NPTEL Online Certification Courses Indian

Institute of Technology Kharagpur

Compiler Design

Assignment - Week 0

TYPE OF QUESTION:MCQ

Number ofquestions:11 Total mark: 11 X 1 = 11

Q1.
Task of a compiler is to

a) Translate one statement at a time and execute it
b) Translate the whole program to machine language
c) Translate one statement of the program at a time
d) None of the other options
ANS : b)

Explanation: A compiler is a program that translates the entire source code of a program written in a high-level programming language into machine code or an intermediate language in one go. The resulting code can then be executed by the machine.

Q2.
In a computer system, number of compilers for a particular programming language may Be

a) Two
b) Three
c) Four
d) Many

ANS: d)

Explanation:

There can be **many compilers** for a particular programming language in a computer system. This is because:

Platform-specific compilers: Different compilers are designed for different hardware architectures or operating systems. For example, compilers like GCC, Clang, and Microsoft C++ Compiler all support the C++ language but are optimized for different platforms.

Optimization and feature differences: Some compilers offer specific optimizations or additional features that others do not. Developers may choose a compiler based on their project's needs.

Open-source vs. proprietary: Both open-source compilers (like GCC or LLVM) and proprietary compilers (like Intel C++ Compiler or Oracle's Java Compiler) exist for many

languages.

Specialized use cases: Some compilers are tailored for education, embedded systems, or research purposes.

Thus, for a single programming language, there can be **many compilers** available.

Q3.
Natural language constructs are

a) Ambiguous
b) Unambiguous
c) May be Unambiguous or ambiguous
d) None of the other options

ANS: c)

Explanation:

Q4.
Suppose there is a compiler for C language that can generate code for Computer A. Which of the following statements is true

a) It can be used for Computer A only
b) It can be used for any computer
c) It can be used only for computers with similar processor and operating system
d) It can be used only for computers with similar processor, operating system and peripherals

ANS: c)

Explanation: A compiler generates machine code that is specific to a processor architecture (e.g., x86, ARM) and, in many cases, tailored to a specific operating system (e.g., Windows, Linux). This is because:

Processor-specific code: Machine code instructions are designed for specific CPU architectures. Code compiled for one processor (e.g., x86) will not run on another (e.g., ARM).

Operating system dependencies: Compilers often generate code that relies on system calls, libraries, and APIs provided by the operating system. Code designed for one OS may not work on another.

Peripherals: While peripherals (e.g., printers, external devices) may vary, they are typically abstracted through drivers and APIs, so their exact configuration is not directly relevant to the compiler.

Q5.
Which of the following data structures may be good if there are frequent search for data items followed by insertion and deletion?

a) Array
b) Link List
c) Tree
d) Hash Table

ANS: d)

Explanation: A Hash Table is efficient for frequent searches, insertions, and deletions because it provides O(1) average time complexity for all these operations. Data is quickly accessed, inserted, or deleted using a key and a hash function, making it highly efficient for dynamic data handling.

Q6.
Task of an interpreter is to

a) Translate one statement of the program at a time
b) Translate one statement at a time and execute it
c) Translate the whole program to machine language
d) None of the other options

ANS: b)

Explanation: An interpreter processes and executes a program one statement at a time. It directly translates the high-level code into machine-executable instructions without producing a separate machine code file.

Q7.
If an Infinite language is passed to Machine M, the subsidiary which gives a finite solution to the infinite input tape is _____

a) Compiler
b) Interpreter
c) Loader and linkers
d) None of the mentioned

ANS : a)

Explanation: A Compiler is used to give a finite solution to an infinite phenomenon. Example of an infinite phenomenon is Language C, etc.

Q8.
Languages of a automata is

a) If it is accepted by automata
b) If it halts
c) If automata touch final state in its life time
d) All language are language of automata

ANS: a)

Explanation: If a string accepted by automata it is called language of automata.

Q9.
Finite automata requires minimum _____ number of stacks.

a) 1
b) 0
c) 2
d) None of the mentioned

ANS: b)

Explanation: Finite automata doesn't require any stack operation .

Q10.
The basic limitation of finite automata is that

a) It can't remember arbitrary large amount of information.
b) It sometimes recognize grammar that are not regular.
c) It sometimes fails to recognize regular grammar.
d) All of the mentioned

ANS: a)

Explanation: Because there is no memory associated with automata.

Q11.

Which of the following languages can be recognized by finite automata?

a) Regular languages
b) Context-free languages
c) Context-sensitive languages
d) None of the mentioned

ANS: a)

Explanation: Finite automata are limited to recognizing regular languages, which are defined by regular expressions. They cannot handle languages requiring memory, such as context-free or context-sensitive languages.

END of Assignment

Compiler Design

Assignment - Week 1

TYPE OF QUESTION:MCQ

Number ofquestions:12 Total mark: 12 X 1 = 12

Q1.

ANS : d) None of the other options

Detailed Solutions:

- **a) Code generation**:
  - The symbol table is essential in the code generation phase. It holds information about variables, functions, and their memory addresses, which are needed to generate the final code.
- **b) Syntax Analysis**:
  - During syntax analysis (parsing), the symbol table is used to store and retrieve information about the program's symbols (e.g., variables, functions). It helps track the symbols' declarations and scopes.
- **c) Lexical Analysis**:
  - Even during lexical analysis (tokenization), the symbol table may be used to track identifiers and keywords, especially for checking whether a variable has been declared and to associate it with its type.
- **d) None of the other options**:
  - All three phases mentioned (code generation, syntax analysis, and lexical analysis) make use of the symbol table in some capacity.

Q2.

ANS: b) It is Second Phase Of Compiler after Lexical Analyzer.

Detailed Solutions:

1. **Lexical Analysis**:
   - First phase of the compiler.
   - Converts source code into tokens.

2. **Syntax Analysis** (Answer: **b**) :
    - Second phase.
    - Uses tokens from lexical analysis to check if they conform to the grammatical rules of the programming language.
    - Builds a **parse tree** (or syntax tree).
3. **Semantic Analysis**:
    - Third phase.
    - Checks for semantic errors (e.g., type checking) and annotates the syntax tree.
4. **Intermediate Code Generation**:
    - Converts the syntax tree into an intermediate representation.
5. **Code Optimization** (Optional):
    - Improves the intermediate code for better performance.
6. **Code Generation**:
    - Produces the target machine code.
7. **Code Linking and Loading**:
    - Links different modules and prepares the code for execution.

Q3.

ANS: A) parse tree

Detailed Solution: • **Parse Tree**:

- The syntax analysis phase (or parsing) constructs a hierarchical tree structure called a parse tree (or syntax tree).
- It represents the grammatical structure of the source code as per the rules of the language's grammar.

- **Other Options**:

    - **Keyword Tree**: This is not a valid term in the context of compilers.
    - **Binary Tree**: While some structures in compilers may use binary trees, the parse tree is not restricted to being binary.

Q4.

ANS: c) Semantic Analysis

Detailed Solution: **Semantic Analysis**:

- Verifies the **meaning** of the program.
- Checks language-specific rules such as:
    - Type compatibility.
    - Validity of operations (e.g., whether integer division is allowed).
    - Scope resolution.
    - Function calls with correct parameters.
- Disallowing integer division would be a rule enforced in this phase.

Q5.

ANS: c) The Symbol table does not ever perform the processing of the assembler derivative.

Deatiled Solution : This is false because processing assembler directives is a function of the assembler itself, not the symbol table. The symbol table's role is only to store and retrieve information about symbols.

Q6.

Answer: b) syntax error

Explanation: No compiler can ever check logical errors.

Q7.

ANS: a) Report multiple errors

Detailed Solution: **Error Recovery** in compilers is a mechanism used to continue processing after encountering an error, allowing the compiler to detect and report additional errors rather than terminating immediately. This approach helps developers identify and fix multiple errors in a single compilation run.

Q8.

ANS: a) Silicon Compilation
Detailed Solution:

- A process in hardware design where a high-level hardware description (often written in a Hardware Description Language, or HDL) is automatically translated into a lower-level representation suitable for physical fabrication, such as gate-level design or chip layout.
- This process is analogous to compiling a program in software development but applied to hardware design.

Q9.
ANS : B) loop body is repeated several times

Detailed Solution: This is true because the loop body executes multiple times during the program's execution. Optimizing loops can significantly improve the performance of a program since even small improvements in the loop body can lead to substantial time savings.

Q10.

Answer: c) CPU registers

Detailed Solution: • Registers are the fastest storage locations available in a computer because they are part of the CPU itself.
• Allocating temporary variables to registers minimizes memory access time and speeds up

Q11.

ANS: c) Retargeting code

Detailed Solution: • **Intermediate Code**:

- Intermediate code is an abstraction between the source code and the target machine code.
- It is platform-independent, allowing the compiler to generate target code for multiple architectures without reanalyzing or rewriting the source program.

- **Program Analysis**:

  - While intermediate code may assist in program analysis (like semantic checks), this is not its primary purpose.

- **Code Optimization**:

  - Intermediate code simplifies optimization, but the main focus is retargeting for multiple architectures.

- **Code Check**:

  - Intermediate code is not directly related to code checking.

Q12.

Answer: b
Explanation: This Produce the file “myfile.yy.c” which we can then compile with g++.

END of Assignment

Compiler Design

Assignment 2

No. of Question 13

Q1.

Ans: C) Constituent strings of a language

Detailed Solution: Regular expressions are used to define patterns that match strings in a language. They describe the set of strings (constituent strings) that belong to a specific language. For example, the regular expression `a*b` represents strings in the form of zero or more occurrences of `a` followed by a single `b`, such as `b`, `ab`, `aab`, etc.

Q2.

Ans : c)
Explanation:
Lexeme
Token category
Sum "Identifier"
= "Assignment operator"
3 "Integer literal"
+ "Addition operator"
2 "Integer literal"
; "End of statement"

Q3.

Ans: C)

Detailed Solution: In Fortran, the statement `DO 5 I = 1.25` is ambiguous without proper tokenization because `DO 5 I` could initially be interpreted as part of a **DO loop** construct. However, upon encountering the `.` (in `1.25`), the tokenizer realizes that this is not a DO loop but rather an assignment statement (`DO5I = 1.25`) where `DO5I` is an identifier.

Q4.

Ans : d)

Explanation: Different Lexical Classes or Tokens or Lexemes Identifiers, Constants, Keywords, Operators.

Q5.

Ans: d)

Deatiled Solution: Regular expressions are used to define patterns for **regular languages**. However, the language that accepts strings with **exactly one more `1` than `0`s** is not a regular language. This is because keeping track of the difference between the number of `1`s and `0`s requires a form of counting or memory, which regular expressions (and finite automata) cannot handle.

This type of language can only be recognized by more powerful computational models, such as a **pushdown automaton** (for context-free languages) or a **Turing machine**. Hence, it is **not possible** to write a regular expression for this condition.

Q6.

Ans : C)

Explanation: The regular expression has two 0′s surrounded by (0+1)* which means accepted strings must have at least 2 0′s.

1. Finite automata is an implementation of

a) Regular expression
b) Any grammar
c) Part of the regular expression
d) None of the other options

Ans: A)

Deatiled Solution: Finite automata (both deterministic and non-deterministic) are mathematical models used to recognize or implement **regular languages**, which are the types of languages described by regular expressions.

- Regular expressions provide a way to describe patterns in strings, while finite automata serve as the computational model that accepts or rejects strings based on those patterns.
- There is a direct correspondence between regular expressions and finite automata: for every regular expression, there exists an equivalent finite automaton, and vice versa.

Q7.

Ans: A)

Deatiled Solution: Finite automata (both deterministic and non-deterministic) are mathematical models used to recognize or implement **regular languages**, which are the types of languages described by regular expressions.

- Regular expressions provide a way to describe patterns in strings, while finite automata serve as the computational model that accepts or rejects strings based on those patterns.
- There is a direct correspondence between regular expressions and finite automata: for every regular expression, there exists an equivalent finite automaton, and vice versa.

Q8.

Ans: A)

Detailed Solution: In a **Nondeterministic Finite Automaton (NFA)**, it is possible to transition from one state to another **without consuming any input symbols**. These transitions are called **epsilon (ε) transitions**.

- **NFA**: Allows epsilon transitions, which means the automaton can move to a new state without reading any input.
- **DFA (Deterministic Finite Automaton)**: Does not allow epsilon transitions; every transition in a DFA must consume an input symbol.
- **Pushdown Automaton**: While it allows stack operations, epsilon transitions are not specifically related to finite state machines like NFAs.
- **All of the mentioned**: Incorrect because DFA does not support epsilon transitions.

Q9.

Ans: A)

Explanation: The ε-closure of a set of states, P, of an NFAis defined as the set of states reachable from any state in P following e-transitions.

Q10.

Ans: c)

- Detailed Solution: Both **Nondeterministic Finite Automata (NFA)** and **Deterministic Finite Automata (DFA)** are computationally equivalent, meaning they recognize the same class of languages: **regular languages**.

- **NFA**: Easier to design and can have multiple transitions for the same input or epsilon transitions. However, it is not deterministic.
- **DFA**: Deterministic in nature and has no epsilon transitions or multiple transitions for the same input. It is easier to implement in software or hardware.

While an NFA may appear more "flexible," any NFA can be converted into an equivalent DFA. The DFA might have exponentially more states in the worst case, but it recognizes the exact same language as the NFA.

Thus, NFAs and DFAs are **equally powerful** in terms of the languages they can recognize.

Q11.

Ans: A)

Explanation: The conversion of a non-deterministic automata into a deterministic one is a process we call subset construction or power set construction.

Q12.

Ans: C)

Explanation: Thompson Construction method is used to turn a regular expression in an NFA by fragmenting the given regular expression through the operations performed on the input alphabets.

Q13.

Ans: D)

Detailed Solution: n the scenario where a compiler automatically corrects errors like changing "fi" to "if," the error correction involves **transposing** the characters.

- **Transpose character**: This involves swapping two adjacent characters in a string to correct a mistake, such as changing "fi" to "if."

Compiler Design

Assignment- Week 3

TYPE OF QUESTION:MCQ

Number ofquestions:10 Total mark: 10 X 1 = 10

Q1.

Ans: d)

Detailed Solution: **Lex**: A lexical analyzer generator that was one of the first tools used to create scanners in Unix environments.

1. **Flex**: An enhanced and faster version of Lex.
2. **JFlex**: A lexical analyzer generator specifically designed for Java-based projects.

All of these tools are used for lexical analysis, which involves breaking input text into tokens as part of the compilation process.

Q2.

Ans: d)

Detailed Solution: In a Lex specification file, **"?"** is not a valid operator. Common operators in Lex include:

- ***: Matches **0 or more occurrences** of the preceding regular expression.
- **+**: Matches **1 or more occurrences** of the preceding regular expression.
- **?**: This is not a valid Lex operator and is typically associated with other regular expression engines (e.g., in some contexts, it means 0 or 1 occurrence).

Q3.

Ans: a)

Detailed Solution:

A DFA can potentially have more states than an NFA. When converting an NFA to a DFA (using the subset construction algorithm), the number of states in the DFA can be exponentially larger in the worst case. For example, an NFA with n states could result in a DFA with up to 2^n states.

Q4.
Ans: d)

Deatiled Solution:

Q5.

Ans: d)

Deatiled Solution:

A Lex specification file is divided into three sections, separated by %%:

1. **Definition Section**: Contains definitions for macros and declarations of variables.
2. **Rules Section**: Specifies patterns and corresponding actions.
3. **User Code Section**: Optional section containing additional C code to be included.

Each section is demarcated by %%. For example:

Definition Section

%%

Rules Section

%%

User Code Section

Q6.

Ans: a)

Detailed Solution:

ε-closure (epsilon-closure) of a state in an NFA is the set of all states that can be reached starting from that state using **only ε (epsilon) transitions**, including the state itself.

- An ε transition is a transition that consumes no input.
- The ε-closure of a state is computed by recursively exploring all states that can be reached through ε transitions from the given state.

Q7.
Answer: d)

Deatiled Solution: Let's analyze each option regarding the **ε-closure** of a subset S of states Q:

1. **a) Every element of S**:
   This is true because the ε-closure of S includes every element of S itself.
2. **b) For any q∈ε-closure, every element of δ(q,ε)\ is in ε-closure**:
   This is true because the ε-closure of a state includes all states reachable by ε-transitions.
3. **c) No other element is in ε(S)**:
   This is true because the ε-closure of S is precisely defined as the set of all states reachable by ε-transitions, and no other states are included.

Q8.
Ans: b)

Detailed Solution: When a Lex program is processed, the Lex tool generates a C source code file containing the lexical analyzer. By default, this file is named **lex.yy.c**.

- This file includes the code for the scanner based on the rules defined in the Lex specification file.

- **a) Lex.c**: Not a default output of Lex.
- **b) Lex.yy.c**: The correct output file generated by Lex.
- **c) Lex.l**: The input specification file for Lex (not the output).
- **d) Lex.yy.l**: Not a valid file name in the Lex context.

Q9.
Ans: b)

Detailed Solution: Regular languages can be represented using multiple equivalent formalisms. Let's examine the options:

1. **i) DFA**:
   DFAs can represent all regular languages, but there may be regular languages that are easier to describe using NFAs or regular expressions.
2. **ii) NFA**:
   NFAs can represent all regular languages. They are equivalent to DFAs in expressive power but are often more concise.

3. **iii) Regular Expressions**:
   Regular expressions can describe all regular languages. They provide a more compact and human-readable way to define patterns.

Q 10.
Ans: c)

Detailed Solution:

A Lex program is divided into **three sections**, separated by %%:

1. **Definition Section**:
   This section contains declarations, macros, and regular definitions. These are used to define constants, patterns, or reusable components.
2. **Rules Section**:
   Contains the core of the Lex program, where regular expressions are associated with actions (C code) to execute when the patterns are matched.
3. **User Code Section**:
   An optional section that includes any additional C code or functions that support the Lex program.

Definitions Section
%%
Rules Section
%%
User Code Section

Compiler Design

Assignment- Week 4

TYPE OF QUESTION:MCQ

Number ofquestions:11 Total mark: 11 X 1 = 11

1.

Ans: a)

- Solution: In formal language theory and automata, a **language** consists of words (strings) that are formed from a set of **terminals**.
- **Terminals** are the basic symbols from which strings (words) of a language are composed.
- **Non-terminals** are used in the production rules of a grammar but do not appear in the final words of the language.

Thus, the words of a language are made up **only of terminals**.

2.

Ans: a)
Solution:

 A grammar is **ambiguous** if a single string (sentence) derived from the grammar has **more than one distinct parse tree** or derivation.

 The given grammar:

$E \rightarrow E+E \mid E*E \mid id$

allows multiple ways to derive the same expression. For example, for the string id + id * i, we can have two different parse trees:

3.

1. **Parse tree considering leftmost derivation:**
   - $E \rightarrow E + E$
   - $E \rightarrow id$
   - $E \rightarrow E * E$
   - $E \rightarrow id$
   - $E \rightarrow id$
2. **Parse tree considering rightmost derivation:**
   - $E \rightarrow E * E$
   - $E \rightarrow E + E$
   - $E \rightarrow id$
   - $E \rightarrow id$
   - $E \rightarrow id$

- These two distinct parse trees show that the grammar does not enforce a **unique structure** for the expression, making it **ambiguous**.

How to remove ambiguity?

- Introduce operator precedence and associativity explicitly (e.g., * has higher precedence than +).
- Use different non-terminals for different levels of precedence.

Thus, **the given grammar is ambiguous.**

Ans: a)
Explanation: Deterministic CFGs are always unambiguous , and are an important subclass of unambiguous CFGs; there are non-deterministic unambiguous CFGs, however.

4. A language that admits only ambiguous grammar:

a) Inherent Ambiguous language
b) Inherent Unambiguous language
c) Context free language
d) Context Sensitive language

Answer: a
Explanation: A language is called inherently ambiguous if every possible grammar that generates it is ambiguous.

- This means that no unambiguous grammar can be written for such a language.
- Some context-free languages (CFLs) are inherently ambiguous, meaning that ambiguity cannot be removed by rewriting the grammar.

5.

Ans: b)

Solution:

A grammar contains **left recursion** if a non-terminal appears as the first symbol on the right-hand sic of its own production, leading to a recursive derivation.

Let's analyze the given production rules:

1. **A → Ba | Cb**
2. **B → CA**
3. **C → c | ε**

Now, checking for **left recursion**:

- Consider **B → CA**
  - If **C → ε**, then **B → A**
  - This means that **B indirectly leads to A**, causing left recursion in combination with **A → Ba**.
  - Since **A → Ba**, substituting **B → CA** would lead to a derivation involving **A** on the leftmost si again.

Since **A** depends on **B**, and **B** depends on **A**, this results in **left recursion**.

6. For the grammar rules {S → Aa | bB, A → c |ε}, FIRST(S) is

Ans: c)
Solution:

Explanation:

To determine FIRST(S) for the given grammar:

Given grammar rules:

1. S → Aa | bB
2. A → c | e (where e represents ε, i.e., an empty string)

Step 1: Calculate FIRST(A)

- A → c ⇒ FIRST(A) includes {c}
- A → e (ε) ⇒ A can derive ε
  Thus, FIRST(A) = {c, ε}

Step 2: Calculate FIRST(S)

- S → Aa
  - A can be c or ε
  - If A → c, then Aa starts with c
  - If A → ε, then Aa starts with a
  - Thus, from this production, FIRST(S) includes {c, a}
- S → bB
  - The first symbol is b, so FIRST(S) includes {b}

Final FIRST(S) set:
FIRST(S)=a,b,c

7. The grammar {E →E + T | T, T →T * F | F, F → id} is

Ans: b)

Solution:

A grammar is **unambiguous** if every string in the language has **a unique parse tree** (or derivation).

Given Grammar:

1. **E → E + T | T**
2. **T → T * F | F**
3. **F → id**

This is a **standard unambiguous expression grammar** because:

- **Operator precedence is correctly enforced**:
  - Multiplication (*) has **higher precedence** than addition (+).
- **Associativity is properly defined**:
  - Both + and * are **left-associative**.

Step 1: Checking for Ambiguity

Consider the input **"id + id * id"**.

Using the given grammar, the **only valid parse tree** follows standard precedence rules:

r
CopyEdit

```
    E
   / \
  E   +  T
  |     / \
  T     T   *  F
  |     |      |
  F     F      id
  |     |
 id     id
```

- The **multiplication (*) happens first**, followed by **addition (+)**, which is **correct operator precedence**.
- There is **only one unique parse tree**, meaning the grammar is **unambiguous**.

Step 2: Why is This Grammar Unambiguous?

- The grammar explicitly enforces **operator precedence** (* before +).
- The grammar ensures **left-associativity**.
- Every valid string **has only one possible parse tree**, proving that the grammar

is **unambiguous**.

8.

Ans: a)
Solution:

A top-down parser constructs the parse tree from the start symbol (root) to the leaves, applying production rules from left to right.

- In a leftmost derivation, at each step, the leftmost non-terminal is expanded first.
- This ensures that the derivation follows the leftmost rule application, making it compatible with top-down parsing techniques such as Recursive Descent Parsing and LL(1) Parsing.

Example:

Given the grammar:

1. $S \rightarrow A\ B$
2. $A \rightarrow a$
3. $B \rightarrow b$

For the input "ab", a leftmost derivation (LMD) would be:

1. $S \Rightarrow A\ B$ (Expand S)
2. $A\ B \Rightarrow a\ B$ (Expand A first, leftmost non-terminal)
3. $a\ B \Rightarrow a\ b$ (Expand B)

Here, the leftmost non-terminal is always expanded first, confirming leftmost derivation.

9.

Ans: a)

Solution:

Left recursion removal is mandatory for top-down parsing because top-down parsers (like Recursive Descent and LL(1) parsers) cannot handle left-recursive grammars.

What is Left Recursion?

A grammar has left recursion if a non-terminal refers to itself as the leftmost symbol in its production.

Example of left recursion:

A→Aα|βA \rightarrow Aα | βA→Aα|β

Here, the non-terminal A appears at the beginning of its own production (A → Aα), causing an infinite recursion in a top-down parser.

Why is Left Recursion a Problem?

- Top-down parsers expand the leftmost non-terminal first.
- If the grammar contains left recursion, the parser keeps calling the same rule indefinitely, leading to infinite recursion.

Solution: Removing Left Recursion

The left-recursive rule:

A→Aα|

Can be converted to an equivalent right-recursive form:

A→βA
A′→αA′|ε

This transformation eliminates left recursion, making the grammar suitable for top-down parsing.

10
.

Ans: b)

Solution:

A grammar is ambiguous if there exists at least one string that has more than one distinct parse tree or derivation.

Key Concept of Ambiguity:

- Ambiguity arises when a string can be derived in multiple ways, leading to different parse trees.
- This can happen when there is more than one leftmost derivation or more than one rightmost derivation.

11
.

Ans: a)
Explanation: Backtracking problem is solved by constructing a tree of choices called as the state-space tree. Its root represents an initial state before the search for a solution begins.

END of Assignment

Compiler Design

Assignment- Week 5

TYPE OF QUESTION:MCQ

Number ofquestions:10 Total mark: 10 X 1 = 10

1. Ans: a)

Solution: In **shift-reduce parsing**, a handle is the substring of the input that matches the right side of a production rule and whose reduction to the non-terminal on the left side of the rule represents one step in the reverse of a rightmost derivation.

- In the **stack-based implementation** of shift-reduce parsing, the handle will always be located at the **top of the stack** because the parser shifts symbols onto the stack and then reduces them when a handle is identified.
- Reduction replaces the handle at the top of the stack with the corresponding non-terminal.

2. Ans: c)

Solution: In **shift-reduce parsing**, two types of conflicts can arise:

1. **Shift-Reduce Conflict**:
   - This occurs when the parser cannot decide whether to shift the next input symbol onto the stack or reduce the handle on top of the stack.
   - Common in ambiguous grammars.
2. **Reduce-Reduce Conflict**:
   - This happens when the parser finds two or more possible reductions at the same point, and it cannot decide which production to use.
3. **Shift-Shift Conflict (Not Possible)**:
   - This is not possible because shifting means reading the next input symbol, and there is no ambiguity about which symbol to shift. The parser always shifts if shifting is allowed, with no competing shift operations.

3. Ans: c)

Explanation:

In **shift-reduce parsing**, a **viable prefix** is a string of symbols that can appear at the top of the stack and still be **a part of a valid rightmost derivation** of the input.

- The **stack** in shift-reduce parsing maintains a **viable prefix** of the input string.
- A **viable prefix** is a **prefix of a right-sentential form** that does not extend beyond a handle (the substring that can be reduced).
- This ensures that every step taken in shift-reduce parsing leads to a valid derivation.

Explanation of Other Options:

- **a) At the bottom we find the prefixes** ✗ (Incorrect, because the bottom of the stack holds the start symbol or intermediate derivations, not necessarily a prefix.)
- **b) None of the mentioned** ✗ (Incorrect, since option c is correct.)
- **d) Stack consists of viable prefixes** ✗ (Incorrect wording— the stack does not consist of multiple viable prefixes, it contains only the **current viable prefix**.)

4. Answer: b)

Solution: Solution:

Step 1: Understanding Follow(A)

The **Follow(A)** set consists of symbols that can appear **immediately after** A in some derivation.

Rules for computing FOLLOW:

1. If **A → αBβ**, then **everything in FIRST(β) (except ε) is in FOLLOW(B)**.
2. If **A → αB** or **A → αBβ** and **FIRST(β) contains ε**, then **everything in FOLLOW(A) is in FOLLOW(B)**.
3. The **start symbol** includes **$ (end of input marker)** in its FOLLOW set.

Step 2: Finding Follow(A)

- From the production:
  S → AB
  - Since **B** is after **A**, we check **FIRST(B)**.
  - **B → abbS | bS | ε → FIRST(B) = {a, b, ε}**.
  - Since **B** can be **ε**, **FOLLOW(S) is added to FOLLOW(A).**
  - **FOLLOW(S)** includes **$** (since S is the start symbol).
  - Thus, **FOLLOW(A) = FIRST(B) ∪FOLLOW(S) = {a, b} ∪{$} = {a, b, $}**.

5.

Answer: b)
Solution:

Shift-reduce parsers are a type of bottom-up parser. Here's why:

1. Bottom-up Parsing
    - Shift-reduce parsing begins with the input string and attempts to build the parse tree by reducing the input symbols to the start symbol of the grammar.
    - It works in reverse order of a rightmost derivation, meaning it traces the derivation backwards from leaves (input symbols) to the root (start symbol).
2. How Shift-Reduce Parsing Works:
    - Shift: Move the next input symbol onto the stack.
    - Reduce: If a handle (substring that matches the right side of a production rule) is on top of the stack, replace it with the corresponding non-terminal.
    - Accept: If the entire input is reduced to the start symbol, the parser accepts the input.
    - Error: If no valid move exists (shift or reduce), the parser reports an error.

Common Examples of Shift-Reduce Parsers:

- LR Parsers (LR(0), SLR(1), LALR(1), CLR(1))
- Operator-Precedence Parsers

6.

Answer: a)
Solution: In shift-reduce parsing, a handle is a substring that matches the right-hand side of a grammar production and whose reduction represents one step in the reverse of a rightmost derivation.

- During parsing, symbols are pushed onto the stack as part of the shift operation.
- When the parser identifies a handle at the top of the stack, it performs a reduce operation, replacing the handle with the corresponding non-terminal.

Thus, the handle is always located at the top of the stack when it's ready for reduction.

7.

Answer: c)

Solution: In **Operator Precedence Parsing**, special symbols like <• and •> are used to denote the precedence relationship between terminals.

- **<• (Left Precedence)** indicates that the terminal on the left has lower precedence than the one on the right.
- **•> (Right Precedence)** indicates that the terminal on the left has higher precedence

than the one on the right.

The **handle** is the substring located **between <• and •>**, as this represents the part of the input that corresponds to a reducible substring (i.e., it matches the right-hand side of a production and can be replaced by the corresponding non-terminal).

Consider the string and precedence relations:

id <• + •> id

- The handle is located **between <• and •>**, which corresponds to `id` in this case. This is the part that can be reduced based on a grammar rule.

Thus, the handle in operator precedence parsing is always **between <• and •>**.

8.

Ans: b)

Solution: In **Operator Precedence Parsing**, the symbol **≐** indicates **"equal precedence"** between two terminals, meaning they belong to the same handle and should be reduced together.

For the rule:
B → abbS

- **a ≐ b**: Since `a` and `b` are adjacent in the production `abbS`, it implies that `a` and `b` have **equal precedence** (≐).
- **b ≐ b**: The two consecutive `b` symbols in `abbS` indicate that **`b` has equal precedence with another `b`**, as they form part of the same handle.

Thus, the correct set of precedence relations is:
a ≐ b and b ≐ b.

9.

Ans: d)

Solution: Explanation:

An **operator-precedence parser** is a type of **bottom-up** parser that is also a **shift-reduce** parser and constructs derivations in **reverse order**.

Here's how each option is correct:

1. **Shift-Reduce Parser** ☑
   - Operator-precedence parsers use **shift** (to push operators and operands onto the stack) and **reduce** (to apply grammar rules).
   - This confirms that it follows a **shift-reduce parsing strategy**.
2. **Bottom-Up Parser** ☑
   - It starts from the **input (tokens) and constructs the parse tree from leaves to root**, making it a **bottom-up parser**.
   - Unlike top-down parsers, it does not predict rules but instead **reduces input based on precedence**.
3. **Constructs Derivation in Reverse** ☑
   - **Bottom-up parsers construct rightmost derivations in reverse**.
   - That means the derivation is built from **leaf nodes to the start symbol**.
   - Since operator-precedence parsers fall under bottom-up parsing, they **also construct derivations in reverse order**.

10.

Ans: d)

Solution: Bottom-up parsing is a method that constructs the parse tree starting from the input symbols and works its way up to the start symbol of the grammar. It involves the following key techniques:

1. Shift-Reduce Parsing:
   - This technique shifts symbols onto a stack and reduces them when a handle (a substring that matches the right-hand side of a production) is found.
   - Shift: Move the next input symbol onto the stack.
   - Reduce: Replace the handle on the stack with the corresponding non-terminal.
2. Handle Pruning:
   - Handle pruning is the process of identifying and reducing handles during bottom-up parsing.
   - It involves recognizing the substring that corresponds to the right-hand side of a grammar rule and replacing it with the left-hand side (a non-terminal). This builds the parse tree in reverse order.

END of Assignment

Compiler Design

Assignment- Week 6

TYPE OF QUESTION:MCQ

Number ofquestions:11 Total mark: 11 X 1 = 11

1.

Ans: c)
Solution:

Step 1: Compute Augmented Grammar

We first augment the grammar by adding a new start symbol:

S' → S
S → CC
C → cC | d

Step 2: Compute Closure of State 0

The LR(1) parsing algorithm starts with the closure of the initial state containing the augmented production:

[S' → .S, $]

Since S → CC, we expand:

[S' → .S, $]
[S → .CC, $]

Now, since C appears immediately after the dot (.), we need to expand C. Looking at the production C → cC | d, we add:
C → .cC, (lookahead: Follow(C))
C → .d, (lookahead: Follow(C))

To determine Follow(C):

- In S → CC, Follow(C) comes from Follow(S), which includes $.
- But we are only interested in the lookahead from S → CC, meaning First(C) = {c, d}.

Thus, for C → .cC and C → .d, the lookahead set is {c, d}.

Step 3: Identify the Correct Answer

From the closure computation, we see that the item C → .cC has lookahead {c, d}, meaning the correct item is:

C → .cC, c, d

2.

Ans: c)
Solution:

Step 1: Compute Augmented Grammar

We augment the grammar by adding a new start symbol:

S' → S
S → CC
C → cC | d

Step 2: Compute Closure of State 0

The LR(1) parser starts with:

[S' → .S, $]

Expanding S → CC gives:

[S' → .S, $]
[S → .CC, $]

Since C appears immediately after the dot (.), we expand C → cC | d and determine the **look ahead**.

The **Follow(C) in S → CC** is **First(C)**, which is {c, d}.

Thus, when expanding C, we get:

C → .cC, {c, d}
C → .d, {c, d}

Step 3: Identify the Correct Answer

From the closure computation, we see that **the item C → .d has lookahead {c, d}**, meaning the correct item is:

C → .d, c, d

3. Ans: c)
Solution:

4. Ans: b)

Solution: We are given the items in state I:

1. $A \rightarrow \alpha.$ (A is completed, meaning it can be reduced)
2. $B \rightarrow \delta.$ (B is also completed, meaning it can be reduced)

Additionally, we know:

- First(A) contains 'a'
- Follow(A) contains 'a'
- Follow(B) contains 'a'

Step 1: Understanding Reduce-Reduce Conflict

A reduce-reduce conflict occurs when two or more reductions are possible in the same parser state for the same input symbol.

Here, both $A \rightarrow \alpha$ and $B \rightarrow \delta$ are ready for reduction, and since Follow(A) and Follow(B) both contain 'a', the parser will not be able to decide which reduction to apply when 'a' appears in the input.

This results in a reduce-reduce conflict.

Step 2: Evaluating the Options

- Option a) "Shift-reduce conflict" ❌
  - A shift-reduce conflict occurs when a shift action and a reduce action are both possible, but here we only have two reductions.
- Option b) "Reduce-reduce conflict" ☑
  - Since both $A \rightarrow \alpha$ and $B \rightarrow \delta$ are completed and both have 'a' in their follow sets, this leads to a reduce-reduce conflict.
- Option c) "Both shift-reduce and reduce-reduce conflicts" ❌
  - There is no shift action in this scenario, so a shift-reduce conflict does not exist.
- Option d) "No conflicts" ❌
  - There is a conflict (reduce-reduce), so this is incorrect.

5. Which of the following statements is true regarding LR parsers?

a) SLR and Canonical LR have the same number of states.
b) LALR and Canonical LR have the same number of states.
c) SLR and LALR have the same number of states.
d) All three have the same number of states.
Ans: c)
Explanation:

- **SLR and LALR have the same number of states**, but LALR distinguishes states using lookaheads to reduce conflicts.
- **Canonical LR has more states** than both because it does not merge states with the same core items.

6. Ans: b)

Solution: Different LR parsing techniques handle reductions differently:

1. SLR (Simple LR):
    - Uses Follow(A) to determine reduce actions.
    - Requires the Follow set.
2. Canonical LR (LR(1)):
    - Uses lookahead symbols explicitly attached to each item.
    - Does NOT require the Follow set because the lookaheads are computed directly from the grammar.
3. LALR (Look-Ahead LR):
    - Similar to Canonical LR, but merges states.
    - Also uses lookahead symbols rather than the Follow set.
    - Does NOT require the Follow set.

7. Ans: a)
Solution:

Step 1: Constructing Items in State 0

State 0 starts with the closure of the augmented grammar:

$E' \rightarrow .E$
$E \rightarrow .aEbE$
$E \rightarrow .bEaE$

E → .

Since E → . (i.e., E → ε) is a completed item, it means that E can be reduced whenever the lookahead belongs to Follow(E).

Shift Possibility:

- The items E → .aEbE and E → .bEaE indicate that we can shift 'a' and 'b'.
- The First(aEbE) = {a} and First(bEaE) = {b}, so both 'a' and 'b' can be shifted.

Reduce Possibility:

- The ε-production (E → .) means that E can be reduced to ε whenever Follow(E) contains 'a' or 'b'.
- Follow(E) in SLR parsing is computed using the grammar:

Follow(E') = { $ }
Follow(E) = { a, b, $ } (Since E appears in recursive positions)

Since 'a' and 'b' are in Follow(E), the parser will try to reduce E → ε whenever it encounters 'a' or 'b'.

8. Ans: a)
   Solution:

9. Ans: b)

   Solution:

   Step 1: Constructing Items in State 0

   The initial items in state 0 include:

   S → .B
   S → .SabS
   B → .bB
   B → .

   Now, let's analyze the shift and reduce possibilities.

   Shift Possibility:

   - The item S → .SabS allows shifting 'a'.
   - The item B → .bB allows shifting 'b'.

   Reduce Possibility:

   - The item B → . (i.e., B → ε) means that B can be reduced whenever the lookahead belongs to Follow(B).
   - Follow(B) is computed using:
     - S → B, so Follow(B) contains Follow(S).

- Follow(S) = { $, a, b } (since S appears at the start and within SabS).

Thus, Follow(B) = { a, b, $ }.

Step 2: Identifying the Conflict

For 'b':

- The parser can shift using B → .bB.
- The parser does not reduce immediately because B → bB takes precedence.

For 'a':

- The parser can reduce using B → ε (since 'a' is in Follow(B)).
- The parser can shift using S → .SabS.

Since both shift (S → .SabS) and reduce (B → ε) actions are possible on 'a', a shift-reduce conflict occurs for 'a'.

10. Which of the following parser types is the most powerful in terms of recognizing a broader class of grammars?
a) LL(1)
b) SLR(1)
c) LALR(1)
d) LR(1)
Ans: d)

Explanation:

- **LL(1)** is the weakest among these as it only looks one symbol ahead and is top-down.
- **SLR(1)** is more powerful than LL(1) but can still fail for some grammars.
- **LALR(1)** is an improvement over SLR(1) but still merges states, making it weaker than LR(1).

11. Ans: a)

END of Assignment

NPTEL Online Certification Courses Indian

Institute of Technology Kharagpur

Compiler Design

Assignment- Week 7

TYPE OF QUESTION:MCQ

Number ofquestions:12 Total mark: 12 X 1 = 12

1.

Ans: c)

Explanation:

YACC (Yet Another Compiler Compiler) is a tool used in compiler construction to generate parsers. It is designed to take a context-free grammar (CFG) as input and generate a parser for that grammar.

Why is it called "Yet Another Compiler Compiler"?

- The name follows the tradition of humorous naming in computer science, where "Yet Another" was commonly used to describe new tools (e.g., YAML: Yet Another Markup Language).
- YACC generates parsers, which are a part of compiler construction, hence "Compiler Compiler."

2.

Ans: b)

Explanation:

- YACC (Yet Another Compiler Compiler) generates a C source file (y.tab.c) that contains the parser code.
- Additionally, YACC produces a header file (y.tab.h), which defines token identifiers for use by Lex (Lexical Analyzer).
- Lex uses y.tab.h to ensure that the token values assigned in YACC are consistent in the lexical analyzer.

File Descriptions:

- y.tab.c → Contains the parser's implementation in C.
- y.tab.h → Contains token definitions, used by Lex.
- y.parse.c & y.parse.h → These are not standard YACC output files.

3.

Ans: c)

Explanation**:**

A YACC (Yet Another Compiler Compiler) specification file is divided into three sections, separated by %% mar

kers:

1. Declarations Section (Definitions)
    - Specifies token definitions, data types, and other settings.
    - Includes:
        - %token declarations for terminal symbols.
        - %start to define the starting non-terminal.
        - %union for defining value types (when using yylval).

```
%{
#include <stdio.h>
%}
%token NUMBER
%left '+' '-'
```

Rules Section (Grammar Rules)

- Defines the context-free grammar (CFG) rules.
- Specifies actions in C code that execute when a rule is reduced.
- Example

```
%%
expr : expr '+' expr { $$ = $1 + $3; }
   | expr '-' expr { $$ = $1 - $3; }
   | NUMBER      { $$ = $1; }
   ;
%%
```

User Subroutines Section (C Code)

- Contains additional C functions, such as the yyerror() function and the main() function.
- Example:

```
int main() {
  yyparse();
  return 0;
}

void yyerror(char *s) {
  fprintf(stderr, "Error: %s\n", s);
```

}

4.

Ans: c)
Explanation:

In **YACC's rules section**, $$ refers to the **value of the non-terminal on the left-hand side (LHS) of a production rule**.

*How **$$** Works in YACC?*

- YACC uses **semantic actions** inside { } to associate C code with grammar rules.
- Each grammar rule's **LHS non-terminal receives a value** based on the right-hand side (RHS) expressions.
- $$ is used **to assign a value to the LHS non-terminal**.

Example:

```
expr : expr '+' expr { $$ = $1 + $3; }
   | expr '-' expr { $$ = $1 - $3; }
   | NUMBER       { $$ = $1; }
   ;
```

$$ represents the value of expr (LHS non-terminal).
$1, $2, $3 refer to RHS elements (expr, '+', expr respectively).
The action { $$ = $1 + $3; } means the value of expr (LHS) is set to the sum of the first and third symbols.

5.

Ans: a)
Explanation:

An annotated parse tree is a parse tree where each node is associated with attributes that store information relevant to semantic analysis. These attributes help in syntax-directed translation (SDT) and are used in syntax-directed definitions (SDD).

Key Features of an Annotated Parse Tree:

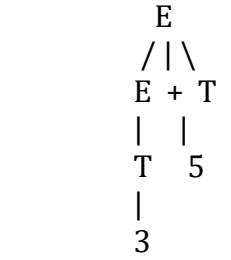
1. Each node is labeled with its corresponding grammar symbol.
2. Attributes are attached to nodes to store semantic information.
3. Computation of attributes follows semantic rules defined in the grammar.
4. It helps in intermediate code generation, type checking, and other semantic tasks.

Example:

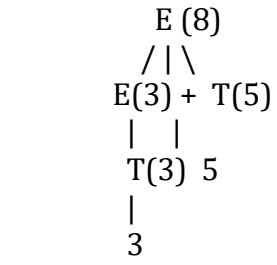
Consider the grammar:

E → E1 + T { E.val = E1.val + T.val }
T → NUMBER { T.val = NUMBER.lexval }

A parse tree for 3 + 5 would be:



If attributes are added to store values:



This annotated parse tree shows attribute values (val) computed at each step.

6.

Ans: c)

Explanation:

In **YACC's rule section**, **positional notation** ($n) is used to refer to elements on the **right-hand side (RHS) of a production rule**.

For the production rule:

A → B b C

- $1 refers to B (the first RHS symbol).
- $2 refers to b (the second RHS symbol).
- $3 refers to C (the third RHS symbol).

Thus, the correct reference for C is $3.

Example Usage in YACC:
A : B b C { $$ = $1 + $3; }
Here, $$ refers to A (LHS non-terminal).
$1 refers to B.
$2 refers to b (terminal).
$3 refers to C.

7. Syntax directed translation helps in

   a) Creating parse tree
   b) Check syntactical correctness of input
   c) Check if the input has foreign symbols
   d) None of the othrt options

Ans: a)

Explanation:

Syntax-Directed Translation (SDT) is a method used in compilers where semantic rules are associated with grammar rules to perform syntax analysis and translation.

SDT helps in:

- Building a parse tree (or annotated parse tree).
- Performing semantic analysis, such as type checking.
- Generating intermediate code for further compilation stages.

8.

Ans: c)
Explanation:

When **YACC (Yet Another Compiler Compiler)** processes a grammar file (e.g., parser.y), it generates the following files:

1. **y.tab.c**
   - This contains the **C source code for the parser**.
   - It includes the parsing function yyparse(), which is used in syntax analysis.
2. **y.tab.h**
   - This is a **header file containing token definitions**.
   - It defines token names and their corresponding numerical values.
   - It is used by **Lex (Lexical Analyzer)** to ensure consistency between **Lex and YACC**.

9.

Ans: d)

Explanation:

- yylval is a global variable used by **Lex and YACC** to store the **attribute value** of a token.
- It is used in **semantic analysis** where tokens carry additional information (like numerical values, variable names, etc.).
- When a token is identified by the **Lexical Analyzer (Lex)**, its associated value (attribute) is stored in yylval and passed to **YACC**.

Why Other Options Are Incorrect?

- a) yytext
  - yytext stores the actual string representation of the token, but not its attribute value.
  - It is useful in Lex but does not return token attributes.
- b) yylen
  - yylen is not a standard YACC or Lex variable.
  - There is no such variable that handles token attributes.
- c) yyval
  - yyval is used in YACC for handling semantic values within grammar rules, but it does not store token attributes.

10. YYSTYPE defines the type for

a) Stack
b) Token
c) Input
d) Queue

Ans: a)

Explanation:

- YYSTYPE is a data type definition used in YACC (Yet Another Compiler Compiler) to define the type of values stored in the parser's stack.
- Since YACC supports semantic values (like integers, floats, or complex structures), YYSTYPE allows users to define the type of attributes that are stored in the parsing stack.

How YYSTYPE Works?

- By default, YYSTYPE is defined as int, but it can be customized using a union to store multiple data types.

- Example

```
%union {
   int ival;
   float fval;
   char *sval;
}
%token <ival> NUMBER
%token <sval> IDENTIFIER
```

- Here, YYSTYPE can store integers, floats, and strings, depending on the token type.

Why Other Options Are Incorrect?

- b) Token ✗
  - YYSTYPE does not define a token, it defines the data type associated with tokens.
- c) Input ✗
  - YYSTYPE is not related to the input, which is handled by yytext and yylval.
- d) Queue ✗
  - YYSTYPE is used in the parsing stack, not in a queue.

Q11.

Which of the following statements about YACC is true?

a) YACC generates a lexical analyzer
b) YACC is used for syntax analysis
c) YACC does not support semantic actions
d) YACC does not require grammar rules

Ans: (b)

Explanation:

YACC (Yet Another Compiler Compiler) is a parser generator used for syntax analysis in compilers. It takes a context-free grammar (CFG) and generates a C program that performs parsing based on the specified grammar. Lex/Flex is used for lexical analysis, while YACC is used for syntax analysis.

Q12.

Which of the following correctly describes the role of *yyparse()* in YACC?

a) It performs lexical analysis
b) It calls the scanner (Lex) and parses tokens
c) It generates the final executable code
d) It stores token attributes

Ans: (b)

Explanation:

The function ***yyparse()*** is the main parsing function in YACC, automatically generated from the grammar rules. It retrieves tokens from Lex (using ***yylex()***) and applies grammar rules to build a syntax tree or process the input. It does not perform lexical analysis itself (which is done by Lex/Flex).

END of Assignment

Compiler Design

Assignment- Week 8

TYPE OF QUESTION:MCQ

Number ofquestions:13 Total mark: 13 X 1 = 13

1.

Ans: b)

Explanation:

Type checking is a part of **semantic analysis** in a compiler, where it ensures that operations are performed on compatible data types.

- **Lexical Analysis (Option a)** → Handles tokenization (e.g., recognizing keywords, identifiers, operators). It does **not** check types.
- **Syntax Analysis (Option c)** → Ensures correct structure based on grammar rules (e.g., matching parentheses, valid statement formation), but **not type correctness**.
- **Semantic Analysis (Option b)** → Checks **meaning**, including **type checking**, to ensure valid operations.

Example of Type Checking Error (Semantic Error):

```
int x = "hello";  // Error: assigning a string to an integer
```

2. Ans: c)

Explanation:

The **type expression** for an argument list follows the order in which the arguments are passed to the function. Given that:

- The **first argument** is an **Integer**
- The **second argument** is a **Real number**
- The **third argument** is an **Integer**

The correct notation for the type expression is: Integer X Real X Integer

3. Ans: c)

Explanation:

Array bound checking ensures that an array index is within valid limits to prevent out-of-bounds errors. This can be done in two ways:

1. Static Array Bound Checking (Compile-time)
   - If the array size and indices are known at compile time, some compilers can

detect out-of-bounds errors.
- Example (C/C++ with static analysis)

```
int arr[5];
arr[10] = 20;  // Compiler may warn about out-of-bounds access
```

- • Languages like Ada, Rust, and some static analysis tools in C/C++ support static bound checking.

- Dynamic Array Bound Checking (Runtime)

- When arrays are allocated dynamically or indices depend on user input, bound checking happens at runtime.
- Example (Python – automatic bound checking)

```
arr = [1, 2, 3]
print(arr[5])  # Raises Index Error at runtime
```

Languages like **Python, Java, and C++ (with** `std::vector::at()` **method)** perform dynamic bound checking.

4. Ans: a)

Explanation:

Type equivalence is the process of determining whether two **type expressions** represent the same type. This is an important part of **type checking** in programming languages.

- **Structural Equivalence:** Two types are considered equivalent if they have the same structure.
- **Name Equivalence:** Two types are equivalent only if they have the same declared name.

Example of Type Equivalence (Structural Equivalence in C-like languages)

```
typedef struct {

   int x;

   float y;

} A;

typedef struct {

   int x;

   float y;

} B;

A var1;
```

B var2;

// Structural equivalence: var1 and var2 have the same structure, but name equivalence would say they are different.

Since **type equivalence checks if two type expressions are the same**.

5. Ans: c)
Explanation:

In programming languages, a **statement** typically performs an action but does not return a value. Therefore, its type is usually **void**. However, if the statement contains an invalid operation, it results in a **type error**.

Cases:

1. **Valid Statement → Type is Void**
    - Example (C++/Java)

```
int x = 10;  // This is a valid statement, type is void
```

Invalid Statement → Type Error

- Example (Python)

```
x = "hello" + 5  # Type Error: can only concatenate str (not "int") to str
```

The operation is **not valid**, so a **type error** occurs.

6. Ans: a)
Explanation:

Type checking ensures that operations are performed on compatible data types. The type checking done by the compiler is called static type checking because it happens at compile time before the program runs.

Key Points:

- Static Type Checking (Done by Compiler)
    - Performed at compile time.
    - Helps catch type errors before execution.
    - Used in statically typed languages like C, C++, Java, Rust, Go.
    - Example (C++):

```
int x = "hello";  // Compilation error: invalid conversion
```

Dynamic Type Checking (Done at Runtime)

- Performed at runtime by the interpreter.
- Used in dynamically typed languages like Python, JavaScript.

- Example (Python):

```
x = "hello" + 5  # Raises TypeError at runtime
```

7. Ans: c)

Explanation:

A **weakly typed language** is one where type rules are more flexible, allowing implicit type conversions (**type coercion**) and catching some type errors only at runtime.

1. **Less Constraints on the Programmer**
   - Weakly typed languages allow **implicit conversions** between data types, making coding easier but potentially leading to unexpected behaviors.
   - Example (JavaScript – automatic type conversion)

```
let x = "10" + 5;  // "105" (string + number → string)
```

Some Type Errors are Caught Dynamically

- In weakly typed languages, type errors may not be detected at **compile time** and are instead caught **at runtime**.
- Example (Python – runtime type checking):

```
x = "hello" + 5  # Raises TypeError at runtime
```

8. Ans: c)

9. Ans: b)
Explanation:

Type casting is the process of converting a value from one data type to another. It can be explicit (manual) or implicit (automatic, called type coercion).

- Type Coercion refers to the automatic conversion of a value from one data type to another by the language itself.
  - Example (JavaScript - implicit coercion)

```
let x = "5" * 2;  // x = 10 (string "5" is coerced to a number)
```

- The language automatically converts `"5"` (string) to `5` (integer).

- **Explicit Type Casting** is a **manual conversion** where the programmer enforces type conversion.

- Example (C++ - explicit casting):

```
double num = 5.7;

int x = (int) num;  // Explicit casting: x = 5
```

10. Ans: d)

Explanation:

The given type expression:

$$(\text{Integer} \times \text{Real}) \to (\text{Integer} \to \text{Real})$$

can be interpreted step by step:

1. **Left Side:** $(\text{Integer} \times \text{Real})$
   - This means the function **takes two arguments**:
     - First argument: **Integer**
     - Second argument: **Real**
2. **Right Side:** $(\text{Integer} \to \text{Real})$
   - The function **returns another function** that:
     - Takes an **Integer**
     - Returns a **Real**

Consider a function **F** with the given type:

F: (Integer, Real) → (Integer → Real)

F takes two arguments: an Integer and a Real.F returns another function (G), where G:

- Takes an Integer
- Returns a Real

Thus, this matches option d:

11.

Type inference in programming languages refers to:

a) Explicitly specifying types for all variables
b) Automatically deducing the type of an expression at compile-time
c) Checking types dynamically during runtime
d) Ignoring type checking in a program

Ans: (b)
Explanation: Type inference is the process by which a compiler automatically determines the type of an expression without explicit type annotations. Many modern languages (e.g., Haskell, Scala, and TypeScript) use type inference to make coding more concise while still maintaining strong typing.

12.
Which of the following statements about type conversion is TRUE?

a) Implicit type conversion always results in data loss
b) Explicit type conversion is also called type coercion
c) Implicit type conversion is performed automatically by the compiler
d) Both (b) and (c)

Ans: (c)
Explanation: **Implicit type conversion**, also known as type promotion, is automatically handled by the compiler when a smaller data type is converted to a larger data type (e.g., int to float). **Explicit type conversion** (also called type casting, not coercion) requires manual intervention by the programmer.

13.
Dynamic type checking is necessary in languages that:

a) Perform all type checking at compile-time
b) Allow variables to change their type during execution
c) Do not support type inference
d) Have a very strict static type system

Ans: (b)
Explanation: **Dynamic type checking** is used in languages where variables can hold values of different types at runtime (e.g., Python, JavaScript). It ensures type safety during execution, unlike static type checking, which happens at compile-time.

END of Assignment

Compiler Design

Assignment- Week 9

TYPE OF QUESTION:MCQ

Number ofquestions:13 Total mark: 13 X 1 = 13

1. Self-organizing list-based symbol tables improve performance primarily due to:

a) Locality of reference in the input program
b) Locality of reference in the compiler's symbol access patterns
c) Both (a) and (b)
d) None of the above

Ans: (c)
Explanation:

A self-organizing list (SOL)-based symbol table dynamically adjusts the order of symbols based on access frequency, moving frequently accessed symbols to the front. This helps reduce lookup time and improves efficiency. The performance improvement comes from locality of reference, which applies in two key ways:

1. Locality of Reference in the Input Program:
   - Programs tend to use certain variables, functions, and identifiers repeatedly within short time spans (temporal locality).
   - Symbols declared together are often accessed together (spatial locality).
   - A self-organizing list benefits from this pattern by keeping frequently accessed symbols at the front, making lookups faster.
2. Locality of Reference in the Compiler's Symbol Access Patterns:
   - The compiler itself follows structured phases like lexical analysis, parsing, and semantic analysis.
   - During these phases, specific symbols (e.g., keywords, operators, variable names) are repeatedly accessed, forming a predictable access pattern.
   - A self-organizing list optimizes lookup time by adapting to these patterns.

2.

Ans: d)
Explanation:

A symbol table is a crucial data structure used by a compiler to store information about identifiers (variables, functions, etc.) encountered in the source code. The most frequent operation performed on a symbol table is lookup, for the following reasons:

1. During Compilation, Symbols Need to Be Accessed Frequently:
   - The compiler frequently checks whether an identifier has been declared before using it.

- Every occurrence of a variable, function, or keyword requires a lookup to retrieve associated information (e.g., type, scope, memory location).

2. Other Operations Occur Less Frequently:
   - Insert happens when a new identifier is encountered (e.g., a variable declaration). However, this happens once per identifier, while lookup happens multiple times.
   - Modify occurs when attributes of an identifier change (e.g., type updates in some languages), but it is less common than lookup.
   - Delete is rare, as symbol tables typically persist throughout a compilation phase or even until the end of the compilation process.

3.

Ans: b)
Explanation:

The main motivation behind using a self-organizing list (SOL) for a symbol table is to take advantage of program locality, which improves lookup efficiency. This is because:

1. Program Locality (Temporal & Spatial Locality):
   - Temporal Locality: Programs tend to access the same symbols repeatedly within a short period (e.g., looping over the same variable multiple times).
   - Spatial Locality: Symbols declared close to each other are often accessed together (e.g., function parameters or struct members).
   - A self-organizing list moves frequently accessed symbols to the front, reducing the average lookup time.
2. Why Not the Other Options?
   - (a) Ease of Implementation: While self-organizing lists are relatively simple to implement compared to other data structures (e.g., balanced trees, hash tables), ease of implementation is not the main motivation—efficiency is.
   - (c) Insertion of Symbols: The insertion operation is not a bottleneck in symbol tables, as lookups dominate. Optimizing insertion is not the primary reason for using SOL.
   - (d) None of the other options: This is incorrect because program locality is a well-established reason for using SOL.

4.

Ans: c)
Explanation:

The symbol table is a critical data structure in a compiler, used for storing and retrieving information about identifiers efficiently. To minimize access time, the best choice is a

hash table, because:

1. Constant Time Average Lookup (O(1)):
    - A hash table allows near constant-time lookup on average, which is much faster than other structures like trees or lists.
    - This is crucial since lookup is the most frequent operation in a symbol table.
2. Efficient Insertion & Deletion:
    - Hash tables provide fast insertion and deletion operations, typically O(1) on average.
    - Other structures like trees require O(log n) time for insertions and deletions.

5.

Ans: c)
Explanation:

An activation record (also called an activation frame) is a data structure used in stack-based function calls to manage information related to a procedure (function). It typically stores:

1. Parameters (Function Arguments)
    - These are values passed to the function when it is called.
    - Stored in the activation record so the function can access them.
2. Local Variables
    - Variables that are declared inside the function and exist only during the function's execution.
    - Stored in the activation record because they need memory allocation during execution.
3. Other Components (Not Listed in the Options)
    - Return Address – To store the address to which the function should return after execution.
    - Saved Registers – To preserve values of registers before calling another function.
    - Control Link (Dynamic Link) – A pointer to the caller's activation record (used in nested function calls).

6.

And: d)
Explanation:

The symbol table is a crucial data structure in a compiler, used across multiple phases to store and retrieve information about identifiers (variables, functions, types, etc.). Let's analyze how each phase uses the symbol table:

1. Semantic Analysis (Uses Symbol Table
    - Ensures that identifiers (variables, functions, types) are used correctly.
    - Uses the symbol table to check if a variable is declared before use and if

function calls have the correct parameters.

2. Code Generation (Uses Symbol Table
    - Converts high-level representations to low-level machine code.
    - Uses the symbol table to fetch memory locations, offsets, and variable information.
3. Code Optimization (Uses Symbol Table
    - Improves the efficiency of the generated code (e.g., eliminating redundant calculations).
    - Uses the symbol table for constant propagation, inline expansion, and register allocation.

7.

Ans: c)
Explanation:

When two types have the same name, they can be:

1. Name Equivalent:
    - In name equivalence, two types are considered the same if they have the same name, regardless of their structure.

Typedef int myInt;
Typedef int anotherInt;

- Here, myInt and anotherInt may not be name equivalent unless explicitly defined as the same type.

Structurally Equivalent:

- In structural equivalence, two types are considered equivalent if they have the same structure, even if their names are different.
- Example

struct A { int x; };
struct B { int x; };

- Here, A and B are structurally equivalent but not name equivalent because they have different names.

Both Name and Structurally Equivalent:

- If two types have the same name and the same structure, they are both name and structurally equivalent.

8. Which type of compiler typically benefits from using a separate symbol table for each scope?

a) Single-pass compilers
b) Multi-pass compilers
c) Both single and multi-pass compilers

d) None of the given options

Ans: a)

9.

Ans: c)
Explanation:

10.

Ans: d)
Explanation:

A symbol table is a data structure used by a compiler to store information about identifiers (variables, functions, constants, etc.). The following information is typically stored in a symbol table:

1. (A) Name
    - The identifier name (e.g., variable names, function names, class names, etc.).
2. (B) Location
    - The memory address or relative location of the identifier in storage (stack, heap, or static memory).
3. (C) Scope
    - The visibility and lifetime of the identifier (e.g., global, local, block scope).

Since all the given options are commonly stored in a symbol table, the correct answer is:

(D) None of the other options.

11.

What is the primary purpose of a symbol table in a compiler?

a) To store machine code instructions
b) To optimize runtime performance
c) To store and retrieve identifier-related information efficiently
d) To generate intermediate code

Ans: (c)
Explanation:

A symbol table is a key component of a compiler that maintains information about identifiers (variables, functions, types, constants) encountered during compilation. It is mainly used for:

- Efficient lookup of identifiers (names, types, scope, location, etc.).
- Semantic analysis to check declarations and type consistency.
- Code generation, where it provides memory locations of variables and function addresses.

12.

Which of the following data structures is best suited for managing scope information in a symbol table?

a) Stack
b) Queue
c) Linked List
d) Heap

Ans : (a)

Explanation: In compilers, scope information is usually managed using a stack-based symbol table approach.

- When a new scope (e.g., function, block) is encountered, a new symbol table is pushed onto the stack.
- When the scope ends, the table is popped, ensuring that variables from an inner scope do not conflict with those in outer scopes.
- This method efficiently handles nested scopes, making lookup and removal of variables efficient.

13.

Which optimization technique benefits the most from information stored in a symbol table?

a) Constant propagation
b) Loop unrolling
c) Dead code elimination
d) Instruction pipelining

Ans: (a)
Explanation:

Constant propagation optimization replaces occurrences of a variable with its known constant value to improve efficiency.

- The symbol table stores constant values assigned to variables, which helps the compiler identify opportunities for constant propagation.
- Example

```
int x = 10;
int y = x + 5;  // Can be optimized to y = 15;
```

- The symbol table helps track that x = 10, allowing the compiler to replace y = x + 5 with y = 15.
- Other optimizations (loop unrolling, dead code elimination, instruction pipelining) may use symbol table data, but constant propagation relies on it the most.

END of Assignment

Compiler Design

Assignment- Week 10

TYPE OF QUESTION:MCQ

Number ofquestions:13 Total mark: 13 X 1 = 13

1.

Ans: c)

Explanation:

An activation record (also called an activation frame) is a data structure used in a program's runtime stack to store information about function calls, including local variables, parameters, return addresses, and saved registers.

If the activation record is static, this means that each function has a fixed memory location allocated at compile time rather than dynamically allocating space on the stack during execution. This has significant implications:

- (A) Passing parameters → Possible
  - Parameters can be passed using registers or a statically allocated memory area.
- (B) Creating local variables → Possible
  - Local variables can be allocated in fixed memory locations determined at compile time.
- **(C) Supporting recursion → NOT possible
  - Recursion requires multiple instances of an activation record for the same function, each with different values for local variables and return addresses. If activation records are static, a new instance cannot be created dynamically for each recursive call, making recursion impossible.
- (D) None of the other options → Incorrect
  - Since recursion is not possible, option (D) is false.

2.

Ans: b)

Explanation:

The control link (also known as the dynamic link) in an activation record is a pointer to the activation record of the calling (parent) function. It helps maintain the correct

function call sequence and is crucial for returning to the correct activation record after a function call completes.

Here's how the options relate:

- (A) Current activation record → Incorrect
  - The control link does not point to the current activation record itself. Instead, it links to the previous (parent) activation record.
- (B) Parent activation record → Correct
  - The control link stores the address of the activation record of the calling function (parent), allowing the program to return correctly when the called function completes execution.
- (C) Child activation record → Incorrect
  - A function may call multiple child functions, but the control link does not track child activation records. Instead, it always links to the caller.
- (D) None of the other options → Incorrect
  - Since (B) is correct, (D) is false.

3.

Ans: b)
Explanation:

Intermediate Code Generation (ICG) is an optional phase in a compiler. It is used to create an intermediate representation (IR) between the source code and the final machine code. The purpose of ICG is to improve portability and ease optimization.

However, a compiler can be designed without an intermediate representation by directly translating source code into machine code (such as in Just-In-Time (JIT) compilers or interpreters). Therefore, ICG is not mandatory but is commonly used in multi-stage compilation.

