08 February 2025

Language Processor:

Language processors are software systems that can translate and process programs written in high level languages into lower level languages that a computer can understand and execute.

Transpiler (src to src compiler) Converts code from one high level language to another.

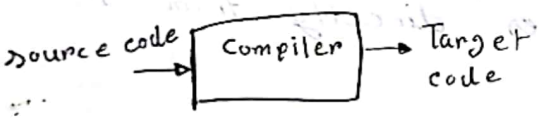
- Input ⟶ Output.
- High level language ⟶ (High level language) ⟶ Targeted Code.

There are different types of language processor out there.

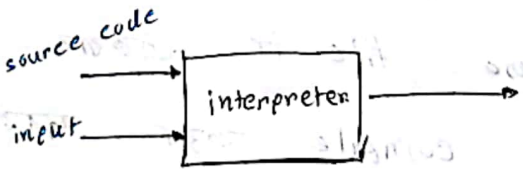
⇒ Assembler (Assembly language ⟶ machine language)

⇒ Compiler (Input Code ⟶ Target code)

↳ source code টাকে কোন language এ translate করতে চাচ্ছি, it can be assembly code, machine code, or any other low level language.



⇒ Interpreter.



⇒ Preprocessor

↳ Processes source code before compilation to handle directives like #include, #define, or macros.

- Compiler পুরোটা code একবারে compile করবে; পুরোটা compilation শেষে কোথায় কোথায় error আছে, সবগুলো (error) show করবে,

- Whereas, interpreter line by line execute করে, execution এর time এ একদমাইনে error থাকলে পাওয়া যাবে না।

Compiler এ input হিসেবে direct source code যায় না বরং modified source code যায়।

Hybrid Compiler ⇒ Compiler + Interpreter.

Difference between Compiler and Interpreter. ★★★

- Compiler translates entire source code at once into machine code; whereas, interpreters translates the code line by line.
- Compiler generates an executable file; Interpreter doesn't generate intermediate files; executes directly from the source.
- Compiler → faster execution time, after the code is compiled, because file টা একবার compiled হয়ে গেলে বারবার compile করার দরকার নাই; ঐ compiled file টা call করলেই হচ্ছে।

  Interpreter → Slower execution time due to repeated translation during execution.

- Debugging or Error handling is hard in compiler as it detects and reports all errors after the complete compilation process; whereas, interpreter stops at first encountered error, making the debugging easier.

- Compiler → Compiled code is platform dependent and Interpreter → Code is platform independent.

- Compiler can perform extensive optimization during compilation for performance; whereas, interpreter typically performs minimal or no optimization.

■ Advantages of compiler over interpreter and interpreter over compiler.

⇒ Compiler advantage:

- faster execution: compiled code is optimized and directly executed as machine code.
- Optimization: compilers perform extensive optimization during translation.
- The executable (compiled programs) don't require compiler at runtime, making distribution easier.
- Since, the translation is done once, the runtime system doesn't need to allocate extra resource for interpolation, leading to potentially lower overhead during execution.

Interpreter advantage.

- Ease of debugging: Since interpreter execute codes line by line and stop at first error encountered, it is easier to identify and fix errors during development.

- Platform independence: Interpreted language can be more platform independent because the same source code can be run on any system with an appropriate interpreter, without needing to generate a machine specific executable.

☑ Advantage of compiler producing Assembly language instead of Direct Machine Language.

⇒ • Human Readability:

Assembly language is much more understandable than raw machine code, making it easier to debug code.

• Optimization.

Developers can fine tune the assembly code for performance improvements or to ensure that certain optimizations are applied.

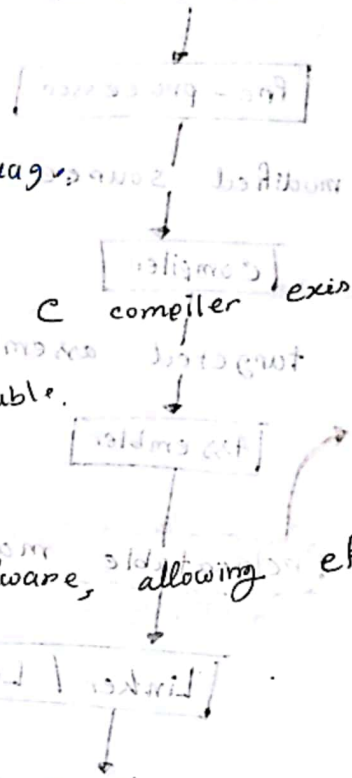
In case of mapping inputs to outputs compiler > interpreter

Better at error diagnostics compiler < interpreter

Advantage of using C as Target language.

