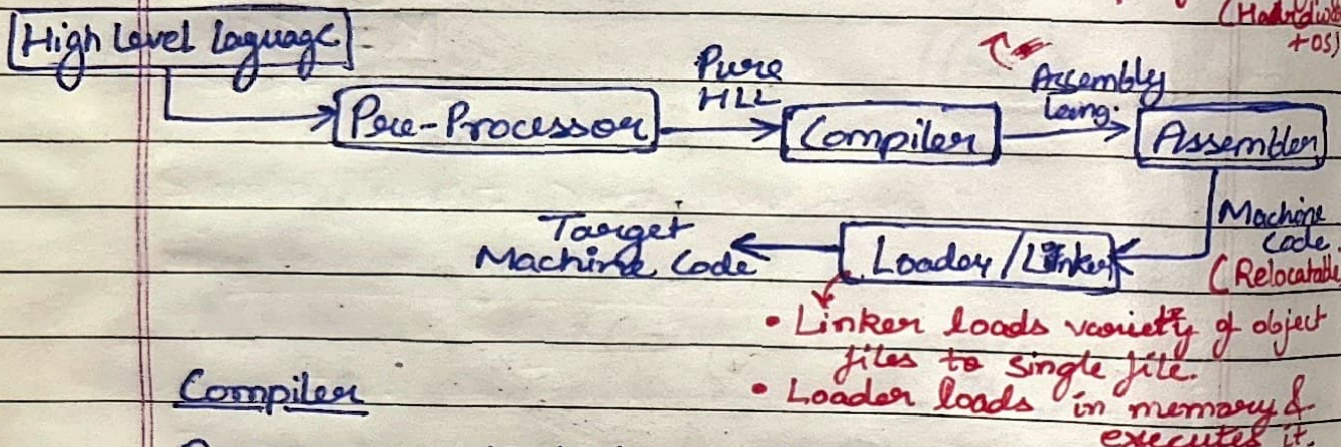
Compiler Design

Date: 1/1/-1

Introduction to Compilers

Language Processing System

Programs written in high-level languages are converted to machine usable in a series of steps.

