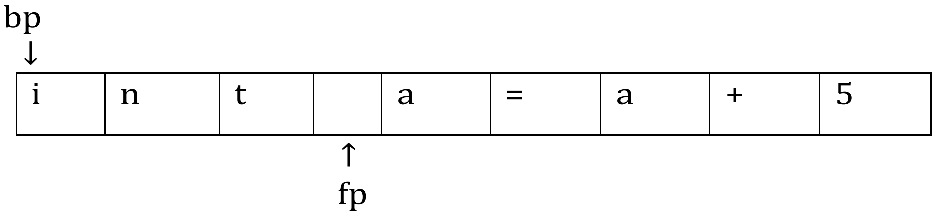
i	n	t		a	,	b	;	a	=	a	+	5	;		b	=	b	*	3	;

↑
fp

- The input character is read from secondary memory. But reading from secondary memory is costly.
- Hence, buffering technique is introduced.
- A block of data is first read into a buffer and then scanned by lexical analyser.
- There are two methods used: one buffer scheme and two buffer scheme.

One buffer scheme

- In one buffer scheme, only one buffer is used store the input string.
- If the lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to refilled, that makes the overwriting the first part of lexeme.

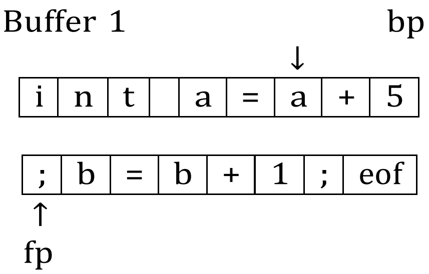


one buffer scheme

It is the problem with this scheme.

Two buffer scheme

- To overcome the above said problem, two buffers are used to store the input string. The first buffer and second buffer are scanned alternately.
- When the end of the current buffer is reached the other buffer is filled.
- The problem with this scheme is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.
- Initially, both the pointers bp and fp are pointing the first buffer. Then the fp moves towards right in search of the end of lexeme.
- When blank character is identified, the string between bp and fp is identified as corresponding token.
- To identify the boundary of the first buffer end of buffer character should be placed at the end of first buffer.
- In the same way, end of second buffer is also recognized by the end of first buffer mark present at the end of second buffer.
- When fp encounters first eof, then one can recognize end of first buffer and hence filling up of second buffer is started.
- In the same way when second eof is obtained then it indicates end of second buffer.
- Alternately, both the buffers can be filled up until end of the input program and stream of tokens identified.
- This eof character introduced at the end is called sentinel which is used to identify the end of buffer.



Code for input buffering

```
if(fp == eof(buff1))
{
 /*Refill buffer 2*/F
p++;
}
else if (fp == eof(buff2))
{
 /*Refill buffer1fp++;
```

```
      }
     Buffer 2
           else if(fp == eof(input))
          return;
  else
  fp++;
```

Specification of Tokens

- For programming languages, there are many types of tokens. They are constants, identifiers, symbols and so on. The token is normally represented by a pair of token type and token value.
- The token type tells the category of token and token value gives us the information regarding token. The token value is also called token attribute.
- Lexical analysis created the symbol table. The token value can be a pointer to symbol table in case of identifier and constant.
- The lexical analyser reads the input program and generates table for tokens.

```
main()
{
        int x =10;
        x = x * 5;
}
```

Lexeme	**Token**
main	Identifier
(	Operator
)	Operator
{	Special Symbol
int	Keyword
x	Identifier
=	Operator
10	Constant
;	Special symbol
*	Constant
}	Special symbol

The while spaces and new line characters are ignored. This stream of token will be given to syntax analyser.

Token Recognition by transition diagram

1. Recognition of Identifiers
2. Recognition of Delimiter (new lines, tabs, white spaces)
3. Recognition of operator (<, <=, >, >=, ==, !=)
4. Recognition of Keywords (if,else,for…)
5. Recognition of numbers (int / float)

A recognizer for language is a program that takes as input a string x and accepts if x is in+ the language and rejects if it is not in the language. We compile a regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.

Regular expressions are declarative specifications and finite automaton is the implementation. It can be deterministic or non-deterministic, both are capable of recognizing briefly the regular sets. Mathematical model of finite automata consists of:

Finite Automata has five tuples,

FA M= {Q, ∑, δ, $q_{0,}$ F} where

- **Q** is a set of states

- **∑** input alphabet,

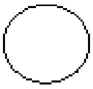
- δ transition function

-**q_0**is a start state

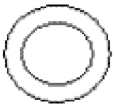
- **F** is set of final states F or accepting states, and

Pictorial notation

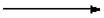
A state



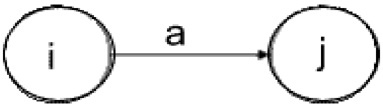
A final state



Transition



Transition from state i to state j on an input a

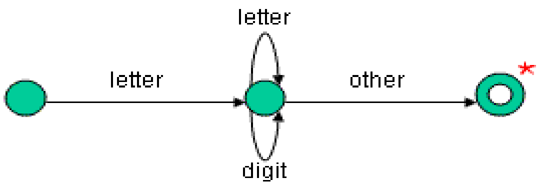


A state is represented by a circle, a final state by two concentric circles and a transition by an arrow.

Recognize tokens

Example. *id → letter (letter | digit)**

Transition diagram for identifier: In order to reach the final state, it must encounter a letter followed by one or more letters or digits and then some other symbol.


letter
letter
other
*
digit