
	<b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION		SEMESTER: VI	PREPARED BY:PROF. SHALU PESHWANI

1. Overview of Computing Technology.....	2
1.1. Programming Language Grammars.....	2
1.2. Classification of Grammar.....	2
1.3. Implications in Programming Languages.....	4
1.4. Ambiguity in Grammatic Specification.....	4
1.5. Scanning.....	4
1.6. Parsing:.....	4
1.7. Language Processor Development Tools:.....	6
1.8. Overview of Lex & Yacc:.....	6
2. Introduction of Computation Tools.....	8
2.1. Assemblers.....	8
2.2. Compilers.....	14
2.3. Interpreters:.....	30
2.4. Debuggers:.....	31
2.5. Macro and Macro Processors.....	32
2.6. Linkers and Loaders.....	37
2.7. Loaders.....	40



SILVER OAK  
 UNIVERSITY  
 EDUCATION TO INNOVATION

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

## 1. Overview of Computing Technology

### 1.1. Programming Language Grammars

A programming language grammar defines the syntax and structure of valid programs in that language. It consists of tokens (basic building blocks like keywords, identifiers, operators, etc.) and rules for combining these tokens into valid program statements.

Programming language grammars are formal systems that describe the syntax and structure of programming languages. They consist of rules and symbols that define how valid programs in a language should be written. There are two main types of grammars used to describe programming languages:

#### Context-Free Grammars (CFG):

- These are a set of rules used to define the syntax of a programming language.
- Comprised of terminal and non-terminal symbols, CFGs define the structure of valid statements or expressions.
- The rules in a CFG are expressed in the form of productions, showing how symbols can be replaced by other symbols.
- Popular tools like Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF) are often used to represent CFGs in a human-readable way.

#### Regular Expressions:

- While not as comprehensive as CFGs, regular expressions are used for lexical analysis or tokenization in programming languages.
- They describe patterns of strings using specific syntax rules to match character sequences.
- Often used in defining the rules for individual tokens like identifiers, keywords, operators, etc., within a programming language.


These grammars play a crucial role in various stages of the compilation process, from tokenization and parsing to code generation and optimization. They define the formal rules that ensure the correct structure and interpretation of code written in a specific programming language.

### 1.2. Classification of Grammar

Programming language grammars are typically classified into four types based on their expressive power, as per the Chomsky hierarchy:

- **Type 0 (Unrestricted grammars):** These are the most powerful and can generate any language. They are not often used in programming language design due to their complexity.

- Description: These grammars have the most generative power and can generate any recursively enumerable language.
- Characteristics:
  - Rules are not restricted in form.
  - Can describe languages with complex structures.
- Example: Turing machines and Post systems.
- **Type 1 (Context-sensitive grammars):** These grammars define languages where rules can be applied based on the context. They are used in some programming languages' specifications.
  - Description: These grammars generate context-sensitive languages.
  - Characteristics:
    - Rules have the form  $\alpha A \beta \rightarrow \alpha \gamma \beta$ , where  $A$  is a non-terminal and  $\alpha, \beta, \gamma$  are strings of terminals and/or non-terminals, with  $|\alpha \gamma \beta| \geq |A|$ .
    - Can describe languages where the production of a symbol depends on its context.
  - Example: Some natural languages and some formal language theories.
- **Type 2 (Context-free grammars):** Most programming languages are based on context-free grammars. These have rules where the left-hand side consists of a single non-terminal symbol.
  - Description: These grammars generate context-free languages.
  - Characteristics:
    - Rules have the form  $A \rightarrow \beta$ , where  $A$  is a non-terminal and  $\beta$  is a string of terminals and/or non-terminals.
    - The left-hand side consists of a single non-terminal.
  - Example: Most programming languages' syntax is described by context-free grammars. Example: Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).
- **Type 3 (Regular grammars):** These are the simplest and are used to define regular languages. Regular expressions are an example of this type and are used in lexical analysis.
  - Description: These grammars generate regular languages.
  - Characteristics:
    - Rules have the form  $A \rightarrow aB$  or  $A \rightarrow a$ , where  $A$  and  $B$  are non-terminals,  $a$  is a terminal, and  $\epsilon$  represents an empty string.
    - Express simple patterns that can be recognized by finite automata.
  - Example: Regular expressions and finite automata.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

### 1.3. Implications in Programming Languages

- Type 0 and Type 1: Languages generated by these grammars are extremely complex and aren't commonly used in programming languages due to their difficulty in parsing and managing.
- Type 2: Context-free grammars are widely used for defining the syntax of programming languages. They provide a good balance between expressiveness and ease of parsing.
- Type 3: Regular grammars are utilised in lexical analysis (tokenization) of programming languages due to their simplicity and ability to be recognized by finite automata.

### 1.4. Ambiguity in Grammatic Specification

Ambiguity occurs when a grammar allows for multiple interpretations of a given program statement. For instance, a statement might be parsed in more than one way, leading to confusion for the compiler or interpreter. Ambiguity needs to be resolved in language design or parsing to ensure a clear and unambiguous interpretation of code.

### 1.5. Scanning


Scanning, also known as lexical analysis, is the first phase of the compiler. It involves breaking the input stream of characters into meaningful chunks called tokens. Tokens are the smallest units of syntax in a programming language and serve as the input for the subsequent phase, parsing. Scanning involves identifying keywords, identifiers, operators, and literals within the source code.

### 1.6. Parsing:

Parsing is the process of analysing a sequence of tokens to determine its grammatical structure according to a given grammar. It's a crucial step in language processing and involves building a parse tree or abstract syntax tree (AST) that represents the syntactic structure of the input.

#### 1.6.1. Top-Down Parsing

Top-down parsing starts from the root of the parse tree and tries to construct the tree by starting with the initial symbol of the grammar and working towards the

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

input. Popular top-down parsing techniques include Recursive Descent Parsing and LL Parsing.

- Recursive Descent Parsing: It's a type of top-down parsing where each non-terminal in the grammar is associated with a parsing function. These functions directly correspond to the grammar rules.
- LL Parsing: This refers to a class of parsing methods that read input from Left to right, perform Leftmost derivation and construct Leftmost derivation trees. LL parsers typically use predictive parsing tables or recursive descent.

### 1.6.2. Bottom-Up Parsing

Bottom-up parsing starts from the input symbols and works up to the root of the parse tree. It constructs subtrees until the root of the parse tree is reached. Common bottom-up parsing techniques include Shift-Reduce Parsing and LR Parsing.

- Shift-Reduce Parsing: It operates by shifting input symbols onto a stack until a production's right-hand side is found. Then it reduces the stack's contents based on grammar rules.
- LR Parsing: This method reads input from Left to right and constructs a Rightmost derivation. LR parsers handle a large class of context-free grammars and are efficient in practice. Variants include SLR, LALR, and Canonical LR parsers.

Aspect	Top-Down Parsing	Bottom-Up Parsing
Starting Point	Begins with the start symbol of the grammar	Begins with the input string
Direction	Goes from start symbol towards input	Goes from input towards start symbol
Parse Tree	Constructs a top-down parse tree	Constructs a bottom-up parse tree
Strategy	Tries to apply grammar rules to match input	Uses shift and reduce operations to match grammar rules
Techniques	Recursive Descent, LL Parsing	Shift-Reduce Parsing, LR Parsing



<b>Lookahead</b>	Often requires lookahead for prediction	Doesn't need extensive lookahead
<b>Efficiency</b>	Less efficient for ambiguous grammars	Efficient for a wide range of grammars
<b>Error Detection</b>	Typically detects syntax errors early	Errors may not be detected until later in parsing
<b>Backtracking</b>	May involve backtracking in certain cases	Doesn't usually involve backtracking
<b>Complexity</b>	Easier to implement for simple grammars	More complex for implementation in some cases
<b>Examples</b>	Recursive Descent, LL(1) Parsing	Shift-Reduce Parsing, LR Parsing


### 1.7. Language Processor Development Tools:

**Lexical Analyzers (Lex):** Lex is a tool for generating lexical analyzers (scanners or tokenizers) based on regular expressions. It recognizes patterns in an input stream and converts them into tokens.

