

# COMPILER CONSTRUCTION

LAB Guide Book



## Contents

Introduction.....	2
LAB 1 .....	3
1.1 Variable Definitions .....	3
1.2 Default Target <code>all</code> .....	3
1.2.1 Overall Summary of Workflow:.....	5
1.3 Special Targets .....	5
1.4 Assembler Target.....	6
LAB 2 .....	7
2.1 Lexer File.....	7
2.1.1 What the Lexer Does.....	7
2.1.2 How it Works Internally.....	7
2.1.3 Code in Text Editor (VS CODE) .....	8
2.2 Parser File.....	9
2.2.1 What the Lexer Does.....	9
2.2.2 How it Works Internally.....	9
2.2.3 Code in Text Editor (VS CODE) .....	10
LAB 3 .....	11
LAB 4 .....	12
LAB EVALUATIONS .....	15
LAB 1 .....	15
Sample Input .....	15
Code: .....	15
LAB 2 .....	15
Sample Input .....	15
Code 1.....	15
Code 2.....	16
LAB PERFORMANCE.....	17
Sample Input .....	17
Code 1.....	17
Code 2.....	17
SHORTCUT KEYS .....	18
VIVA .....	19

# Introduction

This lab series is about understanding how a **compiler** works. You'll learn how source code is processed step by step—from reading it, breaking it into tokens (Lex), checking grammar (Bison), and building symbol tables for variables and types.

- **Lab 1** covers compilation steps using Makefile and GCC.
- **Lab 2 & 3** focus on using Flex and Bison to build a calculator that can evaluate or just parse expressions.
- **Lab 4** adds semantic checking and symbol table handling to detect errors in code.

You'll also practice **tokenizing**, **parsing**, and checking for errors in real C-style code. The viva questions help prepare you for theoretical understanding, like compiler phases, hybrid compilers, and syntax/semantic analysis.

This course helps you build a mini-compiler and understand how real compilers work!



# LAB 1

## 1.1 Variable Definitions

In a Makefile, **variables** (sometimes called “macros”) let you name and reuse values—compiler commands, flags, lists of files, etc. So you don’t have to repeat the same text over and over. You assign them once at the top:

```
CC      = gcc
CFLAGS  = -I.
DEPS    = hellomake.h
OBJS    = hellomake.o hellofunc.o
```

- **CC**  
The C compiler to use (`gcc`).
- **CFLAGS**  
Compiler flags—in this case `-I.` to add the current directory to the include path.
- **DEPS**  
Header dependencies (`hellomake.h`).

For Example: **DEPS** feeds into pattern rules like

```
%o: %.c $(DEPS)
    $(CC) -c $< $(CFLAGS)
```

so that any change to `hellomake.h` will auto-rebuild all those `.o` files.

- **OBJS**  
Object files for your library-like build (`hellomake.o hellofunc.o`).

For Example: **OBJS** collects all `.o` filenames into one name, letting you write:

```
hellomake3: $(OBJS)
    $(CC) -o $@ $^ $(CFLAGS)
```

rather than hard-coding each object on its own.

## 1.2 Default Target `all`

When you run `make` with no arguments, this builds and inspects `HelloWorld` in several ways:

## 1. Compile & link

```
gcc -o HelloWorld HelloWorld.c
```

- **Meaning:** Compiles the C source file `HelloWorld.c` into an executable named `HelloWorld`.
- **Short:** Compile and link → executable

## 2. Preprocess only

```
gcc -E HelloWorld.c > HelloWorld.i
```

- **Meaning:** Runs the **preprocessor** on `HelloWorld.c` and saves the output (with expanded macros and headers) to `HelloWorld.i`.
- **Short:** Preprocessing → `.i` file

## 3. Assemble from preprocessed

```
gcc -S -masm=intel HelloWorld.i
```

- **Meaning:** Converts the preprocessed `.i` file into **assembly code** using Intel syntax, outputting `HelloWorld.s`.
- **Short:** Compile to Intel-style assembly → `.s` file

```
as -o HelloWorld.o HelloWorld.s
```

- **Meaning:** Assembles the assembly file (`.s`) into an **object file** (`.o`).
- **Short:** Assemble `.s` → `.o`

## 4. Dump machine code (intel syntax)

```
objdump -M intel -d HelloWorld.o > HelloWorld.dump
```

- **Meaning:** Disassembles the object file into Intel-format **assembly instructions** and saves the output in `HelloWorld.dump`.
- **Short:** Disassemble `.o` → readable dump

## 5. Compile directly to object

```
gcc -c -o HelloWorld.o HelloWorld.c
```

- **Meaning:** Compiles `HelloWorld.c` into an object file only (`.o`), without linking.
- **Short:** Compile only → object file (`.o`)

```
objdump -M intel -d HelloWorld.o > HelloWorld2.dump
```

- **Meaning:** Disassembles the newly created object file again and writes output to `HelloWorld2.dump` (useful for comparison/debugging).
- **Short:** Disassemble `.o` again → second dump

### 1.2.1 Overall Summary of Workflow:

Step	Action	Output File	Purpose
1	Compile & Link	HelloWorld	Final executable
2	Preprocess	HelloWorld.i	Expanded code (macros, headers)
3	Compile to Assembly	HelloWorld.s	Human-readable assembly
4	Assemble	HelloWorld.o	Machine code (object)
5	Disassemble	HelloWorld.dump	Reverse-engineer assembly
6	Compile (only)	HelloWorld.o	Clean object file
7	Disassemble (again)	HelloWorld2.dump	For validation or inspection

### 1.3 Special Targets

- **.PHONY: clean**

Indicates that clean isn't a real file.