Here's an analysis of the options:

- (A) Must → Incorrect
  - While intermediate code is beneficial, some compilers (e.g., direct interpreters) do not generate it.
- (B) Optional → Correct
  - Many compilers use an intermediate representation, but it is not strictly required. Some compilers translate source code directly into machine code.
- (C) Depends on language → Incorrect
  - While the complexity of intermediate code generation can depend on the language, the use of an intermediate representation is a design choice rather than a necessity dictated by the language.
- (D) None of the other options → Incorrect

4.

Ans: a)
Explanation:

P-code (Pseudo-code) is an intermediate code representation used in stack-based virtual machines, such as the Pascal P-machine. It is designed to be executed on an abstract stack-based architecture, meaning that operands are pushed onto and popped from a stack rather than using registers or memory addresses directly.

Here's how the options relate:

- (A) Stack-based machine → Correct
  - P-code is designed for execution in stack-based virtual machines, where operations involve pushing and popping values from a stack.
- (B) Accumulator-based machine → Incorrect
  - An accumulator-based machine primarily uses a single register (the accumulator) for arithmetic operations, whereas P-code relies on a stack for computation.
- (C) Two operand addresses → Incorrect
  - In a two-address machine, instructions explicitly reference two operands in memory or registers. P-code primarily works with stack operations rather than direct memory addresses.
- (D) None of the other options → Incorrect

5.

Ans: b)

Explanation:

In an **activation record (stack frame)**, local variables are typically stored **below the frame pointer (FP)** in memory. The **frame pointer (FP)** is a fixed reference point, and local variables are accessed using **negative offsets** relative to FP.

How memory is arranged in a stack frame:

Higher Memory (↑)	Component	Offset from FP
Return Address	+ve Offset	
Saved Registers	+ve Offset	
Parameters	+ve Offset	
Frame Pointer (FP)	0	
Local Variables	-ve Offset	
Temporaries	-ve Offset	
Lower Memory (↓)		

- **Function parameters** are usually stored **above** the frame pointer (positive offset).
- **Local variables** are stored **below** the frame poi ↓ (negative offset).

6. Access link points to the

(A) Current activation record
(B) Parent activation record
(C) Child activation record
(D) None of the other options

Ans: b)
Explanation:

The access link (also called a static link) is used in languages that support nested procedures or static scoping. It points to the activation record of the lexically enclosing function (parent function) rather than the function that called it.

Why (B) Parent activation record is correct:

- In static scoping, a function can be nested inside another function, and it needs access to the non-local variables of the lexically enclosing function.
- The access link helps in reaching these non-local variables.
- The control link (dynamic link), on the other hand, helps with returning to the caller, but that's different from the access link.

7. If pointer is supported in the high-level language,

(A) Must also be supported in the intermediate language
(B) May not be supported in the intermediate language
(C) Depends on language
(D) None of the other options

Ans: b)
Explanation:

High-level languages (HLL) like C, C++, and Rust support pointers explicitly. However, the intermediate language (IL) used in a compiler does not necessarily have to support pointers in the same way. Instead, the IL might represent pointers using addresses, references, or other constructs that serve a similar purpose without direct pointer manipulation.

For example:

- Some ILs, like LLVM IR, support explicit pointer operations.
- Other ILs, such as Java bytecode, do not use raw pointers but instead rely on references and garbage collection.
- Some functional and managed languages (e.g., Python, Java) may eliminate direct pointer support at the IL level.

8.

Ans: a)

9.

Ans: c)
Explanation:

Explanation:

The frame pointer (FP) is a register that points to the current activation record (stack frame) in memory. It serves as a stable reference point for accessing local variables, function parameters, and saved registers within the current function call.

How the frame pointer works:

- When a function is called, a new activation record is created on the stack.
- The frame pointer is updated to point to the start of this activation record.
- Local variables and temporary storage are accessed using negative offsets from FP.
- Function parameters (passed by the caller) are accessed using positive offsets from FP.

10 An intermediate language should be
.

(A) Close to target machine
(B) Machine independent
(C) All operators of high-level language supported
(D) All of the other options

Ans: b)

Explanation:

An intermediate language (IL) is typically designed to be machine-independent, meaning it can be used across multiple hardware architectures before being translated into machine-specific code.

Evaluating each option:

- (A) Close to target machine → Incorrect
  - Some ILs (like LLVM IR) are low-level and close to machine code, but many ILs (like Java bytecode, .NET CIL) are abstract and machine-independent.
  - The main purpose of an IL is to provide a general representation that works across different machines, so it is not necessarily close to the target machine.
- (B) Machine independent → Correct
  - ILs like Java bytecode, LLVM IR, and .NET CIL are designed to be portable and work across different platforms.
  - Machine-specific optimizations come later when the IL is compiled into actual machine code.
- (C) All operators of high-level language supported → Incorrect
  - Not all high-level language (HLL) features are directly supported in IL.
  - Many HLL constructs (like object-oriented features, closures, or high-level loops) are translated into simpler, lower-level constructs before or during IL generation.
  - IL is often a simpler, more restricted set of operations compared to HLLs.
- (D) All of the other options → Incorrect

11.

Which of the following is a key purpose of the stack pointer in an activation record?

(A) To store the base address of the current function
(B) To manage dynamic memory allocation
(C) To track the top of the runtime stack
(D) To store global variables

Answer: (C)

Explanation: The stack pointer (SP) keeps track of the top of the stack, ensuring that function calls, local variable allocation, and return addresses are managed properly. It helps in pushing and popping activation records dynamically during execution.

12.

What happens if recursion is attempted in a system with a static activation record?

(A) It executes successfully but inefficiently
(B) It results in incorrect execution since previous function calls get overwritten
(C) It executes normally without issues
(D) It leads to infinite recursion by default

Answer: (B)

Explanation: A static activation record means that activation records are allocated in a fixed location, making it impossible to handle multiple function calls at different recursion depths. When recursion is attempted, new function calls overwrite the existing activation record, leading to incorrect execution.

13.

What is the primary advantage of using intermediate code in a compiler?

(A) It simplifies code generation for multiple target machines
(B) It improves the execution speed of the compiled program
(C) It eliminates the need for optimization
(D) It ensures that all high-level language features are directly mapped to assembly

Answer: (A) It simplifies code generation for multiple target machines

Explanation: Intermediate code acts as an abstraction between the high-level source code and the final machine code, making it easier to generate target-specific code for different architectures. It also helps in portability and optimization before generating the final executable code.

END of Assignment

Compiler Design

Assignment- Week 11

TYPE OF QUESTION:MCQ

Number ofquestions:13 Total mark: 13 X 1 = 13

1. For the rule S → L := E, if L is a single variable, L.place is equal to

(A) Null
(B) Some value
(C) Constant
(D) None of the other options

Ans: B

Solution:

In **three-address code (TAC)**, the $L.place$ attribute represents the **memory location or register** where the result of L is stored.

- If L is a **single variable**, it refers to a **specific memory location or register** that stores the variable's value.
- Therefore, $L.place$ is **not null** and **not a constant**, but rather **some value (a memory location or register reference)**.

Why Not the Other Options?

- **(A) Null** → Incorrect, because $L.place$ must store a **valid location**.
- **(C) Constant** → Incorrect, because L is a **variable**, not a fixed constant.
- **(D) None of the other options** → Incorrect, because **(B) is correct.**

2. For Boolean variable B, B.truelist contains

(A) List of locations at which B is true
(B) List of locations to jump to if B is true
(C) List of locations at which B is true and the locations to branch to
(D) None of the other options

Ans: A

Explanation:

In syntax-directed translation for Boolean expressions, a truelist (B.truelistB.truelistB.truelist) is a list of locations in the generated intermediate code where a jump occurs if BBB evaluates to true.

- (A) "List of locations at which B is true" is correct because B.truelistB.truelistB.truelist stores the positions in the code where the control needs to be updated when BBB is true.

- (B) "List of locations to jump to if B is true" is incorrect because it refers to the target of the jump rather than the list of jump instructions that need backpatching.
- (C) "List of locations at which B is true and the locations to branch to" is incorrect since B.truelistB.truelistB.truelist does not include target locations, only the positions that need to be filled later.
- (D) "None of the other options" is incorrect since (A) is correct.

3. When generating code for the Boolean expression "(x ≥ y) AND (p ≠ q)", which locations are left for back patching?

   a) Falselist of x≥y
   b) Falselist of x≥y and falselist of p ≠ q
   c) Falselist of x≥y, falselist of p ≠ q, truelist of p ≠ q
   d) Truelist of x≥y, falselist of x≥y, truelist of p ≠ q, falselist of p ≠ q

   Ans: B
   Explanation:

   - **Short-circuit evaluation for AND (`&&`):**
     1. **If $x \geq y$ is false**, the entire expression is false → Jump to falselist.
     2. **If $x \geq y$ is true**, evaluate $p \neq q$.
     3. **If $p \neq q$ is false**, the whole expression is false → Jump to falselist.
     4. **If $p \neq q$ is true**, the whole expression is true → Jump to truelist.
   - The falselists that need backpatching are:
     - **falselist of $x \geq y$** (if first condition fails)
     - **falselist of $p \neq q$** (if second condition fails)

4. In three-address code, arrays are

   (A) Not supported
   (B) One dimensional
   (C) More than one dimensional
   (D) Supported via pointers

   Ans: C
   Explanation:

Three-address code (TAC) supports **both one-dimensional and multi-dimensional arrays** by comput memory addresses using **index calculations.**

For an **array A[i]** (1D array), the address is computed as:

$$\text{addr} = \text{base}(A) + i \times \text{element size}$$

For a **two-dimensional array A[i][j]**, the address calculation is:

$$\text{addr} = \text{base}(A) + (i \times \text{row size} + j) \times \text{element size}$$

This shows that **TAC supports more than one-dimensional arrays.**

5. For three address code generation of "B → B1 or M B2", M.quad is used to backpatch

(A) B1.truelist
(B) B1.falselist
(C) B2.truelist
(D) B2.falselist

Ans: B

Explanation:

The given Boolean expression follows the **grammar rule:**

$$B \rightarrow B_1 \; OR \; M \; B_2$$

where **M.quad** is used to store the next instruction index at **M's position** in the three-address code.

Step-by-Step Analysis:

1. **Short-circuit evaluation for OR (`||`):**
   - If B_1 **is true**, the entire expression is true → Jump to `B.truelist`.
   - If B_1 **is false**, we must evaluate B_2, so we need to redirect execution to **M**, where B_2 starts.
2. **Role of** $M.quad$:
   - It stores the instruction index where B_2 starts.
   - The **falselist of** B_1 contains locations that need to jump to B_2's evaluation.
3. **Backpatching**:
   - We need to **backpatch** $B_1.falselist$ with **M.quad** to ensure that if B_1 is false, control mov to B_2.