⇒

- Widespread compiler support- C compiler exist করে; so
- Portability: almost সব platform এ generated code দ্বারা widely executable.
- Ease of debugging
- Performance: as C is close to hardware, allowing efficient code generation.



Language Processing System.

একটা language কে কিভাবে process করবে,

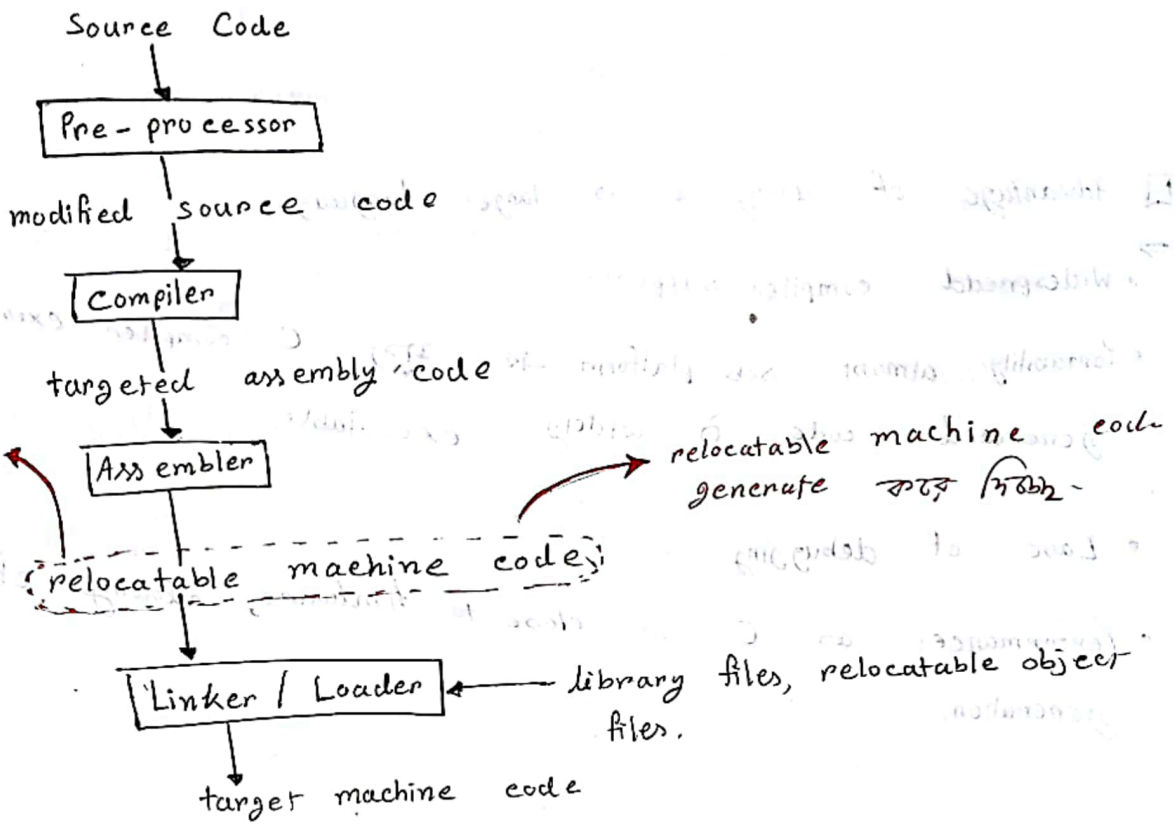


Figure : Compilation Process or a Language Processing system

Preprocessing এ কি কি কাজ হয়:

- removing hashtags.
- adding respective files. — যা যা extra file add করার দরকার preprocessor তা নিয়ে আসবে।
- macro expansion.
- Operator Conversion.