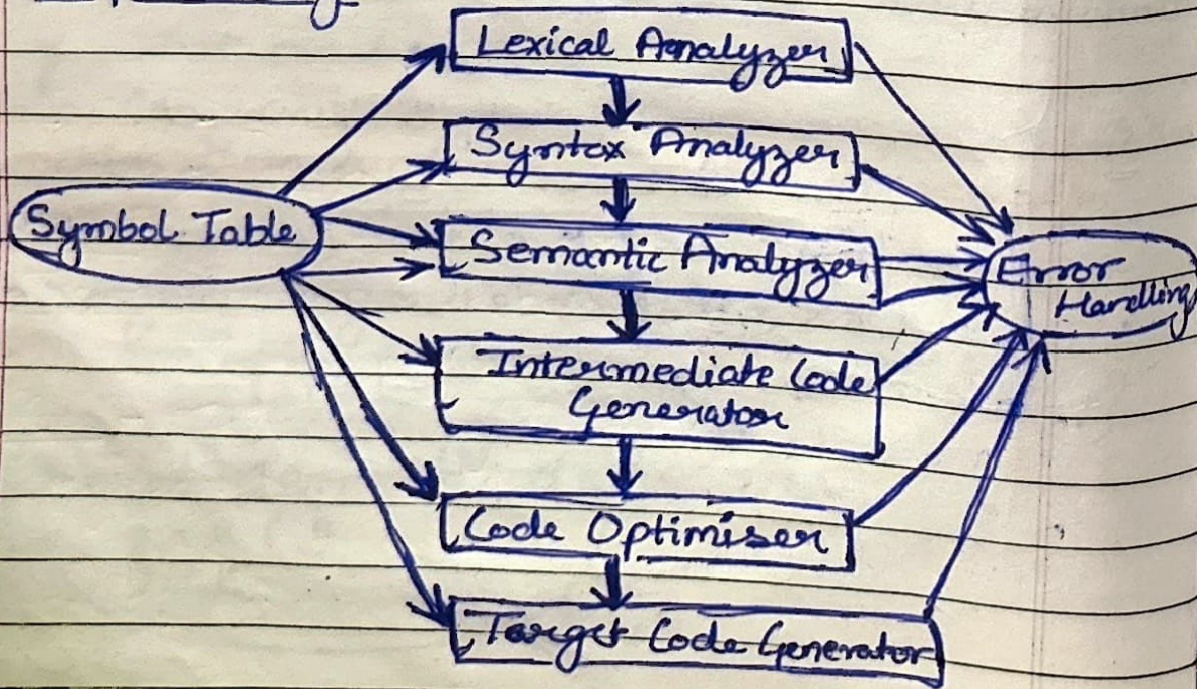


- Neither a binary file nor High Level
- Unique for every platform (Hardware + OS)
- Linker loads variety of object files to single file.
- Loader loads in memory & executes it.

Compiler

A program that translates code written in high level language to low level language (eg. assembly lang.) to create an executable program.

Compiler Design



Date / /

Code check

2 Phases → Analysis Phase
→ Synthesis Phase

Lexical Analyzer ÷ Cleans input text by removing comments, whitespaces etc. & breaks it input text to tokens.

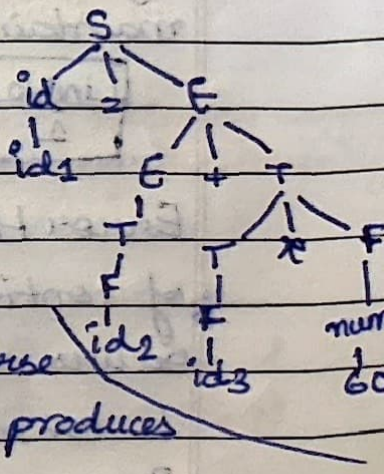
int a = 10; ⟶ Tokens: int, a, =, 10, ;

Syntax Analyzer ÷ Also called Parser, it creates syntax tree & uses production of context Free Grammar to construct parse tree.

S → id = E
E → E+T / T
T → T*F / F
F → id / num

Related Production Rules

$id_1 = id_2 + id_3 * 60$



Semantic Analyzer ÷ Verifies if parse tree is meaningful or not. Also produces verified / Annotated parse tree.

Intermediate Code Generator ÷
Generates intermediate code which is machine independent. To build new compiler steps upto it here are same & last 2 part can be coded machine specific.

$x = y + z * 60$
$t_1 = z * 60$
$t_2 = y + t_1$
$x = t_2$

Date / /

Code Optmizer :- Transform the code so that it consumes fewer resources & produces more speed.

Target Code Generator :- Machine understandable code.

```
MOV R1, Z
MUL R1, 60
ADD R1, Y
STORE X, R1
```

Symbol Table :- Data structure being used & maintained by compiler.

LineNo	Keyword	Identifier	Constant	Operator
1	int	x	10	;

Error Handler :- Sub-routine to take care of continuation of compilation even if error occurs. After phase 3 if error handler object is empty then source code is error free.

3 types of error → Lexical, Syntax, Semantic

int a=10 (int) a+b=c; (c=a+b) a='C'+1.2 (?)

Phases & Passes

Phase - Distinguishable stage, which takes I/P from previous stage, processes & gives O/P that can be used by next stage.

Pass - Traversal of compiler through entire program.

