Q1] Explain the concept of symbol table management with a suitable example?

Ans:-

1. Symbol table is a data structure contains record for each Tokens. It stores information about all Tokens such as name, type ,size, scope and value.
2. It determine the correct use of tokens & type checking and Variable type expressions.
3. Use to record each Tokens and collect the information about it. It Locates run time storage to Tokens.
4. It stores Records for return type of function ,its arguments and its return value.
5. Returns pointer for the identifier and Null when no record is.
6. Basically, symbol table management involves creating and maintaining a data structure to store information about symbols used in a program.
7. It serves as a vital resource for compilers and interpreters to perform various analyses and generate executable code.
8. Symbol table is built in lexical and syntax analysis phase.
9. It binds names to objects.
10. The symbol table is searched every time a name is encountered in the source text. It is also used to retrieve the properties of a names.

Example:-

Symbol Table

int a, b; float c; char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Q2] Illustrate the role of symbol table manager and make list of various operations on Symbol Table.

Ans:-

The role of symbol table manager are as follows:-

1. It stores information about all Tokens such as name, type ,size, scope and value.
2. It determine the correct use of tokens & type checking and Variable type expressions.
3. Use to record each Tokens and collect the information about it. It Locates run time storage to Tokens.
4. It stores Records for return type of function ,its arguments and its return value.
5. Returns pointer for the identifier and Null when no record is.
6. Basically, symbol table management involves creating and maintaining a data structure to store information about symbols used in a program.
7. It serves as a vital resource for compilers and interpreters to perform various analyses and generate executable code.
8. Symbol table is built in lexical and syntax analysis phase.

Operations on symbol table are as follows:-

- Insert(String key , Object binding)
- Object_lookup(String key)
- Begin_scope() and end_scope()
- Handling Reserve keywords
- Insert("PLUS" , PLUS);
- Insert("MINUS" , MINUS);
- PLUS and MINUS are lexeme and other token

Q3] What are some common errors that can occur during the lexical analysis phase of a compiler and what are the different methods used for error recovery in such cases?

Ans:-

Types of error occur during the lexical analysis phase are as follows:-

- Long Identifiers or numeric characters:-

```
#include <iostream>
using namespace std;
int main() {
    int a=2147483647 +1;
    return 0;
}
```

This is a lexical error since assigned integer lies between −2,147,483,648 and 2,147,483,647.

- Apperance of invalid character:-

```
#include <iostream>
using namespace std;
int main() {
    printf("Geeksforgeeks");$
    return 0;
}
```

This is a lexical error since an illegal character $ appears at the end of the statement.

- Unmatched string:-

```
#include <iostream>
using namespace std;
int main() {
    cout<<"GFG!";
    return 0;
}
```

This is a lexical error since the ending of comment "*/" is not present but the beginning is present.

- Spelling Error:-

```
#include <iostream>
using namespace std;
int main() {
    int 3num= 1234;
    return 0;
}
```

Spelling error as identifier cannot start with a number.

- Badly formed numerical:-

```
#include <iostream>
using namespace std;
int main() {
    int 3num= 12sw34;
    return 0;
}
```

Integer value cannot be merged with alphabets.

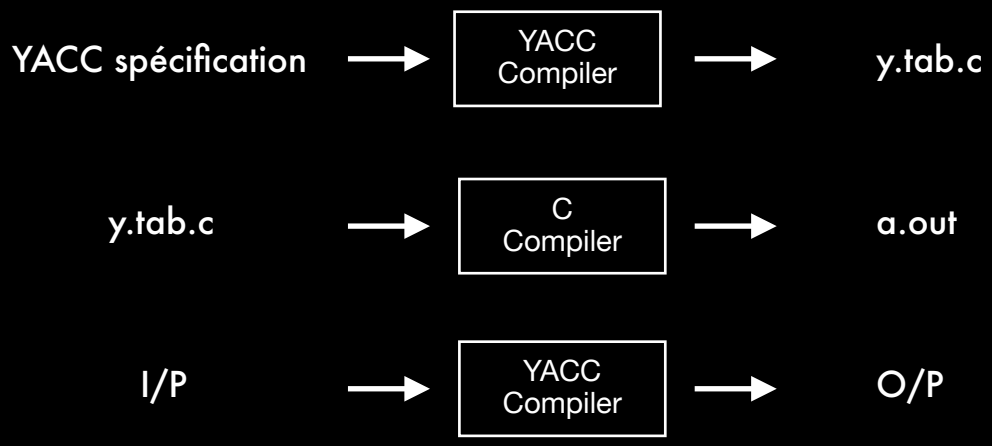
Different methods used for error recovery in such cases are as follows:-

1. Transpose of two adjacent characters.
2. Insert a missing character into the remaining input.
3. Replace a character with another character.
4. Delete one character from the remaining input.

