Compiler Design Questions - Complete Solutions

Q1: What is BNF (Backus-Naur Form)?

Answer: BNF is a formal notation used to describe the syntax of programming languages. It represents grammar rules using symbols and productions.

Explanation: Think of BNF like a recipe that defines what valid sentences look like in a language. It uses special symbols:

- <term> or term = Non-terminal (represents a concept that needs further definition)
- ::= or → = "is defined as"
- | = "or" (alternatives)
- Quoted items = Terminals (actual words/tokens)

Example:

```
<Sentence> ::= <Subject> <Verb> <Object>
<Subject> ::= "The" "cat" | "A" "dog"
<Verb> ::= "chases" | "sees"
<Object> ::= "mouse" | "bird"
```

This says a sentence must have subject-verb-object, with specific choices for each.

Q2: What is Context-Free Grammar?

Answer: A Context-Free Grammar (CFG) is a formal grammar where production rules allow a non-terminal to be replaced by any sequence of terminals and non-terminals, regardless of context.

Explanation: The word "context-free" means the replacement rule doesn't depend on what comes before or after it.

Structure: A CFG consists of:

- Non-terminals (variables like A, B, S)
- Terminals (actual symbols/tokens)
- Production rules (S → AB | ε)
- Start symbol (usually S)

Simple Example:

$$S \rightarrow aS \mid b$$

This generates strings like: b, ab, aab, aaab, etc.

Why it matters in compilers: Most programming languages are defined using CFG because it's powerful enough to describe language syntax but still practical to parse.

Q3: What is an ID (Identifier)? How is it different from a token?

Answer:

Identifier: A name given to variables, functions, arrays, etc. in a program (e.g., `myVar`, `count`, `getUserName`).

Token: The smallest unit recognized by a compiler (e.g., keywords, identifiers, operators, literals).

Difference:

Identifier	Token
Specific name chosen by programmer	General category of meaningful symbol
Example: `x`, `sum`, `myFunction`	Example: identifier, keyword, operator
One type of token	Token is a broader concept
Semantic meaning	Lexical unit

In practical terms: When the compiler sees `int myVar;`, it identifies:

- `int` as a keyword token
- `myVar` as an identifier token
- `;` as a punctuation token

Q4: Give an example of an ambiguous grammar

Answer: An ambiguous grammar is one where a single sentence can be derived in more than one way (multiple parse trees).

Example:

```
E → E + E | E * E | (E) | num
```

Why it's ambiguous: Consider parsing `2 + 3 * 4`

Parse Tree 1: (Addition first)

```
  +
 / \
2   *
   / \
  3   4
```

Result: (2 + 3) × 4 = 20

Parse Tree 2: (Multiplication first)

```
    *
   / \
  +   4
 / \
2   3
```

Result: 2 + (3 × 4) = 14

Both are valid derivations from the same grammar, so it's ambiguous! Compilers need unambiguous grammars to parse correctly.

Q5: What is a Symbol Table?

Answer: A symbol table is a data structure used by a compiler to store information about identifiers (variables, functions, arrays, etc.) encountered during compilation.

What it stores:

- Identifier name
- Type (int, float, string, etc.)
- Scope (global, local, function-level)
- Memory address or location
- Size in memory

- Attributes (const, static, etc.)

Why it's important:

- Tracks variable declarations
- Checks for redeclaration errors
- Stores type information for type checking
- Resolves variable scope during semantic analysis
- Generates code with correct memory addresses

Simple Example:

```
Symbol Table for: int x; float y; int func() { int z; }

| Name | Type  | Scope    | Address |
|------|-------|----------|---------|
| x    | int   | global   | 1000    |
| y    | float | global   | 1004    |
| func | func  | global   | 2000    |
| z    | int   | local    | 3000    |
```

Q6: Write down the difference between Rise and CISC

Answer:

RISC	**CISC**
Reduced **I**nstruction **S**et **C**omputer	**C**omplex **I**nstruction **S**et **C**omputer
Fewer, simpler instructions	Many, complex instructions
Each instruction does one thing	Single instruction performs multiple operations
Faster clock cycles	Slower clock cycles
More instruction calls needed	Fewer instructions needed
Examples: ARM, MIPS, PowerPC	Examples: Intel x86, Pentium
Better for: Mobile, embedded	Better for: Desktops, servers
Lower power consumption	Higher power consumption
Easier to design compiler	Complex compiler design

Simple Comparison:

- RISC: Like a restaurant with simple basic dishes (fast, reliable)
- CISC: Like a restaurant with complex combo meals (versatile, but more complex)

Q7: What is type checking with respect to a compiler?

Answer: Type checking is the process of verifying that operations in a program are valid for the data types involved.

Types of Type Checking:

1. **Static Type Checking** (at compile time)
   - Errors caught before running the program
   - Languages: Java, C, C++
   - Example: `int x = "hello";` → Compile error!
2. **Dynamic Type Checking** (at runtime)
   - Errors caught when program runs
   - Languages: Python, JavaScript
   - Example: Python allows `x = 5; x = "hello";` (no error until operation fails)

What it checks:

- Variable assigned value of wrong type
- Function arguments don't match parameter types
- Invalid operations (e.g., adding string and number)
- Array index is an integer
- Return type matches function declaration

Example:

```
int add(int a, int b) {
   return a + b;
}

add(5, 3);       // ✓ Valid
add(5, "3");     // ✗ Type Error - string instead of int
add(5.5, 3);     // ✓ Valid (float converts to int)
```

Q8: What is parameters passing in a computer language?

Answer: Parameter passing is the mechanism of transferring arguments from a function call to the function definition.

Main Methods:

1. **Pass by Value**
   - A copy of the value is passed
   - Changes inside function don't affect original
   - Example: `void func(int x) { x = 10; }` — doesn't change caller's value
2. **Pass by Reference**
   - The actual memory address is passed
   - Changes inside function DO affect original
   - Example: C++ uses `&`, Java passes object references
3. **Pass by Name**
   - The actual expression is passed and re-evaluated each time
   - Rarely used (Algol 60)
4. **Pass by Result**
   - Values returned back to caller
   - Used for output parameters

Practical Example:

```
// Pass by Value
void change(int x) {
    x = 20;  // Only changes local copy
}
int main() {
    int a = 10;
    change(a);
    cout << a;  // Prints 10 (unchanged)
}

// Pass by Reference
void change(int &x) {
    x = 20;  // Changes original
}
int main() {
    int a = 10;
    change(a);
    cout << a;  // Prints 20 (changed!)
}
```

Q9: Difference between call by name, call by value and call by reference in function calls

Answer:

Feature	Call by Value	Call by Reference	Call by Name
What passes	Copy of value	Memory address	The expression itself
Original affected	No	Yes	Sometimes (depends)
Efficiency	Good (small data)	Good (large data)	Poor (re-evaluated)
Storage needed	New memory	No extra memory	Stores expression
Languages	C, Java, Python	C++, C#, PHP	Algol 60
Speed	Fast	Fast	Slow
Example	`func(x)`	`func(&x)`	`func(name: x)`

Real Example:

```
java
```

```
// Call by Value (Java integers)
void increment(int num) {
    num++;  // Only local copy changed
}
int x = 5;
increment(x);
System.out.println(x);  // Prints 5

// Call by Reference (Java objects)
void changeName(Person p) {
    p.name = "New";  // Original object affected!
}
Person person = new Person("Old");
changeName(person);
System.out.println(person.name);  // Prints "New"

// Call by Name (Conceptual)
def increment(name: int x):
    x += 1
// x re-evaluated each time it's used inside function
```

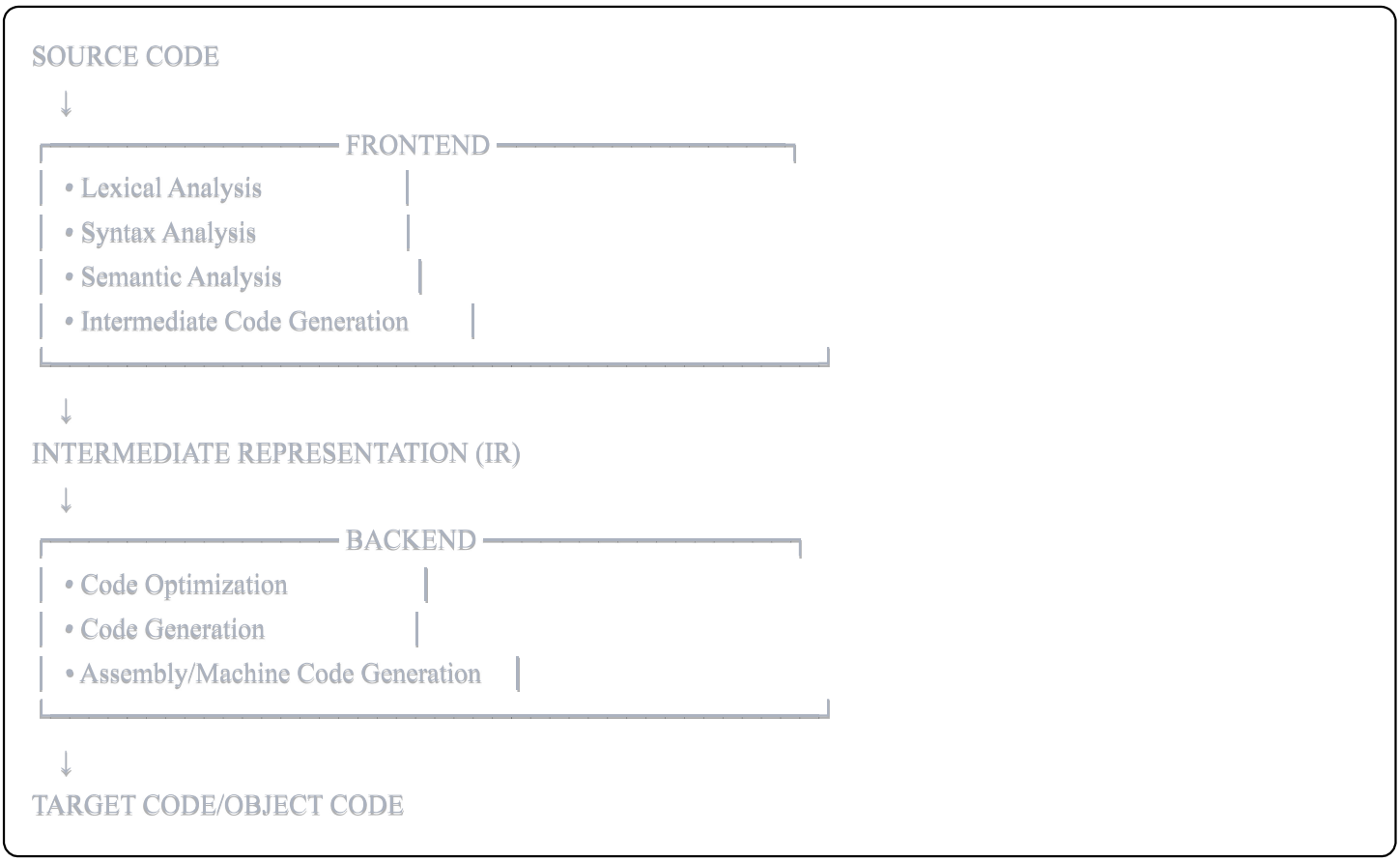
When to use:

- **Value:** Small data types (int, float, bool)
- **Reference:** Large objects, arrays, when you need to modify original
- **Name:** Rarely used in modern languages

Q10: For a compiler system, which modules consist of the frontend and backend?

Answer:

Compiler Architecture:



Frontend Components:

1. **Lexical Analysis (Scanner)**
   - Converts source code into tokens
   - Removes whitespace and comments
   - Input: `int x = 10;` → Output: `[int][identifier:x][=][number:10][;]`
2. **Syntax Analysis (Parser)**
   - Checks grammar structure
   - Creates parse tree/abstract syntax tree (AST)
   - Ensures code follows language rules
3. **Semantic Analysis**
   - Type checking
   - Scope checking
   - Symbol table management
   - Semantic error detection
4. **Intermediate Code Generation**
   - Converts to intermediate representation (like 3-address code)

- Machine-independent representation

Backend Components:

1. **Code Optimization**
   - Makes code faster/smaller
   - Removes redundant code
   - Loop optimization
   - Dead code elimination
2. **Code Generation**
   - Converts IR to target language
   - Assembly or machine code generation
   - Memory allocation
   - Register assignment
3. **Assembly/Linking**
   - Converts assembly to machine code
   - Links with libraries
   - Creates executable

Key Difference:

- **Frontend:** Language-specific, deals with source language
- **Backend:** Machine-specific, deals with target machine
- **Separation:** Allows same frontend for multiple backends, and same backend for multiple frontends