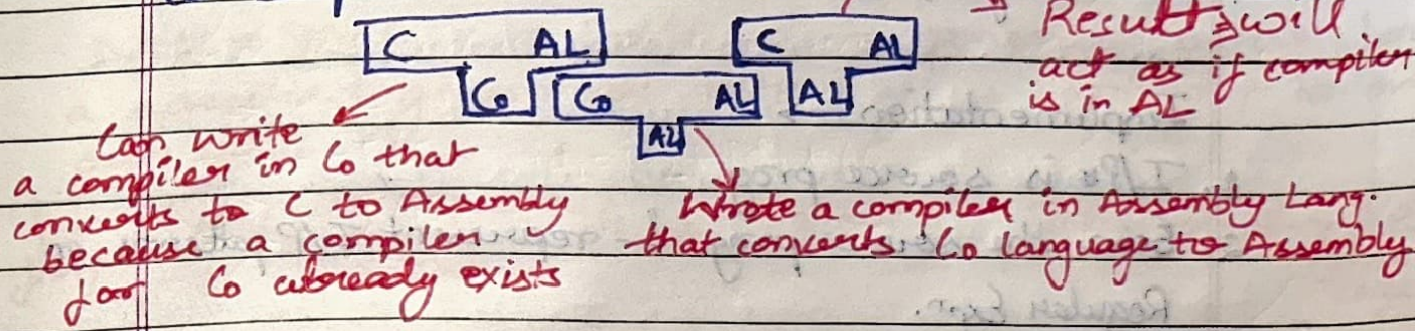
Date / /

→ Single Pass Compiler → Faster, memory efficient, less optimization
→ Multi Pass Compiler → Multiple scans, more optimized code, context awareness more memory usage. Ex → GCC

Ex → Tiny C Compiler, FORTRAN (early)

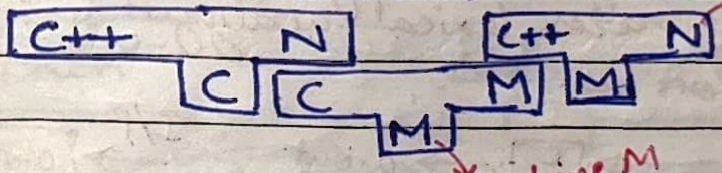
Bootstrapping

Earlier, compilers were written in assembly language but as languages evolved so did writing compilers. Writing a compiler in same language it intends to compile.



Cross Compiler

Allows devs to write code on 1 machine & execute it on another.



Lexical Analyzer

Source Prog → Lexical Analyzer → Token

→ Lexeme - Sequence of ~~Prog~~ characters in source code that matches a pattern for token. It is the actual text
→ Token → Logical meaning assigned to lexeme

Date / /

Lexing can be divided in 2 phase :-

i Scanning - Breaking input string into lexemes & categorizing into tokens

ii Evaluating - Converting lexemes into processed values.

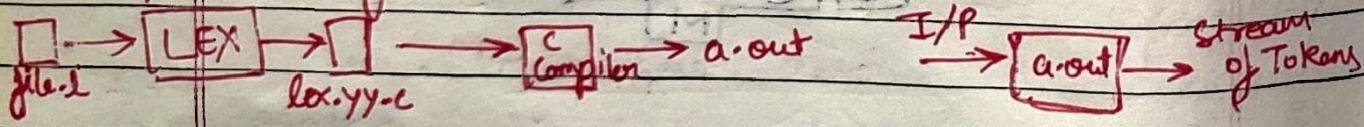
Ex → x = a + b;

[(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b)]

Implementation →

1. I/P is source prog.
2. Scan the source prog. & represent I/P patterns in Regular Expn.
3. RExpn → NFA → DFA → mDFA → lexemes

Note → Instead of us writing, a UNIX utility (LEX) exists that generates lexical analyzer using Regular Expression for us.



Syntax Analysis

Type 2 Grammar (CFG)

$\alpha \rightarrow \beta$ → Variables

$\alpha \in V_n \quad |\alpha| = 1$

$\beta \in \{\Sigma \cup V_n\}^*$ ↓ terminals

Formal Grammar represent specification of programming lang. with use of production rul

Date / /

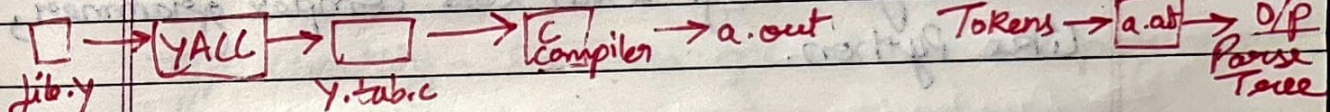
BNF Notation

Backus-Naur Form is notation used to express grammar of computer language.