- **clean:**

Removes all object files.

```
.PHONY: clean

clean:
    rm -f *.o
```

Under the **clean:** rule, this line:

**rm -f \*.o** does the actual “cleanup” work:

- **rm** The Unix “remove” command—deletes files.
- **-f** Stands for “force.” It tells **rm** to:
  1. **Ignore nonexistent files** (no error if there aren't any **.o** files).
  2. **Suppress prompts or warnings**, even if files are write-protected.
- **\*.o** A shell wildcard (“glob”) matching **all** files in the current directory whose names end with **.o** (object files).

So when you run: **make clean**

Make invokes: **rm -f \*.o**

which force-deletes every **.o** file in that directory—without complaining if there aren't any. This keeps your workspace tidy by removing compiled artifacts.

## **1.4 Assembler Target**

```
assembler:  
    C:\masm32\bin\ml /c /coff /Cp prog1.asm  
    C:\masm32\bin\link -entry:main /subsystem:console prog1.obj  
    prog1
```

Uses MASM to assemble and link a Windows-style `prog1.asm` into `prog1.exe`.

# LAB 2

## 2.1 Lexer File

The lexer file is written using Lex/Flex, a tool for generating lexical analyzers (tokenizers). Its main job is to read an input stream and identify patterns (tokens) such as numbers and operators. After writing the code, we save the file with a **.l format**, such as cal.l.

### 2.1.1 What the Lexer Does

1. **Reads input** (like `1 + 12`) **character by character**.
2. **Matches patterns** using regular expressions:
  - `digit digit*` → any number like `1`, `12`, `345` → labeled as `NUMBER`
  - `"+"` → matches plus sign → labeled as `ADD`
  - Spaces and tabs (`[ \t]`) → **ignored**
3. **Prints token type** for each match using `printf`.

### 2.1.2 How it Works Internally

- The lexer uses `flex` to generate C code (`lex.yy.c`) based on the rules.
- The `main()` function runs `yylex()` to start scanning and processing input.

### Sample Input

```
1 + 12
1 + 13 *
```

### Lexer file (.l) work

#### Sections Breakdown:

- `%option noyywrap`: Prevents linking errors by disabling `yywrap()`.
- `digit [0-9]`: Defines a digit pattern.
- `delim [ \t]`: Defines space and tab as delimiters (ignored).

#### Rules (%% ... %%):

- `digit digit*`: Matches integers like `1`, `12`, etc., prints them as `NUMBER`.
- `"+"`: Matches plus sign and prints as `ADD`.
- `delim`: Ignores spaces/tabs.

#### Main Function:

- Calls `yylex()` to perform lexical analysis on input.



## Make File

A **Makefile** automates the steps to compile and run a program. It runs flex to generate code, compiles it using gcc, and executes the output saving time and avoiding manual commands.

For instance;

```
main:
    flex prog2.1
    gcc lex.yy.c -o p
    ./p < $(input) > $(output)
```

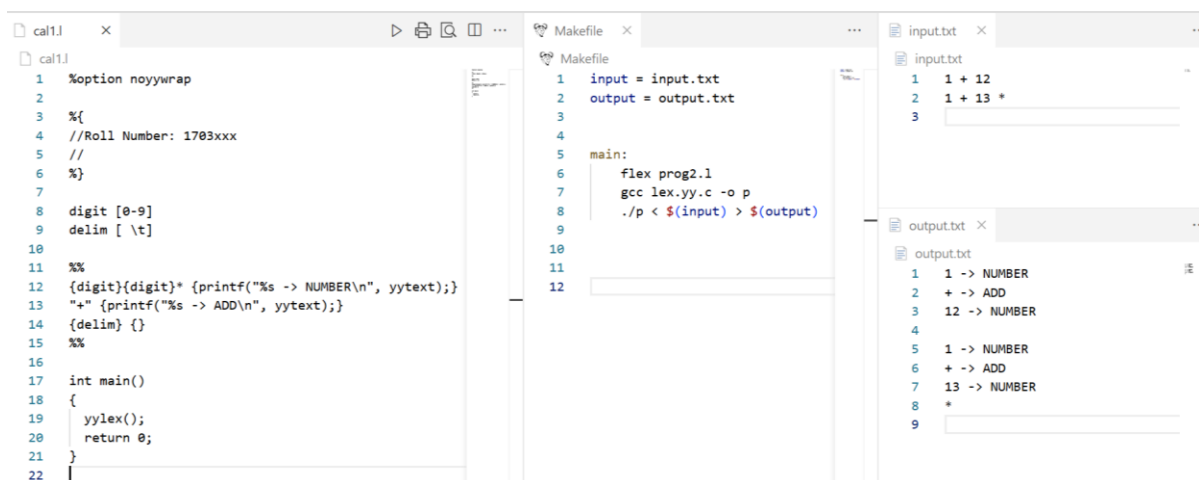
- `flex prog2.1`: Generates `lex.yy.c` from the lexer file.
- `gcc lex.yy.c -o p`: Compiles the C code to create executable `p`.
- `./p < $(input) > $(output)`: Runs the program with redirected input and output.