6. For the rule B → B1 and B2, the operation "B1.false = B.false" requires two passes as

(A) B1.false is not known
(B) B.false is not known
(C) Both B1.false and B.false are unknown
(D) None of the other options

Ans: B

Explanation:

For the Boolean expression $B \rightarrow B_1\ AND\ B_2$, short-circuit evaluation is used:

- If B_1 **is false**, the entire expression is false → No need to evaluate B_2.
- If B_1 **is true**, proceed to evaluate B_2.

Understanding `B1.false = B.false`

- We need to set $B_1.false$ to $B.false$ so that if B_1 is false, the entire expression is false.
- However, $B.false$ (which represents the falselist of B) **is not known at the time of processing** .

Why Two Passes Are Needed?

- In the first pass, we process B_1, but $B.false$ has **not yet been assigned** because it depends on the structure of the entire expression.
- In the second pass, once $B.false$ is determined, we **backpatch** $B_1.false$ **with** $B.false$.

7. In the rule C→C1 AND NC2 the non terminal N is used to remember the start address of:

(A) C
(B) C_1
(C) Both C_1 and C
(D) None of the other options

Ans: B

Explanation:

- The nonterminal N is placed before C_2 to store the instruction address at that point.
- This address is used to **redirect execution to** C_2 if C_1 evaluates to true.
- Thus, N **helps in backpatching** $C_1.falselist$, ensuring the correct jump to C_2 when need

8. In the rule S → if B then M S N else M S, N is used to generate a jump after

(A) then-part
(B) else-part
(C) both then- and else-part
(D) None of the other options

Ans: A

Explanation:

The given rule is:

$$S \rightarrow \text{if } B \text{ then } MSN \text{ else } MS$$

where:

- B: The Boolean condition.
- M: Stores the instruction address (used for backpatching).
- N: Helps generate a jump after the **then-part** to avoid falling into the **else-part**.

Why is N needed?

- **Control Flow in Conditional Statements:**
  - The **"then-part" executes first** if B is true.
  - If there is no jump, execution will **fall through** into the **else-part**.
  - N **generates an unconditional jump** after the **then-part** to skip the else-part when executio should continue past the entire if-else block.

9. In the rule S →if B then M S1, M holds the start address for

(A) S1
(B) S
(C) B
(D) None of the other options

Ans: A
Explanation:

The given rule is:

$$S \rightarrow \text{if } B \text{ then } MS_1$$

where:

- B: Boolean condition.
- M: A marker nonterminal used to store an instruction address.
- S_1: The statement that executes if B evaluates to true.

Role of M

- M **is placed before S_1, meaning it captures the instruction address where S_1 begins.**
- This is necessary for control flow because:
  - If B evaluates to **true**, execution must **jump to S_1**.
  - If B is **false**, execution skips S_1 entirely.

10. For three address code generation of rule "S → while M1 B do M2 S1", B.falselist is backpatched with

(A) M1.quad
(B) M2.quad
(C) Cannot be backpatched at this point
(D) None of the other options

Ans: C

Explanation:

The given rule is:

$$S \rightarrow \text{while } M_1 \ B \text{ do } M_2 \ S_1$$

where:

- M_1 stores the instruction address before evaluating B.
- B is the loop condition.
- M_2 stores the instruction address before executing S_1.
- S_1 is the body of the loop.

Understanding Backpatching in While Loops

- $B.truelist$ → Points to $M_2.quad$ (start of loop body) because if B is **true**, execution continues inside the loop.
- $B.falselist$ → Should point **outside the loop** (to the instruction after the while statement), but at this point, the exit address **is unknown** because the code after S_1 has not been processed yet.
- Since the exit address is **not yet determined**, $B.falselist$ **cannot be backpatched immediately.**

11.

In three-address code (TAC), accessing an array element typically requires:

(A) Direct assignment without indexing
(B) Computing an address using the base address and an offset
(C) Using only registers without memory references
(D) None of the other options

Ans: B

Solution:

In TAC, arrays are accessed using their base address and an index offset.
The address of an array element is computed as:
Address=Base Address+(Index×Element Size)
This ensures efficient memory access and supports multi-dimensional arrays using similar calculations.

12.

In the rule S → while M1 B do M2 S1, the non-terminal M2 is used to remember the start address of:

(A) S
(B) B
(C) S1
(D) None of the other options

Explanation:

- In three-address code (TAC), loops require backpatching to manage control flow.
- The while-loop executes S1 repeatedly as long as B evaluates to true.
- M2 stores the address of S1, so execution can jump back after completing one iteration.

Why Not the Other Options?

- (A) S → Incorrect, because S is the entire statement, not just the body.
- (B) B → Incorrect, because B is a condition, not the loop body's start point.

13.

For a Boolean expression B, the attribute B.falselist contains:

(A) List of locations where B evaluates to false
(B) List of locations to jump to if B is false
(C) List of locations where B is false and the locations to branch to
(D) None of the other options

Ans: B

END of Assignment

Compiler Design

Assignment- Week 12

TYPE OF QUESTION:MCQ

Number ofquestions:12 Total mark: 12 X 1 = 12

1. Backpatching is needed to generate intermediate code using

(A) Single pass
(B) Two passes
(C) Multiple passes
(D) None of the other options

Ans: A

Explanation: Backpatching is a technique used in compiler design, specifically in syntax-directed translation, where placeholders for jump addresses are maintained and later "patched" once the target addresses are determined. This method is commonly employed in **single-pass** compilers to handle forward jumps in control flow constructs like loops and conditional statements efficiently.

2. Jump table is suitable for

(A) Small number of cases
(B) Large number of cases
(C) Any number of cases
(D) None of the other options

Ans: B

Explanation:

A **jump table** (or branch table) is an efficient way to implement **switch-case statements** in programming. Instead of using multiple conditional checks (which can be slow for many cases), a jump table uses an **array of addresses** to directly jump to the correct case, making execution faster. This method is particularly beneficial when there are **many cases** (large number of branches) since it reduces the overhead of sequential comparisons.

3. If case values are widely spaced, it is better to use

(A) Jump table
(B) Table search
(C) Either jump table or simple table
(D) None of the other options

Ans: B

Explanation:

When **case values are widely spaced**, a **jump table** would result in a large, mostly empty table, wasting memory. Instead, a **table search** (such as a binary search or a hash table) is more efficient. This approach avoids excessive memory usage and provides efficient lookup times, making it preferable when case values are sparse.

4. Function call actions are divided into sequences

    (A) Calling and return
    (B) Calling and composition
    (C) Return and composition
    (D) None of the other options

Ans: A
Explanation:
When a function is called, the **calling sequence** handles setting up the stack frame, passing arguments, saving registers, and jumping to the function. The **return sequence** restores the stack, retrieves the return value, and resumes execution at the caller. These two phases ensure proper function execution and control flow.

5. Evaluation of actual parameters is done by

    (A) Callee
    (B) Caller
    (C) Both Caller and Callee
    (D) None of the other options

Ans: B
Explanation:
In most programming languages, the **caller** is responsible for evaluating actual parameters before passing them to the function. This is known as **call-by-value** (used in languages like C, Java for primitives) or **call-by-reference** (used in C++ with references and Java for objects). The callee simply receives the already evaluated arguments and uses them.

6. "In a callee-save register convention, who is responsible for saving registers?"

    (A) Caller
    (B) Callee
    (C) Both Caller and Callee
    (D) None of the above

"In a callee-save register convention, who is responsible for saving registers?"
Ans: B
Explanation:

In a **callee-save register convention**, the responsibility of saving and restoring certain registers falls on the **callee (the called function)**.

How it works:

- Before modifying any **callee-saved registers** (also called **non-volatile registers**), the **callee** must **save** their original values (typically by pushing them onto the stack).
- After execution, before returning control to the caller, the **callee** must **restore** these saved register values to maintain consistency.

- This ensures that the caller's register values remain unchanged across function calls.

Example (x86 calling conventions):

- In the **System V AMD64 ABI** (used in Linux), registers like **RBX, RBP, and R12-R15** are **callee-saved**, meaning the callee must preserve them.
- In Windows **stdcall** and **fastcall** conventions, specific registers follow this rule as well.

7. Local storage is created by

(A) Callee
(B) Caller
(C) Both Caller and Callee
(D) None of the other options

Ans: A
Explanation:

Local storage (such as local variables and stack frames) is typically created by the callee when a function is invoked. This is done by:

1. Allocating space on the stack (or another memory region) for local variables.
2. Adjusting the stack pointer (e.g., decrementing the stack pointer in architectures like x86).
3. Storing saved registers, return addresses, and function parameters as needed.

8. For a switch statement, the expression can result into values in the range -5 to +6. Number of entries in the jump table should be

(A) 5
(B) 6
(C) 11
(D) 12

Ans: D
Explanation:

A jump table is an array indexed based on the possible values of the expression in a switch statement. The number of entries in the jump table is determined by the range of values, calculated as:

Total entries=(Maximum value−Minimum value)+1
Given range: -5 to +6

- Maximum value = +6
- Minimum value = -5
- Total entries = (6−(−5))+1 = 6+5+1 = 12

9. For a switch statement implemented as a jump table, default_case is

(A) A part of jump table
(B) Not a part of jump table
(C) in the middle of the jump table
(D) at the beginning of the jump table

Ans: B
Explanation:

In a **switch statement implemented using a jump table**, the **default case** is **not a direct part of the jump table**. Instead, the jump table only maps **valid case values** to their corresponding addresses.

- If the switch expression **matches a case**, execution jumps to the appropriate entry in the table.
- If the expression **does not match any case**, the program needs a way to handle it.
- The **default case** is typically handled using a separate **branching mechanism** (e.g., a conditional check after a failed lookup).

10. For pair of goto based storage allocation for functions, the second goto statement transfers control to the beginning of

(A) Storage space
(B) Function code
(C) Program
(D) None of the other options

Ans: B
Explanation:

In **goto-based storage allocation for functions**, a pair of goto statements is typically used to manage control flow during function execution.

1. **First goto** → Jumps to a location where storage (e.g., stack space) is allocated for the function.
2. **Second goto** → Transfers control to the **beginning of the function code**, where execution begins.

11.

In a jump table-based switch statement, how is the index for the jump table calculated when the case values include negative numbers?

(A) Using the case value directly as the index
(B) By shifting all case values so that the smallest value maps to index 0
(C) By skipping negative case values in the jump table
(D) By using absolute values of case numbers

Ans : B

Explanation:

A jump table works by using an array where each index corresponds to a case label. However, arrays cannot have negative indices, so all case values are shifted by subtracting the smallest case value.

For example, if the case values range from -5 to +6:

- The smallest case value is -5.
- To make indexing possible, all case values are shifted by +5.
- So, case -5 maps to index 0, case -4 maps to index 1, ..., case 6 maps to index 11.

12.

Which of the following occurs during the calling sequence of a function call?

(A) Saving the return address and passing arguments
(B) Restoring registers and returning control to the caller
(C) Freeing allocated memory and clearing local variables
(D) None of the above

Ans: A

Explanation:

The calling sequence is the set of actions performed before transferring control to the function. These include:

- Saving the return address (so execution can resume after the function completes).
- Passing arguments to the function (via registers or the stack).
- Allocating stack space for local variables.
- Transferring control to the function.

END of Assignment