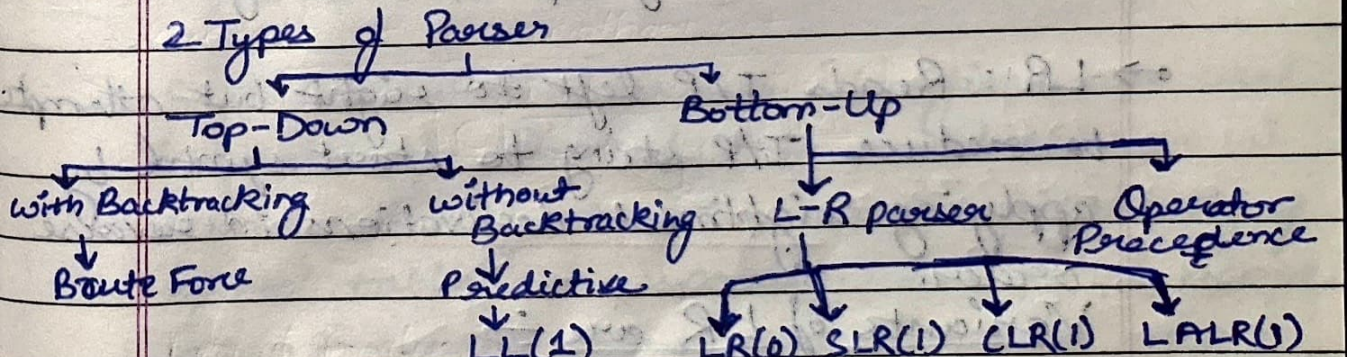
Symbols → Terminals, non-terminals Production Rules

Ex: Expression → Expression + Term | Term
~~Term → Term * Factor | Factor~~
Factor → (Expression) | Number
Number → Digit | Number Digit
Digit → 0|1|2|3|4|5|6|7|8|9

Note → YACC is a tool used to generate parser for a specified grammar instead of us writing it ~~on~~ our own.



Basic Parsing Techniques



Top-Down is method of parsing in which parser starts at root of parse tree & works its way down to leaves.

It cannot work with Left Recursive Grammar, Ambiguous grammar...

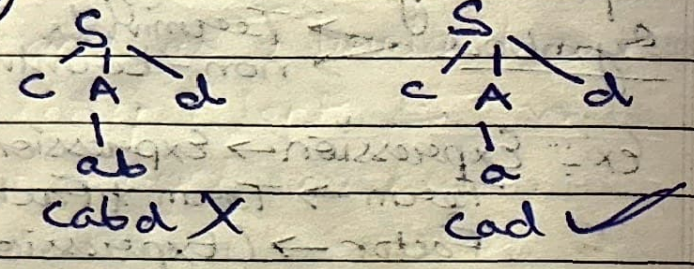
It can work on Non-Deterministic grammar

Date / /

using backtracking but that is inefficient.

→ Brute Force : Try all alternatives

Ex S → cAd
A → ab/a
ω_1 = cad



→ LL (1) : Reads the I/P from left to right & constructing a leftmost derivation of sentence. Uses a parsing table & stack to decide which production to apply.
Suitable for lang. with less complex grammar, like Python.

Bottom-up starts from leaves (I/P tokens) & work their way up to root of tree.

→ LR : Reads I/P left to right but attempts to reduce I/P string to start symbol by applying rightmost derivations in reverse order.
Variants of LR are :-
→ SLR : Simple LR
→ LALR : Lookahead LR
→ CLR : Canonical LR (most powerful type of LR)

LR parsers are used in modern prog. langs.
Ex - C, C++, Java

Date / /

~~Syntax directed translation~~

Semantic Analysis

Takes Abstract syntax tree (AST) generated by syntax analysis phase as its input & performs type checking, scope resolutions & validates semantic consistency according to language's rules.