removing hashtags:

```
#include <scope_table.h>
#include <math.h>
```

এগুলা remove হয়ে যায়, tai er header file টা load করে রাখবে।

এইরকম hashtag গুলো remove করবে; because compiler এর কাছে এইরকম hashtag গুলো process করার মতো grammar থাকে না।

macro expansion.

code এর মধ্যবর্তী default value নিয়ে কাজ করে। যেমন:

আমরা code এ pi একটা default আকারে define করলাম; যার value 3.14, এখন, আমরা code compile করার সময় যদি লিখি pi+1; তাহলে compiler এ তখন pi এর value থাকবে না; as compiler এর কাছে default value থাকে না so এটা solve করার জন্য,

`pi = 3.14` → এটার value preprocessor এ store করে রাখতে হবে.

default value খুলে macro expansion.

Preprocessor এর কাছে যে default value খুলে আছে তা নিয়ে সাজে memory তে, so that compiler এর কোনোরকম problem না হয়;

so compiler এর কাছে 3.14 + 1 যাবে.

Operator Conversion.

i++ ⇒ i = i + 1

j-- ⇒ j = j - 1

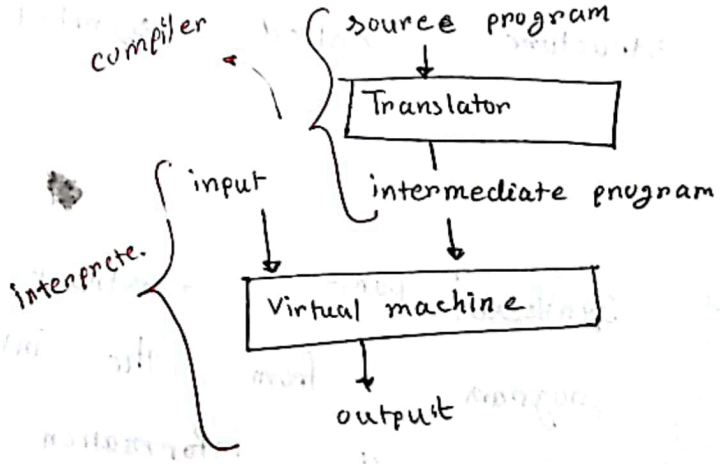


Figure: A hybrid compiler.

The structure of a compiler.

একটা compiler এর মধ্যে 6 টা phase থাকে এবং এই 6 টা phase কে একত্রে 2 টা stage এ ভাগ করা যায়,

(i) analysis ~~phase~~ part

(ii) synthesis ~~phase~~ part

The analysis phase breaks up the source code into constituent pieces and applies a grammatical structure on them. এই part থেকে একটা intermediate representation of source code create হয়, analysis stage এ যদি কোনো error ধরা পড়ে তাহলে user কে informative error message show করে for correction.

analysis part also collects information about the source program and stores it in a data structure called symbol table.

> # Symbol table এ hashtable use হয়,

Synthesis part construct the desired target program from the intermediate representation and the information in the symbol table.

symbol table stores the information about the entire source program & used by all phases in the compiler

Compiler এর সর্বমোট ৫টি phase নিম্নরূপ:

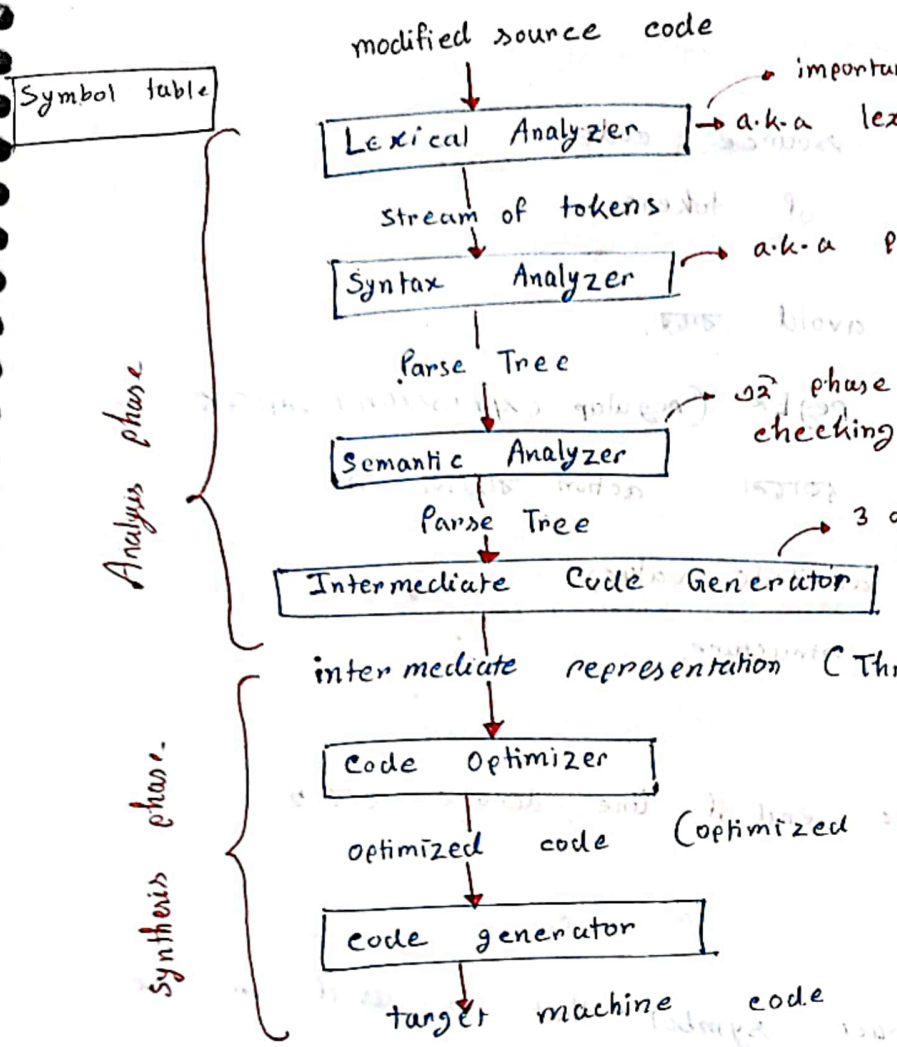


Figure: Phases of a compiler

Compiler এ line by line এ parallelly কাজ হয়।

Lexical Analyzer:

- The first phase of compiler is known as lexical analysis or scanning.
- It reads the stream of characters making up the source program and group the character into meaningful sequences called lexemes.

lexemes → meaningful sequence (stream) of characters

lexical analysis এর,
input হচ্ছে modified source code
output হচ্ছে stream of tokens.

lexer এর white space avoid করে,

white space এর জন্য regEx (regular expression) ব্যবহার করে white space " " এ কোনো action ব্যবহার হয়,

<token name, attribute value>

figure: Token structure.

int a ; → lexeme ; end of line denote করছে,
lexeme data type meaning করছে. lexeme variable name meaning করছে.

Token name is an abstract symbol that is used in the syntax analysis and attribute value points to an entry in the symbol table for this token.

কোন lexeme এর জন্য কোন token generate হবে তা নিজে থেকে define করতে হবে, কোন flow তে কোন matter করবে,

	token name
ID (identifier)	~~{Identifier}~~ {ID}
int	{INT}
float	{FLOAT}
string	{String}

• regEx এর ভিত্তিতে token generate হবে এবং flow matters

```
int   a   ;
INT   ID  SEMICOLON   → Stream of tokens.
```

প্রত্যেক line এ stream of tokens generate হয়,

line by line বা; parallely কাজ হয়.

```
int   a ;   INT (ID 0)   SEMICOLON
float d ;   FLOAT (ID 1)  SEMICOLON
```

• symbol table কে array হিসেবে consider করলে array এর কোন index এ আছে।

identifier এর info গুলো কোথায় store করে রাখা যেতে হবে, আমরা যেগুলো symbol table এ রাখি, যেখানে token info থাকবে

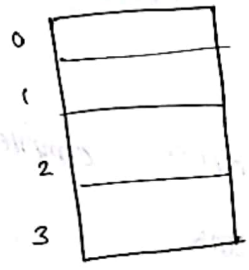
symbol table এ hashtable implement করা হয়, because hashtable এ search fast হয়। কিন্তু এগুলোকে রাখলে, waste of memory

actually symbol table এ index না রেখে আমরা object reference রাখি।

```
int   a ;
INT 2  ID  SEMICOLON
```