**Yacc (Yet Another Compiler Compiler):** Yacc is a parser generator tool that takes a formal grammar description in Backus-Naur Form (BNF) and generates a parser for that grammar. It's used for syntax analysis (parsing) of programming languages.

### 1.8. Overview of Lex & Yacc:

- **Lex:** Lex helps create lexical analyzers. It takes regular expressions and corresponding actions and generates C code for a lexical analyzer. It's often used in combination with Yacc.
- Lex is a lexical analyzer generator that generates code for lexical analyzers or tokenizers in compiler construction.
- It takes regular expressions and generates C code to recognize tokens in the input stream.
- Lex operates by recognizing patterns in the input stream and converting them into tokens or symbols that are meaningful to the parser.
- **Yacc:** Yacc helps generate parsers. It takes a grammar description in BNF and produces C code for a parser. It deals with syntax analysis (parsing) of a language based on the provided grammar rules.
- Yacc (Yet Another Compiler Compiler) is a parser generator that generates code for syntax analyzers or parsers in compiler construction.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY:PROF. SHALU PESHWANI

- It takes context-free grammar as input and generates C code for a parser that can recognize the structure of the input according to the grammar rules.
- Yacc works by building a parsing table based on the grammar rules to analyze the syntax of the input and generate a parse tree.

The workflow often involves using Lex to generate a lexical analyzer that tokenizes the input, and Yacc to generate a parser that processes these tokens according to the specified grammar rules.


Both Lex and Yacc are often used together. Lex generates the scanner to recognize tokens, and Yacc generates the parser that works with these tokens to parse the input according to the grammar rules.

These tools significantly ease the development of language processors by automating the generation of scanners and parsers based on specified rules, reducing manual effort and ensuring consistent parsing based on the language's syntax rules.



**SILVER OAK UNIVERSITY**  
 EDUCATION TO INNOVATION



 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

## 2. Introduction of Computation Tools

### 2.1. Assemblers

An assembler is a type of software or a program that translates assembly language code, which is written by programmers using mnemonic instructions and symbols, into machine code or object code that a computer's processor can directly execute. It serves as an intermediary between human-readable assembly language and the machine language understood by the hardware. Assemblers convert each assembly language instruction into its corresponding binary representation, enabling the computer to perform specific operations as defined by the programmer.

#### 2.1.1. Elements of Assembly Language Programming:

- **Mnemonics:** These are symbolic representations of machine instructions. For example, MOV might represent a "move" operation.
- **Labels and Symbols:** These are used to represent memory addresses or constants. Labels help in referencing memory locations and making the code more readable and maintainable.
- **Directives:** They provide instructions to the assembler rather than being converted into machine code. Examples include directives for defining constants, allocating memory, or including external libraries.
- **Comments:** Annotations within the code that are not interpreted by the assembler but serve as notes for programmers to understand the code.

#### 2.1.2. Types of Assembly Statements:

##### Imperative statement

An imperative statement indicates an action to be performed during the execution of the assembled statement.

Each imperative statement typically translates into one machine instruction.


These are executable statements

Some example of imperative statement are given below  
 MOVER BREG,X  
 STOP  
 READ X  
 PRINT Y  
 ADD AREG,Z

##### Declaration statement

Declaration statements are for reserving memory for variables.



 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

The syntax of declaration statement is as follow:

[Label] DS

\*Label+ DC

DS: stands for Declare storage, DC: stands for Declare constant.

The DS statement reserves area of memory and associates name with them.

A DS 10

Above statement reserves 10 word of memory for variable A.

The DC statement constructs memory words containing constants.

ONE DC '1'

Above statement associates the name ONE with a memory word containing the value '1'

Any assembly program can use constant in two ways- as immediate operands, and as literals.

Many machine support immediate operands in machine instruction.

Ex: ADD AREG, 5

But hypothetical machine does not support immediate operands as a part of the machine instruction. It can still handle literals.

A literal is an operand with the syntax=". EX: ADD AREG,='5'

It differs from constant because its location cannot be specified in assembly program.

### Assembler Directive

Assembler directives instruct the assembler to perform certain action during the assembly program.

**START**


This directive indicates that first word of machine should be placed in the memory word with address .

START Ex: START 500

First word of the target program is stored from memory location 500 onwards.

**END**

This directive indicates end of the source program.

	<b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION		SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

The operand indicates address of the instruction where the execution of program should begin.

By default it is first instruction of the program. END Execution control should transfer to label given in operand field.

### 2.1.3. Basic function of Assembler:

- Translate mnemonics opcodes to machine language.
- Convert symbolic operands to their machine addresses.
- Build machine instructions in the proper format
- Convert data constants into machine representation.
- Error checking is provided.
- Changes can be quickly and easily incorporated with a reassembly.
- Variables are represented by symbolic names, not as memory locations.
- Assembly language statements are written one per line. A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonics.

### 2.1.4. Advantages:

- Reduced errors
- Faster translation times
- Changes could be made easier and faster.
- Addresses are symbolic, not absolute
- Easy to remember


### 2.1.5. Disadvantages:

- Assembler language are unique to specific types of computer
- Program is not portable to the computer.
- Many instructions are required to achieve small tasks
- Programmer required knowledge of the processor architecture and instruction set.

### 2.1.6. Translation phase of Assembler:

The six steps that should be followed by the designer

1. Specify the problem
  2. Specify data structure
  3. Define format of data structure
  4. Specify algorithm
  5. Look for modularity
  6. Repeat
- 1 through 5 on modules

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

### Functions / Purpose of Assembler:

An assembler must do the following

1. Generate instruction
  - a. Evaluate the mnemonics in the operation field to produce the machine code
  - b. Evaluate the subfield-fine the value of each symbol. Process literals and assign addresses.
2. Process pseudo ops
  - a. Pass 1 (Define symbol and literals)
    - i. Determine length of machine instruction ( MOTGET)
    - ii. Keep track of location counter (LC)
    - iii. Remember value of symbol until pass 2 (STSTO)
    - iv. Process some pseudo ops(POTGET1)
    - v. Remember literal (LITSTO)
  - b. Pass 2 (Generate object Program)
    - i. Look up value of symbol (STGET)
    - ii. Generate instruction (MOTGET2)
    - iii. Generate data (for DS, DC and Literal)
    - iv. Process pseudo ops (POTGET2)

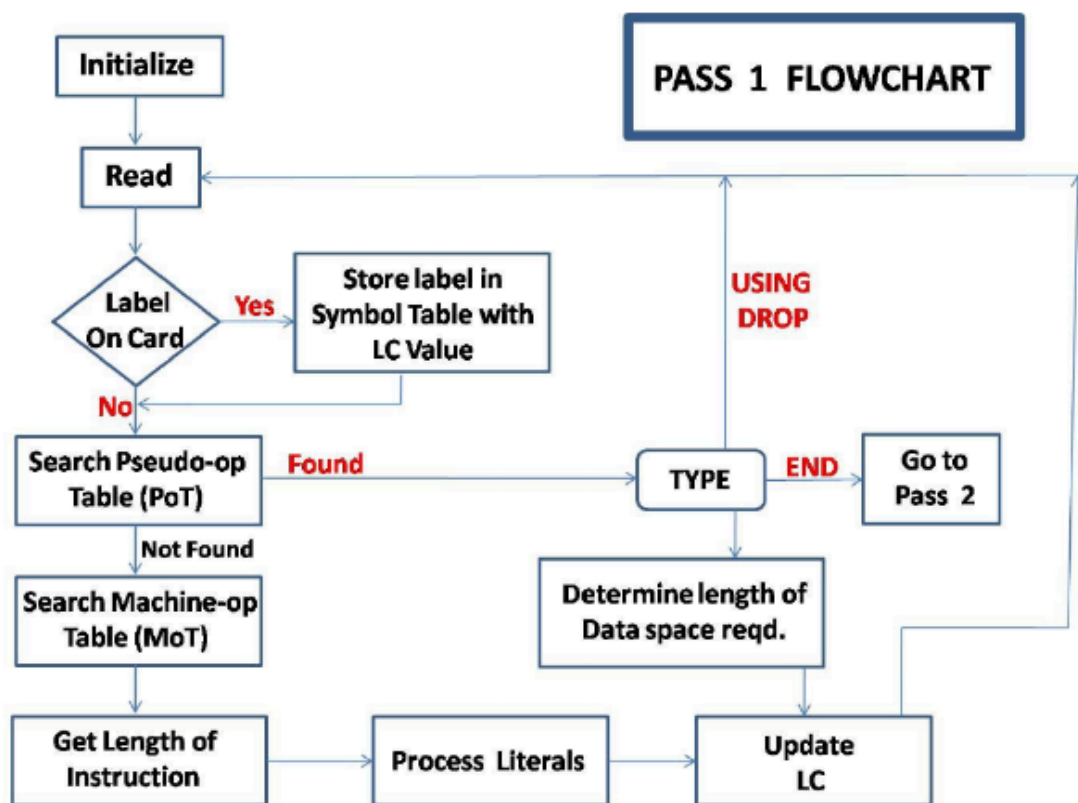
Design data structure for assembler design in Pass-1 and Pass-2 with flow chart:

#### Pass -1

1. Input source program
2. A location counter used to keep track of each instruction location.
3. A table, the machine –operation table (MOT) that indicate the symbolic mnemonics for each instruction and its length (tow, four or six bytes)
4. A table, the pseudo operation table (POT) that indicate the symbolic mnemonics and action to be taken for each pseudo-op in pass-1
5. A table, the literal table (LT) that is used to store each literal encounter and its corresponding assigned location.

6. A table, the symbol table (ST) that is used to store each label and its corresponding value.

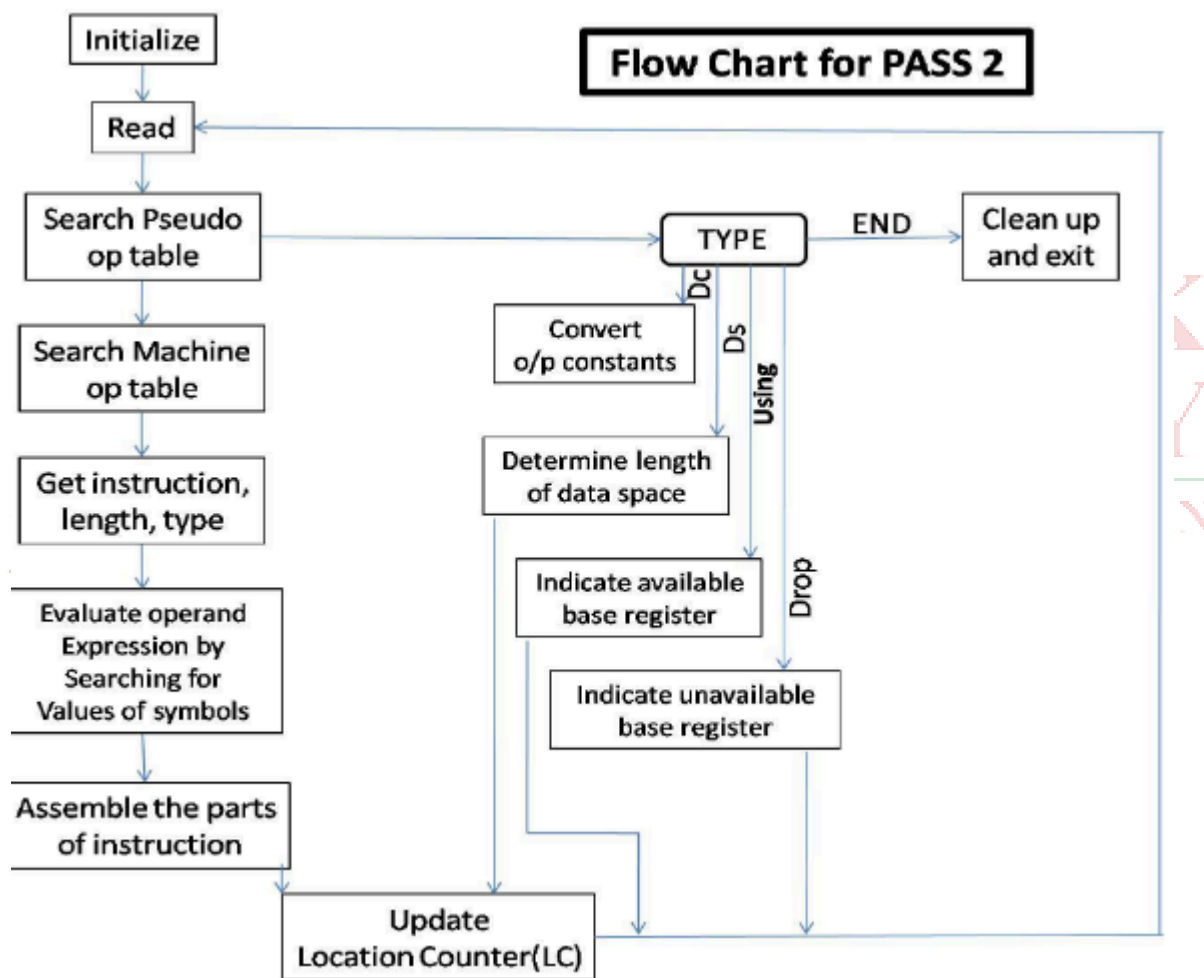
7. A copy of the input to be used later by Pass-2. This may be stored in a secondary storage device.



### Pass 2:


1. Copy of source program input to pass-1
2. Location counter
3. A table the MOT that indicates for each instruction
  - a. Symbolic
  - b. Mnemonics
  - c. Length
  - d. Binary machine op-code
  - e. Format (RR, RS, RX, SI, SS)
4. A table the POT that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in Pass-2
5. The ST prepare by Pass-1, containing each label and its corresponding value.

6. A table, BT that indicate which register are currently specified by base register by USING pseudoops and what are the specified contents of these register.
7. A work space INSR that is to hold each instruction as its various parts are being assembled together.
8. A workspace PRINT LINE used to produce a printed listing
9. A workspace PUNCH CARD used prior to actual outputting for converting assembled instruction into the format needed by the loader.
10. An output deck of assembled instruction in the format needed by the loader.



### 2.1.7. Types of Assemblers:

- One-pass Assemblers: These assemblers read the source code only once, sequentially translating instructions into machine code. They often work with simpler assembly languages but might face challenges with forward referencing.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- **Multi-pass Assemblers:** These assemblers make multiple passes over the source code. In the first pass, they gather information about labels and symbols, and in subsequent passes, they generate the machine code. They can handle more complex assembly languages and resolve forward references.
- **Macro Assemblers:** These assemblers support macros, allowing programmers to define reusable code templates. They expand macros into assembly language instructions during the assembly process.
- **Cross Assemblers:** Cross assemblers generate machine code for a different type of processor or architecture than the one on which they run. They are used to develop software for embedded systems or when the target system isn't readily available.

Assemblers are critical in the software development process, translating human-readable assembly language code into machine code that computers can execute, bridging the gap between low-level programming and the hardware architecture.


## 2.2. Compilers

A compiler is a specialised software tool that translates source code written in a high-level programming language into an equivalent code in a lower-level language or machine code. Its primary function is to convert human-readable code written in languages like C, Java, Python, etc., into a form that a computer's processor can directly execute.

### 2.2.1. Key Functions of a Compiler:

- **Parsing:** The compiler analyzes the syntax of the source code and builds a parse tree or an abstract syntax tree (AST).
- **Semantic Analysis:** It verifies the semantics of the code, ensuring that it adheres to the language's rules and constraints.
- **Optimization:** The compiler may perform various optimizations to improve the efficiency and performance of the generated code without changing its functionality.
- **Code Generation:** Finally, it generates the target code, which could be machine code, bytecode, or an intermediate representation that can be further processed or executed.