## Expected Output

```
1 -> NUMBER
+ -> ADD
12 -> NUMBER
```

```
1 -> NUMBER
+ -> ADD
13 -> NUMBER
*
```

### 2.1.3 Code in Text Editor (VS CODE)



## 2.2 Parser File

A **parser** is a program that reads input (usually from a lexer) and checks whether the input follows a particular grammar (rules of a language). It then builds a **parse tree** or evaluates expressions based on the structure.

### 2.2.1 What the Parser Does

A parser file defines grammar rules and evaluation logic to process input tokens from the lexer. It checks if the input is valid, computes the result, or shows an error if invalid. This file is saved with a **.y** extension and is used by Bison to generate the parser code.

### 2.2.2 How it Works Internally

#### 1) **bison -d cal.y:**

- a) Bison reads **cal.y**.
- b) It generates two files:
  - i) **cal.tab.c**: C code for the parser.
  - ii) **cal.tab.h**: Header file defining tokens like **NUM**, **ADD**.

#### 2) **flex cal.l:**

- a) Flex reads **cal.l**.
- b) Generates **lex.yy.c**: the **lexer** that identifies numbers, **+**, **-**, etc.

#### 3) **gcc cal.tab.c lex.yy.c -o p:**

- a) Compiles both lexer and parser into executable **p**.

#### 4) **./p < input > output:**

- a) Runs the parser, feeding it input expressions (like  $2 * 3 * 10 / 2 * 5$ ).
- b) The parser computes the result and prints it.

## Sample Input

```
2 * 3 * 10 / 2 * 5
3 * 2 * 10 / 5 * 2
3 * 20 * 20 / 3 * 2
23 + 3 - (3 * 2) / (2*5)
```

## Parser file (.y) work

### Sections Breakdown

- **%{ ... %}**: Includes standard headers and declares **yyerror()**, **yylex()**.
- **%token NUM ADD SUB MUL DIV**: Declares tokens received from lexer.
- **%left MUL DIV** and **%left ADD SUB**: Set operator precedence and associativity.
- **%start cal**: Defines start symbol for parsing.

### Grammar Rules (between %% ... %%):

- `cal: cal exp '\n' { printf("Result = %d\n", $2); } | ;`  
- Parses one or more expressions and prints results line by line.

```
exp: exp ADD exp { $$ = $1 + $3; }      | exp SUB exp { $$ = $1 - $3; }  
| exp MUL exp { $$ = $1 * $3; }        | exp DIV exp { $$ = $1 / $3; }  
| '(' exp ')' { $$ = $2; }              | '{' exp '}' { $$ = $2; }  
| '[' exp ']' { $$ = $2; }              | NUM { $$ = $1; };
```

- Defines arithmetic expressions and evaluates them.

### Main Function:

- Calls `yyparse()` to start parsing and evaluating input.

### Error Handling:

- `yyerror(char *s):` Prints error messages on parsing errors.