Q4] Provide an overview of YACC (Yet Another Compiler Compiler) with labelled diagram in the context of compiler design, its purpose, and how it is utilized. Explain in short automatic error recovery in YACC.

Ans:-

1. YACC stands for Yet Another Compiler Compiler.
2. It is a tool which generates LALR Parser.
3. Syntax Analyser & parse tree.
    1. It take tokens as input
    2. As a output it produces syntax tree.
4. Diagramatic Representation–



gram.y – file containing desired grammar in yacc format

Yacc – yacc program

Y.tab.c – c source program created by YAAC

cc/gcc – c - compiler

a.out–Executable program ie. It will parse grammar given in gram.y

Automatic error recover in yaac:-

Definition – Section A

All codes b/w %% is copied to the 'c' file. It configure various parser features like defining token quotes, establishing operator precedence & associativity.

%% – partition

Rules – Section B

Here we specify the grammar rules.

%% – Partition for grammer

Supplementary Code – Section C

It is the section where error can detected & tells us that what should be removed or added.

Q5] Can you discuss the impact of error recovery on the overall efficiency and performance of LR parsing?

Ans:-

1. Increased Parsing Time: Error recovery techniques introduce additional parsing steps and logic, which can increase the parsing time, especially in the presence of frequent errors.

2. Reduces the Number of Parser Shift/Reduce Conflicts: Error recovery can help resolve shift/reduce or reduce/reduce conflicts by making decisions based on the current state and lookahead. This reduces the number of conflicts and improves the parser's efficiency.

3. May Decrease the Parser's Accuracy: Error recovery can result in the parser accepting erroneous input or recovering in a way that deviates from the intended behavior. This may lead to decreased accuracy in error-prone scenarios.

4. Enhances Fault Tolerance: Error recovery allows the parser to continue parsing after encountering errors, making the parser more fault-tolerant and capable of handling erroneous input gracefully.

5. May Increase Memory Usage: Error recovery techniques may require additional data structures and bookkeeping, increasing the memory usage of the parser and potentially impacting the overall system performance in memory-constrained environments.

Q6] Elaborate code optimization. Discuss with suitable examples about machine-independent & machine-dependent optimization.

Ans:-

1. Code optimization is the process of transforming source code to improve its efficiency, speed, and resource utilization.
2. It aims to produce optimized code that executes faster, uses less memory, and generates fewer machine instructions.

3. Machine-Independent Optimization: These optimizations focus on improving the code without considering the specific target machine architecture. Examples include constant folding, common subexpression elimination, loop optimization, and dead code elimination. These optimizations apply universally across different platforms.
    1. Ex:-

    Non-optimized code:-

    ```
    Do{
            Item = 10;
            Value = value + item;
    } while ( value < 100 );
    ```

4. Machine-Dependent Optimization: These optimizations take into account the characteristics and constraints of the target machine architecture. Examples include register allocation, instruction scheduling, and architecture-specific instruction transformations. These optimizations exploit specific hardware features and instruction sets to maximize performance on a particular machine.
    1. Ex:-

    Optimized code:-

    ```
    Item = 10;
    Do{
            Value = value + item;
    } while ( value < 100 );
    ```

Q7] Explain Directed acyclic graph (DAG) with suitable examples. Discuss various applications of DAG.

Q10] Can you explain the concept of a Directed Acyclic Graph (DAG) and its role in code generation?

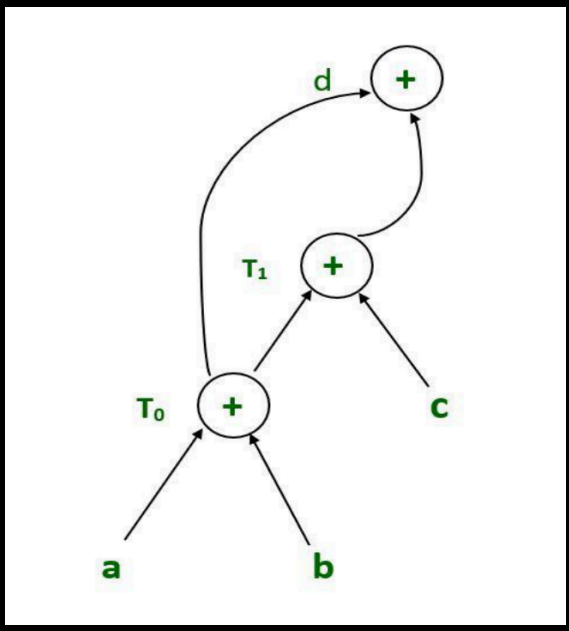
Ans:-

1. The Directed Acyclic Graph is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block.
2. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.
    1. Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
    2. The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
    3. DAG is an efficient method for identifying common sub-expressions.
    4. It demonstrates how the statement's computed value is used in subsequent statements.
3. Ex:-

    T0 = a + b –Expression 1

    T1 = T0 + c —Expression 2

    d = T0 + T1 ––Expression 3



4. Application of DAG are as follows:-
    1. Directed acyclic graph determines the subexpressions that are commonly used.
    2. Directed acyclic graph determines the names used within the block as well as the names computed outside the block.
    3. Determines which statements in the block may have their computed value outside the block.
    4. Code can be represented by a Directed acyclic graph that describes the inputs and outputs of each of the arithmetic operations performed within the code.
    5. Several programming languages describe value systems that are linked together by a directed acyclic graph.

Q8] What is basic block and how to partition a code into basic block? Explain flow graph with suitable example.

Ans:-

Basic Blocks –

1. The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit.
2. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.
3. Here, the first task is to partition a set of three-address code into the basic block.
4. The new basic block always starts from the first instruction and keep adding instructions until a jump or a label is met.
5. If no jumps or labels are found, the control will flow in sequence from one instruction to another.
6. Ex:-

```
int calculate(int a, int b) {
    int sum = a + b;
    int product = a * b;
    int result = sum - product;
    return result;
}
```

Basic Block will be–

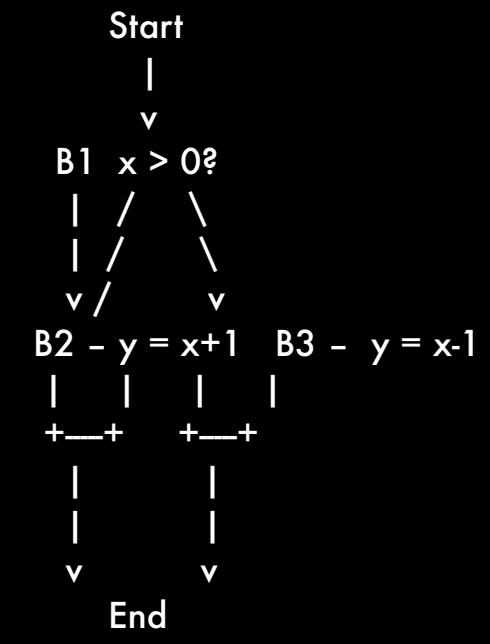
```
int sum = a + b;
int product = a * b;
int result = sum - product;
return result;
```

Flow Graph–

1. It is a directed graph. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph.
2. An edge can flow from one block X to another block Y in such a case when the Y block's first instruction immediately follows the X block's last instruction.
3. The following ways will describe the edge:
    1. There is a conditional or unconditional jump from the end of X to the starting of Y.
    2. Y immediately follows X in the original order of the three-address code, and X does not end in an unconditional jump.
4. Ex:-

```
if (x > 0) {
   y = x + 1;
} else {
   y = x - 1;
}
```

Flow graph will be–



Q9] Elaborate various loop optimization techniques with suitable data. Explain in brief Induction variable removal.

Ans:-

1. Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.
2. If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

3. For loop optimization the following three techniques are important:
    1. Code motion
    2. Induction-variable elimination
    3. Strength reduction

– Code Motion:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

– Induction-Variable Elimination

Induction variable elimination is used to replace variable from inner loop. It can reduce the number of additions in a loop. It improves both code space and run time performance.

– Reduction in Strength

Strength reduction is used to replace the expensive operation by the cheaper once on the target machine. Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop. Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Q11] What are some common peephole optimization techniques used to eliminate redundant or unnecessary instructions?

Ans:-

Peephole optimization techniques are as follows:-

Redundant load and store elimination:

In this technique, redundancy is eliminated.

Initial code:	Optimized code:

y = x + 5;
i = y;
z = i;
w = z * 3;
	y = x + 5;
	w = y * 3;

Constant folding:

The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.

Initial code:	Optimized code:
x = 2 * 3;	x = 6;

Strength Reduction:

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:	Optimized code:
y = x * 2;	y = x + x; or y = x << 1;

Null sequences/ Simplify Algebraic Expressions :

Useless operations are deleted.

a := a + 0;

a := a * 1;

a := a/1;

a := a - 0;

Q12] What is a heuristic ordering in the context of DAGs, and how does it relate to code generation?