AST is traversed & compiler builds & maintains a symbol table that tracks information about variables, functions, classes etc

Type Checking - Ensures operations are performed on compatible types.

```
int a=5;
float b=3.5;
a=a+b; // Error.
```

Scope Resolution - Ensures identifiers are used in correct context & that variables

→ Scope Hierarchy - Compiler manages hierarchy of symbol tables corresponding to nested scope. When traversal enters new scope a new table is created.

→ LookUp - When identifier is referenced it looks in current scope, if not found moves to next outer scope & so on.

```
int x = 10;
void foo() {
    int y=5;
    { int x=3; // This x shadows outer x
      y = x+2;
    }
}
```

Date / /

Intermediate Code Generation

Converts annotated AST into intermediate code, which is abstract representation of source code & acts as a bridge b\w HLL & LLL

⇒ It is machine independent & can be translated to different target machine.

Intermediate Code Forms -

→ Three-Address Code : Each instruction has at most 3 operands (2 source, 1 destination)

Ex → $t_1 = a + b$

$t_2 = t_1 * c$

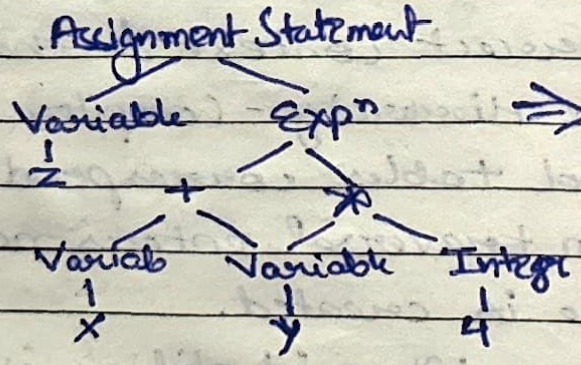
→ Quadruples : ~~Exact~~ $(+, a, b, t_1)$

→ Directed Acyclic Graph : Nodes represent operations & edges represent data flow.

Example →

```
int main(){
  int x=2, y=3, z;
  z = x + y * 4;
  return z;
}
```

⇒



⇒ Intermediate Code gen.

Traverse AST, process each node.

$t_1 := y * 4$

$t_2 := x + t_1$

$z := t_2$

Date / /

~~Target Code Generation~~

Code Optimization

Improving Intermediate code so that final machine code runs efficiently. ~~It~~ transforms intermediate code into equivalent but more optimized form.

Techniques →

- Constant Folding : $a = 2+3 \rightarrow a = 5$
- Constant Subexpression Elimination: $b = a*c$, $d = a*c \Rightarrow b = a*c$, $d = b$
- Dead Code Elimination
- Loop Invariant Code Motion: Code that does not change within loop is moved outside
- Strength Reduction: Replace expensive operations $a*2 \rightarrow a \ll 1$

Machine dependent ↓

- Register Allocation: Assign frequently used variable to registers for faster access

Code Generation

Takes optimized intermediate code & translate to machine code that can be directly executed by target processor.

Tasks in code generation -

- Instruction Selection: choose appropriate machine instruction to represent operations in intermediate code.

  $a = b + c$

  addi x1, x2, x3 (RISC-V processor)

- Register Allocation: Assigning ~~registers~~ variab

Date / /

to registers to minimize memory allocation

a = b * c a assigned to x1
b " " x2
c " " x3

Machine code

mul x1, x2, x3

- Instruction Scheduling: Rearranging instructions to improve pipelining & reduce stalls.

a = b * c
d = a + e

without scheduling
mul x1, x2, x3
add x4, x1, x5

with scheduling
mul x1

- Memory Mangement: Handling memory allocation & deallocation for variables & data structures.