### Expected Output

Result = 150

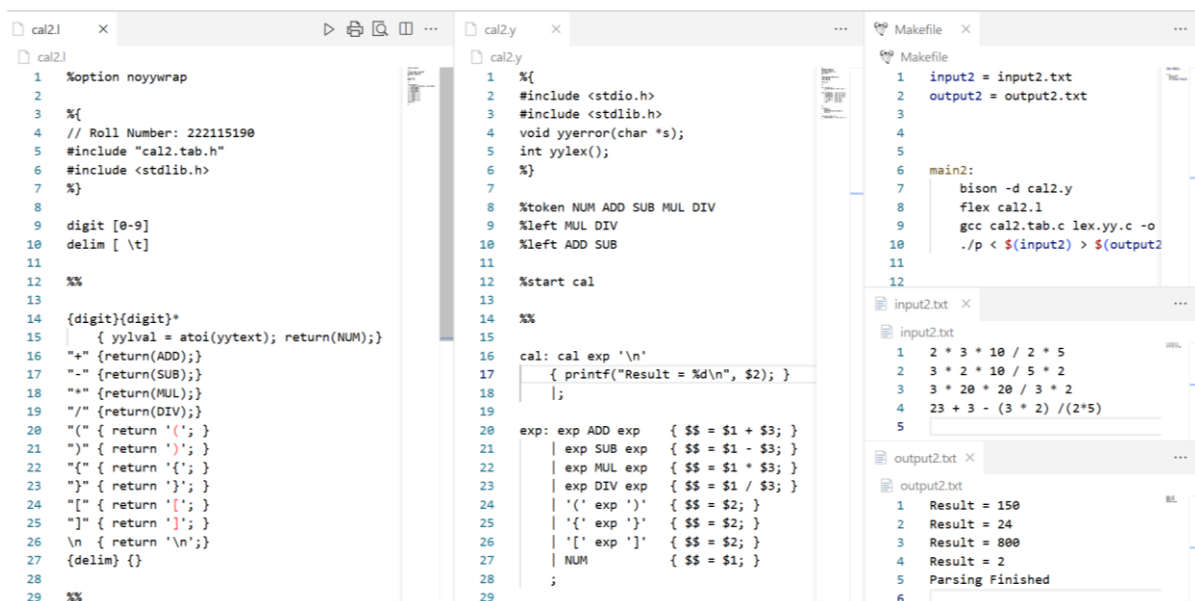
Result = 24

Result = 800

Result = 2

Parsing Finished

### 2.2.3 Code in Text Editor (VS CODE)



```
cal2.l
1 %option noyywrap
2
3 %{
4 // Roll Number: 222115190
5 #include "cal2.tab.h"
6 #include <stdlib.h>
7 %}
8
9 digit [0-9]
10 delim [ \t]
11
12 %%
13
14 {digit}{digit}*
15 | { yyval = atoi(yytext); return(NUM); }
16 "+" { return(ADD); }
17 "-" { return(SUB); }
18 "*" { return(MUL); }
19 "/" { return(DIV); }
20 "(" { return('('); }
21 ")" { return(')'); }
22 "{" { return('{'); }
23 "}" { return('}'); }
24 "[" { return('['); }
25 "]" { return(']'); }
26 \n { return('\n'); }
27 {delim} {}
28
29 %%

cal2.y
1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 void yyerror(char *s);
5 int yylex();
6 %}
7
8 %token NUM ADD SUB MUL DIV
9 %left MUL DIV
10 %left ADD SUB
11
12 %start cal
13
14 %%
15
16 cal: cal exp '\n'
17 | { printf("Result = %d\n", $2); }
18 | ;
19
20 exp: exp ADD exp { $$ = $1 + $3; }
21 | exp SUB exp { $$ = $1 - $3; }
22 | exp MUL exp { $$ = $1 * $3; }
23 | exp DIV exp { $$ = $1 / $3; }
24 | '(' exp ')' { $$ = $2; }
25 | '{' exp '}' { $$ = $2; }
26 | '[' exp ']' { $$ = $2; }
27 | NUM { $$ = $1; }
28 ;

Makefile
1 input2 = input2.txt
2 output2 = output2.txt
3
4
5
6 main2:
7 bison -d cal2.y
8 flex cal2.l
9 gcc cal2.tab.c lex.yy.c -o
10 ./p < $(input2) > $(output2)
11
12

input2.txt
1 2 * 3 * 10 / 2 * 5
2 3 * 2 * 10 / 5 * 2
3 3 * 20 * 20 / 3 * 2
4 23 + 3 - (3 * 2) / (2 * 5)
5

output2.txt
1 Result = 150
2 Result = 24
3 Result = 800
4 Result = 2
5 Parsing Finished
6
```

## LAB 3

**Lab 2** and **Lab 3** are almost the same in structure, using Flex and Bison to parse arithmetic expressions.

The main difference lies in the **output behavior**:

- **Lab 2:**  
Parses and **evaluates the expression**, showing the **calculated result** using `printf("Result = %d\n", $2);`.  
Example output: `Result = 150`.
- **Lab 3:**  
Only performs **parsing without displaying any result**.  
Shows: `Parsing Finished` if there are no syntax errors.  
The `printf("Result = %d\n", $2);` line is removed or commented out.

This helps focus on the correctness of the grammar in **Lab 3**, while **Lab 2** emphasizes evaluation logic. We will see the **Lab Evaluation Chapter** ahead.

## LAB 4

You have:

- **Input File** (contains sample C-like code)
- **Flex (lexer.l)**: Tokenizes input
- **Bison (parser.y)**: Parses tokens, checks grammar, types, and builds symbol table
- **syntab.c/h**: Implements symbol table and type-checking logic
- **Output**: Prints symbol table insertions and semantic/type errors

### Input Code Breakdown

```
int a;  
char a;  
  
double c = 5;  
  
if(b > 3)  
{  
    d = a + c;  
}
```

### How Each File Works Together

#### Lexer and Parser file

As mentioned earlier, the Lexer and Parser files are in Lab 2 and Lab 3.

#### syntab.c + syntab.h (Symbol Table)

**insert(name, type)**

- Adds a new identifier to the symbol table if not declared before.
- Prints message of successful or duplicate insertion.

**search(name)**

- Searches symbol table for a variable.

**idcheck(name)**

- Returns 1 if declared, else prints error and returns 0.

### **gettype(name)**

- Returns stored type of the variable.

### **typecheck(type1, type2)**

- Compares two types.
- If they match: returns common type.
- If not: prints error.