### 2.2.2. Compiler Components:

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Frontend: Handles parsing, syntax analysis, and semantic checks based on the language's grammar and rules.
- Middleend: Performs optimizations on the intermediate representation of the code, enhancing its efficiency.
- Backend: Translates the optimized intermediate representation into the target code suitable for execution by the hardware.

### 2.2.3. Significance of Compilers:


- They enable programmers to write code in high-level languages, abstracting away complexities, and then convert it into machine code that the computer understands.
- Compilers ensure portability by allowing the same code to be executed on different hardware platforms by generating machine code specific to each platform.
- They play a crucial role in software development by automating the process of converting human-readable code into machine-executable instructions, thereby speeding up the development process and reducing human error.

### 2.2.4. Semantic Gap in Compilers:

The semantic gap in compilers refers to the differences in meaning and representation between high-level programming languages and the low-level machine code that computers execute. Several factors contribute to a **large semantic gap in compilers**:

- Abstraction Levels:
  - High-level programming languages often provide abstractions that hide low-level details from the programmer. These abstractions may not have direct counterparts in machine code, leading to a semantic gap.
- Optimizations:
  - Compilers perform various optimizations to enhance the efficiency of the generated code. These optimizations may involve restructuring code, reordering instructions, or eliminating redundancy. As a result, the optimized code may differ significantly from the original source code in terms of execution semantics.
- Platform Differences:
  - The target platform's architecture and capabilities can introduce a semantic gap. High-level languages are designed to be




 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

platform-independent, but compilers must generate code that works efficiently on the specific architecture of the target machine.

- Memory Management:
  - High-level languages often abstract memory management, providing features like automatic memory allocation and garbage collection. In contrast, low-level machine code requires explicit memory management. The translation from high-level constructs to machine code may involve complex memory-related transformations.
- Data Types and Structures:
  - High-level languages support rich data types, structures, and abstractions that may not directly map to the more limited set of data types available at the machine code level. For example, complex data structures in high-level languages may be implemented using a combination of simpler structures in machine code.
- Concurrency and Parallelism:
  - High-level languages may offer constructs for parallelism and concurrency, such as threads or parallel loops. Translating these constructs to efficient machine code that takes advantage of the target machine's parallel capabilities can be challenging and may result in a semantic gap.
- Error Handling:
  - High-level languages often have built-in mechanisms for error handling, exceptions, and runtime checks. Compilers need to generate code that preserves the intended error-handling semantics while adhering to the limitations of the target machine.
- Functionality Abstraction:
  - High-level languages may provide high-level abstractions and functionalities that are not directly expressible in machine code. Compilers must find ways to represent these functionalities in terms of the available machine instructions, which can introduce a semantic gap.

#### 2.2.5. Binding and Binding Times:

Binding in the context of compilers refers to the association between a name (such as a variable or function) and the corresponding entity (such as a memory location or code). The timing at which this association is established is known as the binding

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY:PROF. SHALU PESHWANI

time. There are different types of binding and corresponding binding times in compilers:

### Types of Binding:

#### 1. Static Binding:

- Description: Binding is done at compile-time.
- Characteristics:
  - The association between names and entities is established during the compilation phase.
  - The binding remains fixed during the entire program execution.

#### 2. Dynamic Binding:

- Description: Binding is done at runtime.
- Characteristics:
  - The association between names and entities is determined during the program's execution.
  - This often involves late binding, where the actual binding occurs when the program is running.

### Binding Times:

#### 1. Compile Time:


- Description: Binding occurs during the compilation phase.
- Characteristics:
  - Variable names are associated with memory addresses or storage locations.
  - Static scoping is common, where the scope of a variable is determined at compile-time.

#### 2. Load Time:

- Description: Binding occurs when the program is loaded into memory before execution.
- Characteristics:
  - The addresses of libraries or external modules can be determined and resolved during the loading phase.

#### 3. Runtime:

- Description: Binding occurs during program execution.
- Characteristics:
  - Dynamic scoping is possible, where the scope of a variable is determined dynamically during runtime.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- - Dynamic memory allocation allows the association of names with memory locations to be deferred until the program runs.
- Examples:

#### Static Binding Example:

`int x = 10; // Binding of 'x' to a memory location happens at compile-time.`

#### - Dynamic Binding Example:

`def add(a, b):`

`return a + b`

`result = add(3, 5)`

- The binding of the function name `add` to the actual code happens at runtime.

#### Load Time Example:

- In languages that support linking to external libraries, the addresses of functions in those libraries are determined during the loading phase.

#### Runtime Example:

`for (int i = 0; i < n; i++) {`


`// The binding of 'i' to a memory location happens at runtime.`

`}`

Understanding binding and binding times is crucial for compiler designers and programmers as it influences the behavior, efficiency, and scope resolution of programs. The choice of binding time affects how variables, functions, and other program entities are associated with their corresponding representations in memory or code.

### **2.2.6. Data Structures Used in Compiling:**

- Abstract Syntax Trees (AST): Represents the syntactic structure of the source code after parsing, used for subsequent phases like semantic analysis and code generation.
- Symbol Table: Stores information about identifiers (variables, functions) in the source code, including their types, scope, and memory locations.
- Intermediate Code: An intermediate representation of the source code that simplifies analysis and optimization before generating machine code.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

## 2.2.7. Scope Rules:

### 2.7.1. Static Scoping:

#### Definition:

Static scoping is a scope rule where the scope of a variable is determined at compile-time.

#### Characteristics:

The scope of a variable is fixed and known before the program runs.

Lexical scoping is based on the program's structure and the location of variable declarations in the source code.

#### Example (in a pseudo-code representation):

```
def outer():
    x = 10
    def inner():
        * print(x) # Accesses the 'x' from the outer scope
    inner()
outer()
```

#### Advantages:

- Predictable behavior.
- Easier to reason about the program's structure.

#### Disadvantages:

- Limited flexibility in certain scenarios.

### 2.7.2. Dynamic Scoping:


#### Definition:

- Dynamic scoping is a scope rule where the scope of a variable is determined at runtime.

#### Characteristics:

- The scope of a variable is based on the program's execution path.
- The visibility of a variable depends on the call stack at runtime.

#### Example (in a pseudo-code representation):

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

```
def outer():
    x = 10
    inner()
def inner():
    print(x) # Accesses the 'x' from the calling function's scope
outer()
```

#### Advantages:

- More flexibility in certain situations.
- Allows for dynamic changes in scope.

#### Disadvantages:

- Harder to predict and understand the program's behavior.
- May lead to subtle bugs and difficulties in debugging.

#### **2.2.8. Scope Resolution:**

- Static Scoping:
  - The resolution of identifiers is based on the lexical structure of the program.
- Dynamic Scoping:
  - The resolution of identifiers is based on the runtime call stack.

##### **2.8.1. Language Examples:**

- Languages with Static Scoping:
- Most modern programming languages, including C, C++, Java, Python, and many others, use static scoping by default.
- Languages with Dynamic Scoping:
- Some older languages (e.g., Lisp) and some scripting languages (e.g., Bash) use dynamic scoping.


#### **2.2.9. Hybrid Approaches:**

Some languages may allow a combination of both static and dynamic scoping or provide mechanisms for dynamic scoping within specific contexts.

Choosing between static and dynamic scoping often depends on the language design goals and the desired trade-offs between predictability, flexibility, and ease of understanding the program's behavior. Most mainstream languages today opt for static scoping due to its advantages in terms of predictability and readability.

#### **2.2.10. Memory Allocation:**

Memory allocation in compilers involves the assignment and management of memory for variables and data structures during the execution of a program. The allocation of memory is a crucial aspect of the compilation process, and it can occur

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

through various mechanisms. Here are some key concepts related to memory allocation in compilers:

### 1. Static Memory Allocation:

- Description: Memory is allocated at compile-time.
- Characteristics:
  - The size and location of variables are determined during the compilation phase.
  - Memory remains fixed throughout the program's execution.
- Common in global variables and constants.
- Example (in C):

```
int globalVariable; // Static memory allocation
```

### 2. Dynamic Memory Allocation:

- Description: Memory is allocated at runtime.
- Characteristics:
  - Allocation and deallocation of memory occur during program execution.
  - Allows for flexibility in managing memory based on program requirements.
  - Commonly used for data structures like arrays and linked lists.
- Example (in C using `malloc`):

```
int *dynamicArray = (int*)malloc(5 * sizeof(int)); // Dynamic memory allocation
```


### 3. Stack Allocation:

- Description: Memory is allocated on the program's call stack.
- Characteristics:
  - Memory is automatically managed by a stack data structure.
  - Well-suited for managing local variables and function calls.
  - Memory is reclaimed when a function exits.
- Example (in C):

```
void someFunction() {
    int localVar; // Stack allocation
}
```

### 4. Heap Allocation:

- Description: Memory is allocated from the heap, a region of memory managed by the program.
- Characteristics:
  - Dynamic memory allocation typically involves the heap.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Requires explicit allocation and deallocation (using `malloc`, `free` in C).
- Memory may persist beyond the scope of a function.
- Example (in C using `malloc`):

```
int *dynamicVariable = (int*)malloc(sizeof(int)); // Heap allocation
```

### 5. Memory Management Units (MMU):

- Description: Hardware-based memory management.
- Characteristics:
  - Translates virtual addresses to physical addresses.
  - Allows for memory protection and isolation between processes.
  - Facilitates the use of virtual memory.

### 6. Garbage Collection:

- Description: Automatic memory management for languages with dynamic memory allocation.
- Characteristics:
  - Identifies and reclaims memory that is no longer in use.
  - Eliminates manual memory deallocation responsibilities for the programmer.
  - Common in languages like Java, C#, and Python.

### 7. Memory Alignment:


- Description: Ensures that data is stored at memory addresses that are multiples of a specific size.
- Characteristics:
  - Improves performance by allowing the CPU to access data more efficiently.
  - Alignment requirements depend on the hardware architecture.

Memory allocation strategies can significantly impact a program's performance, reliability, and scalability. Compiler designers and programmers must consider these strategies based on language design, platform characteristics, and the specific requirements of the application.

#### 2.2.11. Compilation of Expressions:

- Parsing: Converts expressions into a parse tree or an abstract syntax tree.
- Semantic Analysis: Ensures expressions adhere to the language's rules and resolves identifiers or types.
- Code Generation: Translates expressions into machine code or intermediate code.



 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

### 2.2.12. Compilation of expression:

The compilation of expressions involves translating high-level programming language expressions into equivalent machine code or intermediate code. This process typically occurs during the code generation phase of a compiler. Here are the general steps involved in the compilation of expressions:

#### 1. Lexical Analysis (Tokenization):

- The expression is broken down into tokens, representing the basic units of the language (e.g., identifiers, operators, literals).

#### 2. Syntax Analysis (Parsing):

- The tokens are analyzed to create a syntactic structure, such as a parse tree or an abstract syntax tree (AST). The parse tree represents the hierarchical structure of the expression.

#### 3. Semantic Analysis:

- The compiler performs semantic analysis to ensure that the expression adheres to the language's rules and resolves any ambiguities.
- Type checking is often performed to ensure that the types of operands are compatible with the operators.

#### 4. Intermediate Code Generation:

- An intermediate representation of the expression is generated. This could be in the form of three-address code, bytecode, or another intermediate language.

- Example (Three-address code):

$$t1 = \text{operand1} + \text{operand2}$$

#### 5. Code Optimization (Optional):


- The compiler may apply various optimization techniques to improve the efficiency of the generated code.

- Common optimizations include constant folding, common subexpression elimination, and strength reduction.

#### 6. Target Code Generation:

- The intermediate code is translated into the target machine code or assembly language.

- The generated code depends on the specific architecture and instruction set of the target machine.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

#### 7. Register Allocation:

- If the target machine has a limited number of registers, the compiler performs register allocation to optimize the usage of registers for variables.

#### 8. Memory Allocation:

- Memory is allocated for temporary variables and any other storage needed during the execution of the expression.

- For dynamic memory allocation, heap management may be involved.

#### 9. Exception Handling (Optional):

- If the language supports exception handling, the compiler generates code to handle exceptions that may arise during the evaluation of the expression.

#### 10. Final Code Emission:

- The compiled code is emitted or generated, ready for execution on the target machine.

#### Example (In C):

Consider the expression `result = a + b * (c - d) / e;`

#### 1. Lexical Analysis (Tokenization):

- `'result', '=', 'a', '+', 'b', '*', '(', 'c', '-', 'd', ')', '/', 'e', ';'``

#### 2. Syntax Analysis (Parsing):

- Construct a parse tree or an abstract syntax tree.

#### 3. Semantic Analysis:

- Ensure that the types of `'a', 'b', 'c', 'd',` and `'e'` are compatible for the arithmetic operations.

#### 4. Intermediate Code Generation:


- Generate intermediate code (e.g., three-address code) based on the expression.

#### 5. Code Optimization (Optional):

- Apply optimizations to the intermediate code.

#### 6. Target Code Generation:

- Translate the optimized intermediate code into machine code or assembly language.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

#### 7. Register Allocation:

- Allocate registers for variables, ensuring efficient use.

#### 8. Memory Allocation:

- Allocate memory for temporary variables and other storage.

#### 9. Exception Handling (Optional):

- Generate code to handle exceptions, if applicable.

#### 10. Final Code Emission:

- Emit the compiled code for execution.

The above steps may vary slightly based on the specifics of the compiler and the programming language being compiled. The compilation of expressions is a complex process that requires careful consideration of language semantics, optimizations, and target machine architecture.

### 2.2.13. Compilation of expression

The compilation of control structures involves translating high-level programming language constructs, such as if statements, loops, and switches, into equivalent machine code or intermediate code. This process is a part of the code generation phase of a compiler. Below are the general steps involved in the compilation of control structures:

#### 1. Lexical Analysis (Tokenization):

- The source code is broken down into tokens, representing the basic units of the language.


#### 2. Syntax Analysis (Parsing):

- The tokens are analyzed to create a syntactic structure, such as a parse tree or an abstract syntax tree (AST). The parse tree represents the hierarchical structure of the control structures.

#### 3. Semantic Analysis:

- The compiler performs semantic analysis to ensure that the control structures adhere to the language's rules.
- Type checking and scope analysis are performed to ensure correctness.

#### 4. Intermediate Code Generation:

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- An intermediate representation of the control structures is generated. This could be in the form of three-address code, bytecode, or another intermediate language.

- Example (Three-address code):

c

if (condition) goto L1;

else goto L2;

L1: // code for if-true

goto L3;

L2: // code for if-false

L3:

5. Code Optimization (Optional):

- The compiler may apply various optimization techniques to improve the efficiency of the generated code.

- Common optimizations include loop unrolling, dead code elimination, and conditional branch optimization.

6. Target Code Generation:

- The intermediate code is translated into the target machine code or assembly language.

- Branch instructions and conditionals are generated based on the control structures.

7. Register Allocation:

- If the target machine has a limited number of registers, the compiler performs register allocation to optimize the usage of registers during control flow.


8. Memory Allocation:

- Memory is allocated for any variables or data structures used within the control structures.

- For dynamic memory allocation, heap management may be involved.

9. Exception Handling (Optional):

- If the language supports exception handling, the compiler generates code to handle exceptions that may arise within the control structures.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

#### 10. Final Code Emission:

- The compiled code is emitted, ready for execution on the target machine.

Example (In C):

Consider the control structure:

```
if (x > 0) {
    // code for if-true
} else {
    // code for if-false
}
```

#### 1. Lexical Analysis (Tokenization):

- `if`, `(`, `x`, `>`, `0`, `)`, `{`, `// code for if-true`, `}`, `else`, `{`, `// code for if-false`, `}`.

#### 2. Syntax Analysis (Parsing):

- Construct a parse tree or an abstract syntax tree.

#### 3. Semantic Analysis:

- Ensure that the types of `x` and `0` are compatible for the comparison.

#### 4. Intermediate Code Generation:

- Generate intermediate code for the control structure.

#### 5. Code Optimization (Optional):

- Apply optimizations to the intermediate code.


#### 6. Target Code Generation:

- Translate the optimized intermediate code into machine code or assembly language.

#### 7. Register Allocation:

- Allocate registers for variables, ensuring efficient use.

#### 8. Memory Allocation:

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Allocate memory for any variables used within the control structure.

#### 9. Exception Handling (Optional):

- Generate code to handle exceptions, if applicable.

#### 10. Final Code Emission:

- Emit the compiled code for execution.

The compilation of control structures is essential for transforming high-level language constructs into executable machine code. The process requires careful consideration of language semantics, optimization techniques, and target machine architecture.

### 2.2.14. Compilation of Control structures

The compilation of control structures involves translating high-level programming language constructs, such as if statements, loops, and switches, into equivalent machine code or intermediate code. This process is a part of the code generation phase of a compiler. Below are the general steps involved in the compilation of control structures:

#### 1. Lexical Analysis (Tokenization):

- The source code is broken down into tokens, representing the basic units of the language.

#### 2. Syntax Analysis (Parsing):

- The tokens are analyzed to create a syntactic structure, such as a parse tree or an abstract syntax tree (AST). The parse tree represents the hierarchical structure of the control structures.

#### 3. Semantic Analysis:

- The compiler performs semantic analysis to ensure that the control structures adhere to the language's rules.

- Type checking and scope analysis are performed to ensure correctness.


#### 4. Intermediate Code Generation:

- An intermediate representation of the control structures is generated. This could be in the form of three-address code, bytecode, or another intermediate language.

- Example (Three-address code):

c

if (condition) goto L1;

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

else goto L2;

L1: // code for if-true

goto L3;

L2: // code for if-false

L3:

#### 5. Code Optimization (Optional):

- The compiler may apply various optimization techniques to improve the efficiency of the generated code.
- Common optimizations include loop unrolling, dead code elimination, and conditional branch optimization.

#### 6. Target Code Generation:

- The intermediate code is translated into the target machine code or assembly language.
- Branch instructions and conditionals are generated based on the control structures.

#### 7. Register Allocation:

- If the target machine has a limited number of registers, the compiler performs register allocation to optimize the usage of registers during control flow.

#### 8. Memory Allocation:

- Memory is allocated for any variables or data structures used within the control structures.
- For dynamic memory allocation, heap management may be involved.

#### 9. Exception Handling (Optional):

- If the language supports exception handling, the compiler generates code to handle exceptions that may arise within the control structures.


#### 10. Final Code Emission:

- The compiled code is emitted, ready for execution on the target machine.

Example (In C):

Consider the control structure:



 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

```

if (x > 0) {
    // code for if-true
} else {
    // code for if-false
}

```

#### 1. Lexical Analysis (Tokenization):

- `if`, `(`, `x`, `>`, `0`, `)`, `{`, `// code for if-true`, `}`, `else`, `{`, `// code for if-false`, `}`.

#### 2. Syntax Analysis (Parsing):

- Construct a parse tree or an abstract syntax tree.

#### 3. Semantic Analysis:

- Ensure that the types of `x` and `0` are compatible for the comparison.

#### 4. Intermediate Code Generation:

- Generate intermediate code for the control structure.

#### 5. Code Optimization (Optional):

- Apply optimizations to the intermediate code.

#### 6. Target Code Generation:

- Translate the optimized intermediate code into machine code or assembly language.

#### 7. Register Allocation:

- Allocate registers for variables, ensuring efficient use.

#### 8. Memory Allocation:


- Allocate memory for any variables used within the control structure.

#### 9. Exception Handling (Optional):

- Generate code to handle exceptions, if applicable.

#### 10. Final Code Emission:

- Emit the compiled code for execution.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

The compilation of control structures is essential for transforming high-level language constructs into executable machine code. The process requires careful consideration of language semantics, optimization techniques, and target machine architecture.

### 2.2.15. Code Optimization:

- **Code Optimization:** Aims to improve program performance by transforming code to consume fewer resources (time, memory), reduce redundancy, and enhance efficiency without altering its functionality.
- **Types of Optimizations:** These include loop optimization, constant folding, dead code elimination, register allocation, and more.

### 2.3. Interpreters:

- **Definition:** An interpreter is a program that directly executes source code written in a high-level programming language line by line without translating it into machine code beforehand.
- **Execution Process:** It reads the source code, parses it, and executes it directly, translating and executing each line or block of code on-the-fly.
- **Advantages:**
  - Quick start-up as there's no separate compilation step.
  - Easier to implement and portable across platforms.
  - Allows for interactive development and immediate feedback.
- **Disadvantages:**
  - Slower execution compared to compiled code as it interprets code on-the-fly.
  - Generally less efficient in terms of performance.
- **Examples:** Python, JavaScript, Ruby, and many scripting languages often use interpreters.

### 2.4. Debuggers:

- **Definition:** A debugger is a software tool that allows developers to identify and correct errors (bugs) in their programs by examining code execution, variables, and program state.
- **Features:**
  - **Breakpoints:** Developers can pause the program's execution at specific points to inspect the state of variables and code flow.
  - **Variable Inspection:** Allows for the examination of variable values during program execution.
  - **Step-through Execution:** Enables developers to execute the code line by line or step by step to track down issues.
  - **Call Stack Inspection:** Helps in understanding the sequence of function calls and their contexts.
- **Types of Debuggers:**



- Integrated Development Environment (IDE) Debuggers: Integrated into IDEs like Visual Studio, Eclipse, or Xcode.
- Command-Line Debuggers: Standalone tools accessed through the command line interface.
- Language-Specific Debuggers: Tailored for specific programming languages.
- Debugging Procedures:
  - Setting Breakpoints: Identify specific points in the code where execution should pause for inspection.
  - Stepping Through Code: Step through code execution line by line or statement by statement, observing variable values and program flow.
  - Variable Inspection: Examine variable values at different stages of execution to identify discrepancies or unexpected values.
  - Call Stack Inspection: Analyze the call stack to understand the sequence of function calls and their contexts.
  - Watching Expressions: Monitor the value of specific expressions or variables during program execution.
  - Memory Inspection: Check memory states and memory usage during runtime for potential issues.
- Usage: Debuggers are crucial during software development for diagnosing and fixing errors, ensuring code correctness, and improving code quality.

#### 2.4.1. Relationship:

- Usage Together: While an interpreter executes code directly, debuggers can be used alongside interpreters to analyze and troubleshoot code during its execution. Developers can set breakpoints, inspect variables, and step through code to identify and fix issues in real-time.
- Improving Code Quality: Combining interpreters and debuggers allows developers to iterate quickly on code, test changes immediately, and refine the code, thereby improving its quality before final deployment.

Aspect	Compiler	Interpreter
<b>Translation Process</b>	Translates entire source code at once	Translates and executes code line by line
<b>Output</b>	Generates executable or object code	No separate output; executes source directly
<b>Execution Speed</b>	Generally faster as it generates optimized machine code	Slower as it interprets code on-the-fly
<b>Optimization</b>	Can perform high-level optimizations	Minimal optimization due to immediate execution



<b>Portability</b>	Output code can be run on multiple platforms with appropriate compilation	Requires an interpreter specific to each platform
<b>Memory Usage</b>	Lower memory usage as compiled code is executed	Higher memory usage due to continuous interpretation
<b>Debugging</b>	Debugging may be more challenging due to indirect mapping of source to machine code	Easier to debug as it works directly with source code
<b>Examples</b>	C, C++, Java, compiled languages	Python, JavaScript, Perl, interpreted languages

## 2.5. Macro and Macro Processors

Macro definition and calls, expansion, and nested macro calls are concepts commonly associated with preprocessors in programming languages like C and C++. These features allow developers to define and use macros, which are essentially text substitutions.

Salient features of Macro Processor:

- Macro represents a group of commonly used statements in the source programming language.
- Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as the expansion of macros.
- Using Macro instructions programmer can leave the mechanical details to be handled by the macro processor.
- Macro Processor designs are not directly related to the computer architecture on which it runs.
- Macro Processor involves definition, invocation, and expansion.

### 2.5.1. Macro Definition and Call:


- Definition: A macro is defined using `#define` in C/C++. It's a preprocessor directive that associates an identifier with a token sequence or code snippet.

Example: `#define PI 3.14159`

- Call: Once defined, macros are invoked using their names.

Example: `float circle_area = PI * radius * radius;`

### 2.5.2. Macro Expansion:

	<b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION		SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Expansion: When a macro is called, the preprocessor replaces the macro name with its associated code before the actual compilation of the program begins. In the example above, PI would be replaced by 3.14159 during the preprocessing stage.

### 2.5.3. Nested Macro Calls:

- Nested Macros: Macros can also contain references to other macros, including themselves.

Example: `#define MAX(x, y) ((x) > (y) ? (x) : (y))`  
`#define SQUARE(x) ((x) * (x))`

- Nested Calls: Macros can be nested within other macros.

Example: `int larger_num = MAX(SQUARE(a), SQUARE(b));`

In this case, `SQUARE(a)` and `SQUARE(b)` are nested within the `MAX` macro call.

- Nested macro calls can be powerful, but they require caution to avoid unexpected behaviour due to interactions between different macro expansions.

### 2.5.4. Design of a Macro Preprocessor

Designing a macro preprocessor involves defining its functionality, syntax, and behaviour. Here's an outline of steps for designing a simple macro preprocessor:

#### Define Syntax and Features:

- Determine the syntax for defining macros (`#define`) and using them in code.
- Decide on the features: support for function-like macros, variable arguments, nested macros, conditional macros (`#ifdef`, `#ifndef`, etc.).

#### Lexical Analysis:


- Implement a lexer to tokenize the input code and recognize preprocessor directives like `#define`, `#ifdef`, etc.
- Tokenize identifiers, literals, symbols, and preprocessors directives.

#### Parsing and Processing:

- Create a parser to understand and interpret the preprocessor directives.
- Store macro definitions in a symbol table or a data structure for efficient lookup.
- Resolve nested macros: Implement logic to handle macros calling other macros.

#### Macro Expansion:

- Implement the logic to replace macro calls with their respective definitions in the code.
- Handle parameters and arguments for function-like macros.

	<b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY:PROF. SHALU PESHWANI	

- Ensure proper handling of whitespace, parentheses, and other syntax elements during expansion.

#### **Conditional Compilation:**

- Support conditional compilation directives like `#ifdef`, `#ifndef`, `#else`, `#endif`, etc.
- Enable or disable parts of the code based on conditional directives.

#### **Error Handling:**

- Implement robust error handling for invalid syntax, undefined macros, circular dependencies, etc.
- Provide clear and informative error messages to aid developers in debugging.

#### **Integration with Compiler:**

- Integrate the preprocessor with the compiler workflow to preprocess the code before actual compilation.
- Ensure seamless communication between the preprocessor and the compiler stages.

#### **Testing and Optimization:**

- Develop comprehensive test cases to verify the functionality of the preprocessor.
- Optimise the preprocessor for performance and memory efficiency.

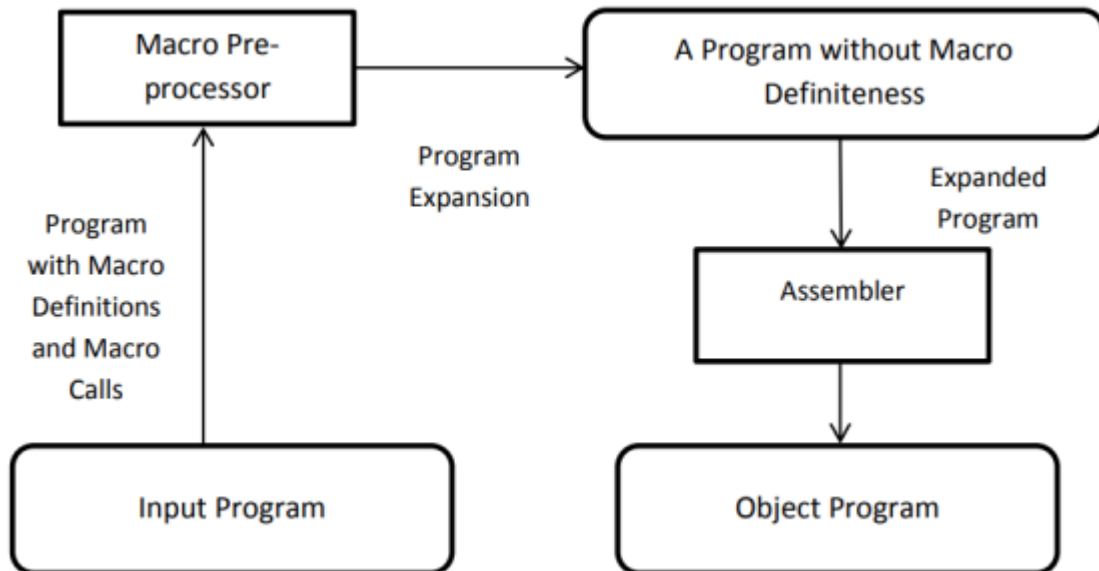
#### **Documentation and Examples:**

- Provide thorough documentation explaining the usage, syntax, and behaviour of the preprocessor.
- Offer examples demonstrating various macro features and their usage.

#### **Iterate and Improve:**

- Gather feedback and iteratively improve the preprocessor based on user experience and requirements.






### 2.5.5. Functions of a Macro Processor

A macro processor serves as a crucial component in a compiler or interpreter, responsible for handling preprocessor directives and performing text substitution before the actual compilation or interpretation takes place. Its functions include:

- **Macro Definition and Expansion:**
  - Define Macros: Allow users to define macros using `#define` directives, associating a name with a token sequence or code snippet.
  - Expand Macros: Replace occurrences of macro names with their corresponding definitions throughout the code.
- **Parameterized Macros:**
  - Support macros with parameters (function-like macros) to enable more versatile code substitutions.
  - Handle arguments passed to these parameterized macros during expansion.
- **Conditional Compilation:**
  - Enable conditional compilation using directives like `#ifdef`, `#ifndef`, `#else`, `#endif`, allowing code inclusion or exclusion based on defined macros or conditions.
- **Include Files:**
  - Process `#include` directives to include external files in the code during preprocessing.
- **Error Handling:**
  - Provide error detection and reporting for issues related to macro definition, expansion, syntax errors in directives, circular dependencies, and undefined macros.
- **Symbol Table Management:**




 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Maintain a symbol table or data structure to store macro definitions for efficient lookup during expansion.
- Nested Macro Handling:
  - Support nested macros, ensuring proper resolution and substitution of multiple levels of macro calls within the code.
- Whitespace Handling and Line Continuation:
  - Manage whitespace, line breaks, and line continuation within macro expansions to ensure correct code structure after substitution.
- Compatibility with Compiler/Interpreter:
  - Integrate seamlessly with the compiler or interpreter workflow, providing preprocessed code for subsequent compilation or interpretation stages.
- Performance Optimization:
  - Optimise the macro processor for efficient handling of large codebases by implementing strategies like caching, efficient symbol lookup, etc.