Ans:-

The heuristic ordering algorithm attempts to make the evaluation of a node the evaluation of its leftmost argument. The algorithm shown below produces the ordering in reverse. The advantage of generating code for a basic block from its DAG representation is that from a DAG we can easily see how to rearrange the order of the final computation sequence than we can start from a linear sequence of three-address statements or quadruples.

Q13] What is a labelling algorithm in the context of code generation, and how does it work, provide suitable data?

Ans:-

Labeling algorithm is used by compiler during code generation phase. Basically, this algorithm is used to find out how many registers will be required by a program to complete its execution. Labeling algorithm works in bottom-up fashion. We will start labeling firstly child nodes and then interior nodes.

Rules of labeling algorithm are:

1. Traverse the control flow graph of the program and assign a label to the first instruction of each basic block.
2. For each jump or branch instruction, assign a label to the target basic block if it does not already have a label.
3. Repeat step 2 until all target basic blocks have labels.
4. Generate the code for each instruction, using the labels to specify the target addresses for jump and branch instructions.
5. The labeling algorithm ensures that each basic block has a unique label and that the labels are assigned in the correct order to ensure that all target basic blocks have labels.

Q14] What is a simple code generator? What are some common challenges or problems encountered during the code generation process?

Ans:-

A simple code generator is a component of a compiler that translates an intermediate representation of a program into the target machine code or assembly instructions.

It maps high-level constructs and operations from the IR to the corresponding low-level instructions that can be executed by the target hardware.

Some common challenges or problems encountered during the code generation process include:

1. Register Allocation:

   The code generator needs to allocate and manage registers effectively to minimize register spills and reloads, reducing memory access and improving performance.

2. Instruction Selection:

   Different architectures have varying instruction sets and capabilities, and the code generator must select instructions that achieve the desired functionality while considering factors like performance and code size.

3. Addressing Modes:

   Different addressing modes, such as immediate, direct, indirect, or indexed addressing, need to be handled appropriately to ensure proper data access and manipulation.

4. Control Flow and Branching:

   The code generator needs to correctly translate the control flow constructs from the IR into the corresponding branching instructions in the target machine code.

5. Code Size and Efficiency:

   Optimizing code size and improving efficiency are ongoing challenges in code generation. Minimizing unnecessary instructions, eliminating redundant computations, and applying various optimization techniques can help reduce code size and improve overall performance.

Q15] What are the main goals or objectives of peephole optimization?

Ans:-

The objective of peephole optimization is as follows:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Elaborate each and every point to get full marks

Q16] What are some best practices for designing and implementing a simple code generator?

Ans:-

Designing and implementing a simple code generator involves several considerations and best practices. Here are some key points to keep in mind:

1. Understand the Target Architecture: Gain a thorough understanding of the target hardware architecture, including its instruction set, addressing modes, memory organization, and performance characteristics. This knowledge is essential for generating efficient code that fully utilizes the capabilities of the target system.

2. Modular Design: Adopt a modular and well-organized design for the code generator. Divide the code generation process into logical components or stages, such as instruction selection, register allocation, and code emission. This modular approach allows for easier maintenance, debugging, and potential reuse of components.

3. Use an Intermediate Representation (IR): Implement and utilize an intermediate representation (IR) of the program. The IR acts as an abstraction layer between the front-end and the code generator, facilitating transformations and optimizations. Ensure that the IR captures all necessary information to generate the target code accurately.

4. Optimize the IR: Apply optimization techniques to the IR before code generation. This includes performing high-level optimizations, such as constant folding, common subexpression elimination, and dead code elimination. Optimizing the IR can simplify the code generation process and improve the quality of the generated code.

5. Efficient Instruction Selection: Develop an efficient instruction selection algorithm that maps high-level operations from the IR to target machine instructions. Consider the available instruction set and select instructions that provide the desired functionality while considering performance, code size, and architectural constraints.

6. Register Allocation: Implement an efficient register allocation algorithm that assigns variables and temporary values from the IR to hardware registers. Consider techniques like graph coloring or linear scan algorithms to minimize register spills and reloads, maximizing register utilization and performance.

7. Handle Control Flow: Properly handle control flow constructs, including conditionals, loops, and function calls. Generate appropriate branching and control instructions to maintain the correct control flow in the target code.

8. Error Handling and Debugging: Implement robust error handling mechanisms to provide informative error messages during code generation. Maintain debugging information, such as source code line numbers and variable names, to aid in the debugging process.

9. Test and Validate: Design comprehensive test suites to validate the correctness and performance of the generated code. Develop both functional and performance tests to ensure the code generator produces correct code and meets the desired performance goals.

10. Documentation and Maintainability: Document the design and implementation details of the code generator to aid in future maintenance and modifications. Use clear and consistent naming conventions and follow coding best practices to ensure the codebase is readable and maintainable.