### **Output Explanation Line-by-Line**

```
int a;
```

- Line 1
- First time declaring a
- `insert("a", INT_TYPE);` is called

#### **Output:**

In line no 1, Inserting a with type INT\_TYPE in symbol table.

```
char a;
```

- Line 2
- Second time declaring a → already in symbol table
- Duplicate declaration

#### **Output:**

In line no 2, Same variable a is declared more than once.

```
double c = 5;
```

- Line 4
- Valid declaration and initialization
- `insert("c", REAL_TYPE);`

#### **Output:**

In line no 4, Inserting c with type REAL\_TYPE in symbol table.

```
if(b > 3)
```

- Line 6
- b is **used** but not **declared** before

```
idcheck("b") fails
gettype("b") returns UNDEF_TYPE
```

Then checks `UNDEF_TYPE > INT_TYPE` using `typecheck`

### Output:

```
In line no 6, ID b is not declared.
In line no 6, Data type UNDEF_TYPE is not matched with Data
type INT_TYPE.
```

```
d = a + c;
```

- Line 8
- d is not declared → `idcheck("d")` fails
- a is `INT_TYPE`, c is `REAL_TYPE` → mismatch
- `typecheck(INT_TYPE, REAL_TYPE)` → mismatch

### Output:

```
In line no 8, ID d is not declared.
In line no 8, Data type UNDEF_TYPE is not matched with Data
type INT_TYPE.
In line no 8, Data type UNDEF_TYPE is not matched with Data
type REAL_TYPE.
```

### Final Output

```
Parsing finished!
```

Means parsing ran till the end despite errors.

### How the Whole System is Connected

Component	Purpose	Key Functions
input file <code>lexer.l</code>	Contains code	Given C-like input
	Scans and identifies tokens	Returns tokens like <code>ID</code> , <code>INT</code> , <code>ASSIGN</code>
<code>parser.y</code>	Builds parse tree, checks semantics	Calls <code>insert</code> , <code>idcheck</code> , <code>gettype</code> , <code>typecheck</code>
<code>symtab.c/h</code>	Maintains symbol table and type rules	Adds, searches, verifies variables
Output	Shows symbol insertions and errors	From <code>printf</code> in symbol table functions

# LAB EVALUATIONS

## LAB 1

Tokenize the code snippet;

### Sample Input

#### Code:

```
int main()
{

float b = 2.5;
Int a = 10;
char c = 'a';
Int aq;

double exp = -10.9e-27;

for( int i = 0; i<= 10; i++){
    printf("The Number is : %d",i);
    scanf("%d",&aq);
    A = A % 5;
}

return 0;
}
```

## LAB 2

Parse the following code snippets;

### Sample Input

#### Code 1

```
int main() {
int score;

printf("Enter the student's percentage score (0-100): ");
scanf("%d", &score);

switch (score / 10) {
case 10:
case 9:
printf("Grade: A\n");
break;
case 8:
printf("Grade: B\n");
break;
```



```
case 7:
printf("Grade: C\n");
break;
case 6:
printf("Grade: D\n");
break;
case 5:
printf("Grade: E\n");
break;
default:
printf("Grade: F\n");
break;
}

return 0;
}
```

## **Code 2**

```
int main() {
int year;

printf("Enter a year: ");
scanf("%d", &year);

if (year % 400 == 0) {
printf("%d is a leap year.\n", year);
} else if (year % 100 == 0) {
printf("%d is not a leap year.\n", year);
} else if (year % 4 == 0) {
printf("%d is a leap year.\n", year);
} else {
printf("%d is not a leap year.\n", year);
}

return 0;
}
```

## LAB PERFORMANCE

Tokenize and Parse the following codes and show the result of parsing;

### Sample Input

#### Code 1

```
int main() {
    int age;

    printf("Enter your age: ");
    scanf("%d", &age);

    if (age < 13) {
        printf("Category: Child\n");
    } else if (age >= 13 && age <= 19) {
        printf("Category: Teenager\n");
    } else if (age > 19 && age <= 64) {
        printf("Category: Adult\n");
    } else {
        printf("Category: Senior\n");
    }

    return 0;
}
```

#### Code 2

```
int main() {
    int num;
    char choice;

    do {
        printf("Enter an integer to see its multiplication table: ");
        scanf("%d", &num);

        printf("Multiplication Table for %d:\n", num);

        for (int i = 1; i <= 10; i++) {
            printf("%d x %d = %d\n", num, i, num * i);
        }


        printf("\nDo you want to calculate another multiplication table? (y/n): ");
        scanf(" %c", &choice);

    } while (choice == 'y' || choice == 'Y');

    printf("Exiting program. Goodbye!\n");

    return 0;
}
```

## SHORTCUT KEYS

- **Open Terminal:**  
`Ctrl + `` (backtick key, under Esc)
- **Open Folder:**  
Ctrl + K, then Ctrl + O (*press one after the other*)
- **Split & Navigate Editors**
  - **Split Editor (Side by Side):**  
Ctrl + \ (backslash key, above Enter)
  - **Switch Between Split Editors:**  
Ctrl + 1, Ctrl + 2, Ctrl + 3 ... (*based on editor position*)
-  **Reload Window**  
**Reload Window:**  
Ctrl + Shift + P → Type Reload Window → Press Enter

# VIVA

## 1. Structure of a Compiler

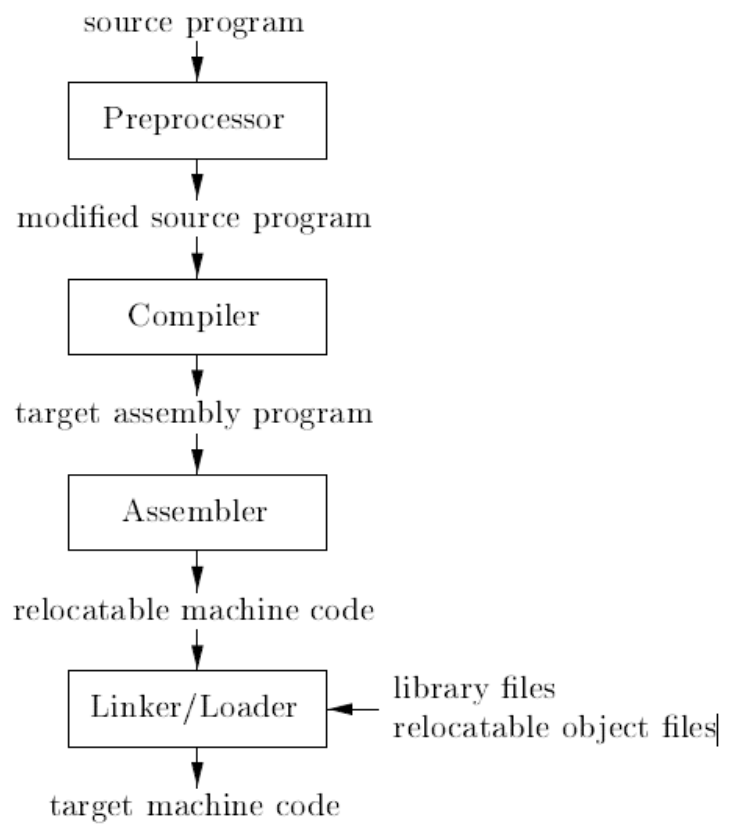
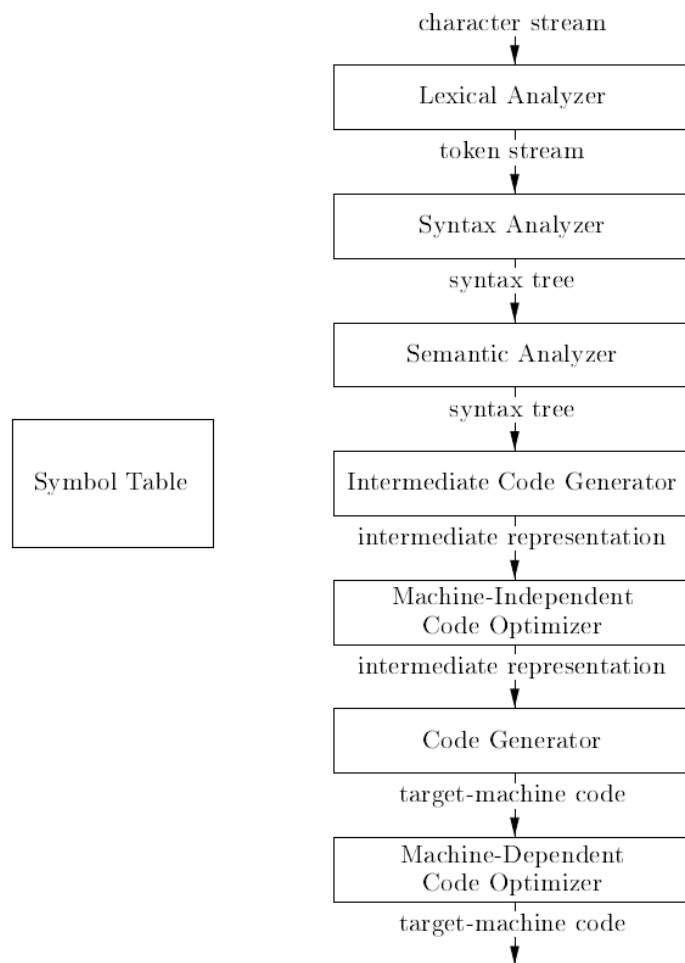


Figure 1.5: A language-processing system

## 2. Phases of a Compiler



### 3. Hybrid Compiler

### a. Ahead-of-Time (AOT) Compilation

- **Definition:** Compiles source code into machine code **before** the program is run.
- **Example:** C/C++, Android apps using AOT (e.g., ART on Android).
- **When it happens:** At **compile time**.
- **Advantages:**
  - Faster startup time (no compilation during runtime).
  - Lower memory usage during execution.
  - Can perform aggressive optimizations.
- **Disadvantages:**
  - Larger executable size.
  - Less flexible – can't optimize based on actual runtime behavior.

### **b. Just-in-Time (JIT) Compilation**

- **Definition:** Compiles code **at runtime**, converting intermediate code (like bytecode) into machine code.
- **Example:** Java Virtual Machine (JVM), .NET CLR, Python (PyPy).
- **When it happens:** During **program execution**.
- **Advantages:**

- Can optimize based on **real-time performance data**.
- Smaller initial file size.
- Allows platform-independent distribution (e.g., Java bytecode).
- **Disadvantages:**
  - Slower startup time (compilation happens at runtime).
  - Higher memory usage.

### Quick Comparison Table

Feature	AOT	JIT
<b>Compilation Time</b>	Before execution	During execution
<b>Startup Speed</b>	Fast	Slower
<b>Runtime Optimization</b>	No	Yes (dynamic)
<b>Code Portability</b>	Less portable	Highly portable
<b>Execution Speed</b>	Faster after startup	May get faster over time

### Real-world Usage

- **AOT:** C, C++, Swift, Rust, Android ART
- **JIT:** Java, Kotlin (JVM), C#, Python (via JIT engines)

#### 4. Tombstone Diagram

- Program
- Compiler
- Interpreter
- Compilation
- Cross-Compilation
- Two stage Compilation
- Bootstrapping
  - Half bootstrapping*
  - Full bootstrapping*

#### 5. Three Address Code

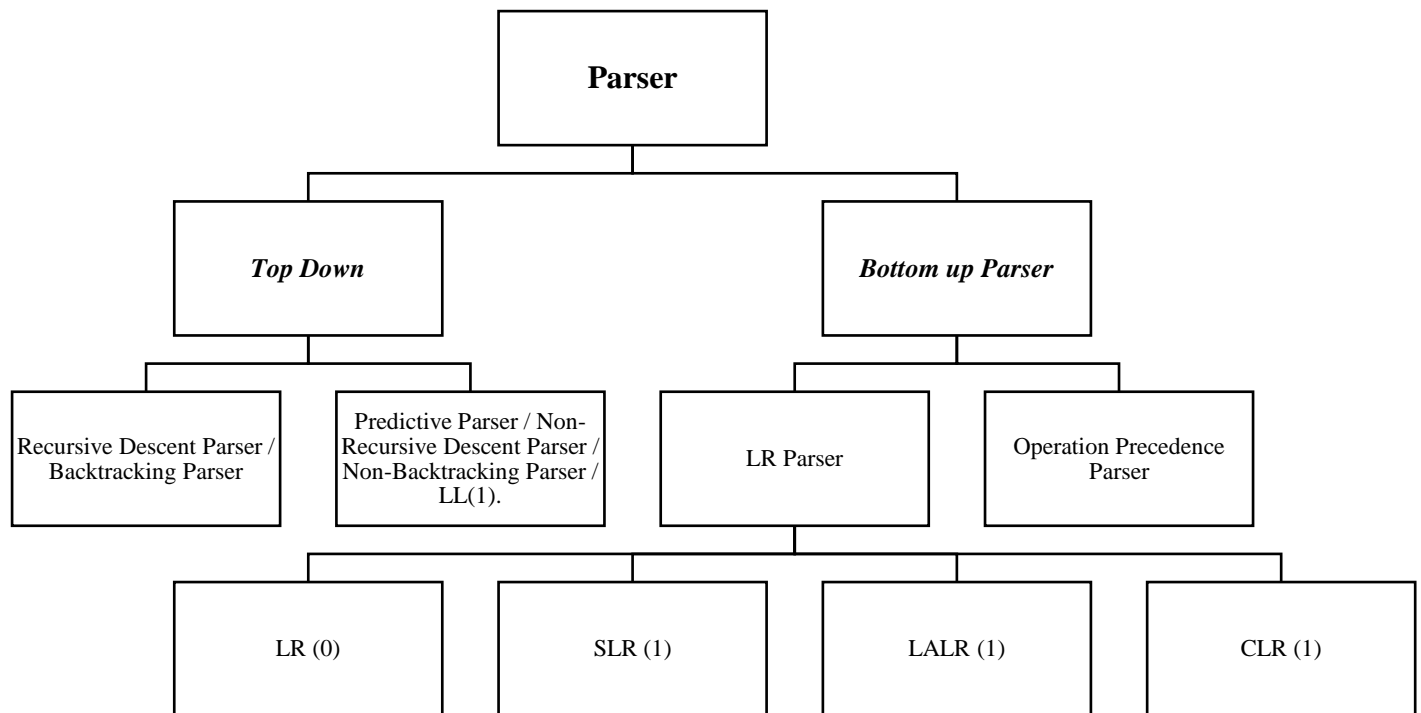
- Quadruple
- Triples
- Indirect Triples

#### 6. Lexical Analysis

- Tokenization

#### 7. Syntax Analysis

- First & Flow
- Types of Parser



## 8. LR Parser

### 1. LL(1) Parser

**Type:** Top-Down Parser

**Direction:** Left-to-right, Leftmost derivation (hence, LL)

**Lookahead:** 1 token

#### Characteristics:

- Uses **recursive descent** or **predictive parsing**
- Relies on **FIRST** and **FOLLOW** sets
- Cannot handle **left-recursive grammars**
- Easy to implement but less powerful than LR parsers

#### **Pros:**

- Simple and fast
- Good for hand-written parsers

#### **Cons:**

- Not suitable for complex or ambiguous grammars

## 2. LR(0) Parser

**Type:** Bottom-Up Parser

**Direction:** Left-to-right, Rightmost derivation in reverse

**Lookahead:** 0 tokens

### Characteristics:

- Uses **items with dot** (•) to represent parser state
- No lookahead symbol, so very limited in power
- Constructed using **LR(0) automaton**
- Can handle left-recursive grammars

### **Pros:**

- Stronger than LL(1)
- Good for detecting syntax errors early

### **Cons:**

- Cannot resolve **reduce/reduce** or **shift/reduce** conflicts due to lack of lookahead

## 3. SLR(1) Parser (Simple LR)

**Type:** Bottom-Up Parser

**Direction:** Left-to-right, Rightmost derivation in reverse

**Lookahead:** 1 token

### Characteristics:

- Extension of LR(0) with **lookahead** based on **FOLLOW sets**
- Reduces conflicts compared to LR(0)
- Uses **SLR parsing table** (ACTION and GOTO)

### **Pros:**

- More powerful than LL(1) and LR(0)
- Can handle a larger class of grammars

### **Cons:**

- Still limited; more complex grammars may require **LALR(1)** or **LR(1)**



Summary Table:			
Feature	LL(1)	LR(0)	SLR(1)
Parsing Type	Top-Down	Bottom-Up	Bottom-Up
Derivation	Leftmost	Rightmost (rev)	Rightmost (rev)
Lookahead	1 token	0 tokens	1 token
Handles Left Recursion	No	Yes	Yes
Grammar Power	Least	Medium	Higher
Conflict Handling	Limited	None	Reduced
Uses FIRST/FOLLOW	Yes	No	FOLLOW only

## 9. Semantic Analysis

- SDD (Syntax Directed Definition)
- SDT (Syntax Directed Translation)

## 10. Intermediate Code Generator

### a. TAC (Three Address Code)

An intermediate code used in compilers where each instruction contains at most three addresses (operands). Example:

$t1 = a + b$

$t2 = t1 * c$

### b. AST (Abstract Syntax Tree)

A tree representation of the abstract syntactic structure of source code. Each node represents a construct occurring in the source code, but it omits unnecessary grammar rules (like parentheses or semicolons).

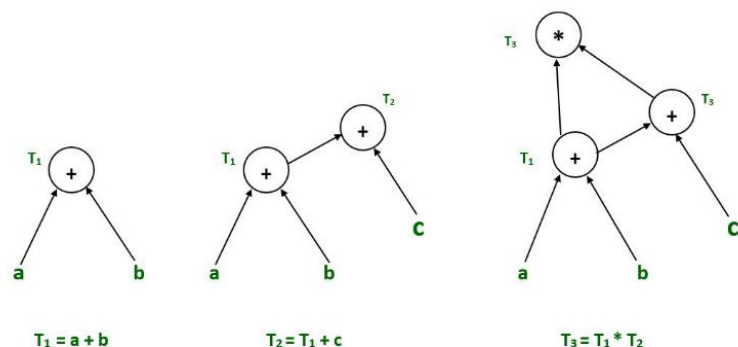
### c. DAG (Directed Acyclic Graph)

A graph used to represent expressions without repeating common subexpressions. It is used for **optimization** in compilers. It has **no cycles** and shows how values are computed. Example:

$T1 = a + b$

$T2 = T1 + c$

$T3 = T1 * T2$



**d. Postfix Notation**

Postfix notation is a way of writing expressions **without parentheses** and where **operators come after their operands**.

Example:

Infix	Postfix
$a + b$	$a b +$
$a + b * c$	$a b c * +$
$(a + b) * c$	$a b + c *$
$a + b + c$	$a b + c +$
$a * (b + c)$	$a b c + *$

**11. Code Optimization (See in [CC Class Notes](#))**

**12. Removal of Left Recursion from a Graph**

**Left recursion** in a grammar occurs when a non-terminal symbol appears as the **leftmost symbol** on the **right-hand side** of one of its own productions.

- |   |                 |
|---|-----------------|
| <b>13. <a href="#">Variable Definitions</a></b> | (LAB 1, Pg. 3)  |
| <b>14. <a href="#">Lexer file Working</a></b>   | (LAB 2, Pg. 7)  |
| <b>15. <a href="#">Parser File Working</a></b>  | (LAB 2, Pg. 9)  |
| <b>16. <a href="#">Symbol Tabel</a></b>         | (LAB 4, Pg. 12) |

**Notes**

1. [ClassNotes](#)
2. [Tombstone Diagram](#)
3. [Removal of Left Recursion from a Graph](#)

3. Class Record

- |                          |   |                          |
|--------------------------|---|--------------------------|
| 1) <a href="#">LAB 1</a> | 2) <a href="#">LAB 2</a>                | 3) <a href="#">LAB 3</a> |
| 4) <a href="#">LAB 4</a> | 5) <a href="#">REVIEW CLASS SESSION</a> |                          |