#### 2.5.6. Design Issues of Macro Processors

Designing a macro processor involves addressing various challenges and considerations to ensure its effectiveness, efficiency, and correctness. Here are key design issues and considerations:

- Macro Expansion Order:
  - Determine the order of macro expansion, considering nested macros and ensuring correct substitution without unintended side effects.
- Scoping and Lifetime of Macros:
  - Define rules for scoping macros within certain blocks or files and handling their visibility and lifespan.
- Parameter Handling:
  - Decide how macro parameters are handled and substituted, considering cases of default arguments, variable arguments, and parameter validation.
- Whitespace and Formatting:
  - Address handling of whitespace and code formatting within macro expansions to maintain code readability and consistency.
- Recursive Macros:
  - Handle recursive macros cautiously to prevent infinite expansions or limit recursion depth.
- Error Handling and Diagnostics:
  - Devise robust error-handling mechanisms to detect and report syntax errors, undefined macros, circular references, and other preprocessing issues.
- Performance and Efficiency:
  - Optimize the macro processor for performance, minimizing processing time and memory usage.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Implement caching or memoization techniques for frequently used macros to reduce redundant processing.
- Compatibility with Language Syntax:
  - Ensure compatibility with the language syntax and semantics to accurately preprocess code without introducing conflicts or ambiguity.
- Conditional Compilation:
  - Implement conditional compilation directives (#ifdef, #ifndef, etc.) accurately, enabling code inclusion or exclusion based on defined conditions.
- Include File Handling:
  - Manage inclusion of external files using #include directives efficiently and securely.
- Debugging Support:
  - Facilitate debugging by providing clear error messages, enabling tracking of macro expansions, and assisting developers in understanding preprocessed code.
- Integration with Toolchain:
  - Ensure seamless integration with the compiler or interpreter, allowing for the preprocessed code to be correctly processed in subsequent stages.


## 2.6. \* Linkers and Loaders

### 2.6.1. Introduction:

Linkers and loaders are essential components of the compilation and execution process for programs:

#### Linkers:

- Function: Linkers are tools responsible for combining multiple object files generated by the compiler into a single executable or library.
- Tasks:
  - Symbol Resolution: Resolves references between different object files by matching symbols (functions, variables) defined in one file with their references in another.
  - Address Binding: Assigns final memory addresses to symbols, creating a coherent address space for the program.
  - Relocation: Adjusts addresses in the object files to reflect the final memory layout of the executable.
  - Generation of Executable/Shared Libraries: Produces executables or shared libraries ready for execution or further distribution.
- Static vs. Dynamic Linking:
  - Static linking combines all necessary code into a single executable file.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Dynamic linking allows sharing of libraries among multiple programs, linking them during runtime.

#### Loaders:

- Function: Loaders are responsible for loading executable files into memory for execution.
- Tasks:
  - Memory Allocation: Allocates memory for the program's code, data, and stack.
  - Address Binding (if dynamic linking): Resolves external references in dynamically linked libraries during runtime.
  - Initialization: Initializes program components before execution.
  - Execution Start: Transfers control to the program's entry point (e.g., main function in C/C++).

#### Key Differences:

- Linkers work before program execution, combining object files and generating the final executable or library.
- Loaders work during program execution, preparing the program for running by allocating memory and resolving dynamic references if needed.

#### Relationship:

- Linking is typically performed before execution by the linker to create a standalone executable or library.
- Loading occurs during runtime when the operating system loads the program into memory.

#### Tools:

- Common linkers include GNU ld, Microsoft Link, and LLVM lld.
- Loaders are part of the operating system and handle program execution and memory allocation.


Both linkers and loaders play critical roles in the execution of programs, ensuring that code from multiple sources is combined, resolved, and executed efficiently in memory. They are integral components of the compilation and execution process in modern computer systems.

### 2.6.2. Design of a Linker:

Designing a linker involves several crucial steps to combine and prepare object files into executable programs or libraries

#### Input Analysis:

- Parsing input object files and understanding their formats (e.g., ELF, COFF) to extract necessary information like symbols, sections, and relocation entries.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

Symbol Resolution:

- Resolving symbols by matching references to definitions across object files.
- Handling symbols defined in one object file and referenced in another.

Address Binding:

- Assigning final addresses to symbols to generate a coherent address space for the program.
- Handling relocation information and adjusting addresses based on the program's memory layout.

Relocation:

- Relocating code and data to appropriate locations in memory, considering the address space layout.

Output Generation:

- Generating the final executable or library file by combining and arranging sections from different input object files.
- Handling symbol tables and debug information for use in debugging tools.

Static vs. Dynamic Linking:

- Support for both static and dynamic linking, as per the requirements and specifications.


### 2.6.3. Static vs. Dynamic Linking:

**Static Linking:**

- Process: Combines all necessary code and libraries into a single executable file.
- Advantages:
  - Produces a standalone executable that doesn't rely on external libraries during runtime.
  - Guarantees that the program will run consistently, as it's not affected by changes in external libraries.
- Disadvantages:
  - Larger executable sizes as all necessary code and libraries are bundled.
  - Potential redundancy if multiple programs use the same library.

**Dynamic Linking:**

- Process: Allows multiple programs to share a single copy of a library, linking it during runtime.
- Advantages:
  - Smaller executable sizes as shared libraries are loaded dynamically at runtime.
  - Allows updates to shared libraries without recompiling the entire program.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

- Disadvantages:
  - Dependency on external libraries during execution, requiring the presence of correct versions of shared libraries.
  - Potential version conflicts or compatibility issues with different versions of shared libraries.

Designing a linker involves handling various complexities like symbol resolution, address binding, and support for different object file formats. Supporting both static and dynamic linking requires careful consideration of the advantages and trade-offs of each approach based on the requirements of the system and the application being developed.

## 2.7. Loaders

Loaders are crucial components of an operating system responsible for loading executable files into memory and preparing them for execution. Here's an overview of the design and functions of loaders:

### 2.7.1. Design of Loaders:

#### Input Handling:

- Recognizing different executable file formats (e.g., ELF, COFF) and parsing them to extract necessary information such as program sections, headers, and entry points.

#### Memory Allocation:

- Allocating memory space for the program's code, data, and stack based on information from the executable file.
- Managing memory layout to ensure proper isolation and protection between processes.

#### Address Binding:

- Resolving memory addresses for relocatable sections of the program.
- Handling relocation information and adjusting addresses to fit the program's memory layout.

#### Symbol Resolution (if dynamic linking):


- Resolving symbols and references to external libraries or shared objects during runtime for dynamically linked programs.
- Loading and linking shared libraries as needed.

#### Initialization:

- Initializing program components such as global variables and static data structures before program execution begins.
- Preparing the execution environment for the program to start.

#### Execution Start:

- Transferring control to the entry point specified in the executable file (e.g., main() function in C/C++) to start the program's execution.

 <b>SILVER OAK UNIVERSITY</b> EDUCATION TO INNOVATION	<u>Computing Architecture and Computation</u>	
LECTURE COMPANION	SEMESTER: VI	PREPARED BY: PROF. SHALU PESHWANI

### Error Handling:

- Handling various error conditions such as insufficient memory, invalid file formats, or missing dependencies.
- Providing meaningful error messages to assist in debugging and troubleshooting.

### 2.7.2. Functions of Loaders:

#### Loading Executable Files:

- Loading the executable file into memory from storage (e.g., disk) for execution.

#### Memory Management:

- Allocating memory space for the program's code, data, and stack according to the executable's requirements.

#### Address Resolution:

- Resolving and adjusting addresses for relocatable sections to fit the program's memory layout.

#### Symbol Resolution (for dynamic linking):

- Resolving external references to shared libraries or objects during runtime for dynamically linked programs.

#### Initialization and Setup:

- Initializing necessary components and preparing the environment for the program's execution.

#### Control Transfer:

- Transferring control to the program's entry point to start its execution.

Loaders play a critical role in the execution of programs, ensuring that executable files are properly loaded into memory, initialised, and prepared for execution in a way that maximises efficiency, security, and correctness. They form an integral part of the operating system's functionality to manage and execute programs on computer systems.