এখানে, INT ID SEMICOLON হওয়ার কথা কিন্তু তা না হয়ে INT 2 হয়েছে কারণ হল যখন check করা হয় তখন first এ integer পাইছে এবং এর first এর 3 টি character মিল থাকে তাই তার corresponding token name INT 2 আসছে, কিন্তু originially INT হওয়ার কথা; that's why flow matters.

```
integer → {INT2}
int     → {INT}
float   → {FLOAT}
ID      → {....}
```



symbol table Carray এর মতো consider করলে)

Semantic analysis এ আসার আগেই error checking, symbol table এ info store হয়ে যায়,

$z = a + b + c$

$t1 = a+b$
$t2 = t1+c$
$z = t2$

} Three address code

} Highest 3 টি address থাকতে পারে;

$t1 = a+b$
$z = t1+c$

code optimization.

Code optimization totally depends on grammar, যার grammar যত optimized output দিতে পারে, optimized output তার যত কম code দিয়ে optimize করে দিতে পারে.

Compiler compiles the code ahead of time before the program runs.

Interpreter translates the code line by line when the program is running.

compiler takes entire program and a lot of time to analyze source code, where the Interpreter takes single line of code and very little time to analyze it.

- Compiled code runs faster; while interpreted code runs slower.
- Compiler displays all errors after compilation. If the code has mistakes, it will not compile. But the interpreter displays errors of each line one by one.

```
int a ; int b ;
```
; → line end.

= → identifier (variable, array, function, class name)

identifier হলেই token id generate করব,

a, b → ID

int → INT

; → SEMICOLON

{identifier} → ID (always last এ রাখবো)

↓ or else সবার আগে identifier token generate হবে,

এই pattern গেলে, we go to regular expression; if regular expression এর সাথে match করে then we generate token ID

Shob gular pattern er sathe check korbo.

int → INT

; → SEMICOLON

{identifier} → ID

$[a-z]^+ [A-Z]^*$

not a pattern; its a notation (not defined)

Suppose, ano identifier ache and I am in Semantic phase nn. Tokho if I need to check values then how do I know kunta A-Bs kunta B er ID; cause shob e to ID.

∴ We make a Symbol table

Symbol table is a datastructure that uses hashtable

symbol table এ just identifier token store করব,

(int symbol table এ রাখব না)

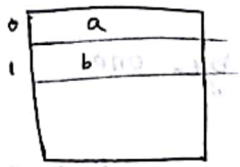
lexican analysis phase এ যদি কোনো pattern এর সাথে না মিলে, not even identifier; then it goes to panic mode (kono defined pattern এর সাথে match করছি,

{identifier} → {id, index}

↳ symbol table এর মধ্যে index ও token হিসেবে থাকে

int a; → INT <ID, 0> SEMICOLON

int b; → INT <ID, 1> SEMICOLON



symbol table.

· Token generation Done.

input for syntax analysis.

{ INT <ID, 0> SEMICOLON
  INT <ID, 1> SEMICOLON }

CFG = Context Free Grammar

Syntax analysis checks if the syntax is okay using Context Free Grammar. and ei grammar diye parse tree banay.

a = 10 , b = 20, c = 30.

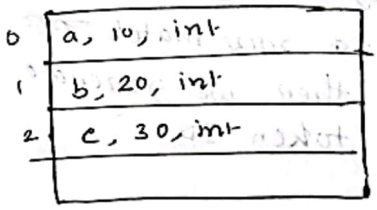
a = b + c

<ID, 0> = <ID, 1> , <ID, 2>

int kina int hole int + int korar por store korbe.

eikhane jei 2 ta value. add hobe, ei 2 tai int hisa, int hole addition possible.

int → INT
; → SEMICOLON
= → =
+ → +



Syntax Analysis এ syntax tree তে interior node representation operations and children node represents the arguments of operations.

Semantic analysis এ type info gather করে এবং syntax table or symbol table এ store করে.

• lexical analysis এ 2 phase:
① Scanning → comments & white space remove
② lexical analysis → finding lexemes and creating their corresponding tokens.

■ lexical analysis vs parsing

Compiler efficiency is improved. A separate lexical analyzer allows us to apply speciald techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly