COMPILER DESIGN

Unit – IV

1. **Explain the concept of symbol table management with a suitable example?**

OR

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

OR

3. **Organization of Symbol Table**
    1. **Linear list**
    2. **Search tree**
    3. **Hash table**

ANS : Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built-in lexical and syntax analysis phases.
- The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.
- It is used by various phases of the compiler as follows:-

    1. Lexical Analysis: Creates new table entries in the table, for example like entries about tokens.

    2. Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

    3. Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

    4. Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

5. Code Optimization: Uses information present in the symbol table for machine-dependent optimization.

6. Target Code generation: Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in the symbol table is associated with attributes that support the compiler in different phases.

Use of Symbol Table-

The symbol tables are typically used in compilers. Basically compiler is a program which scans the application program (for instance: your C program) and produces machine code.

During this scan compiler stores the identifiers of that application program in the symbol table. These identifiers are stored in the form of name, value address, type.

Here the name represents the name of identifier, value represents the value stored in an identifier, the address represents memory location of that identifier and type represents the data type of identifier.

Thus compiler can keep track of all the identifiers with all the necessary information.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by the compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage

- If structure or record then, a pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Operations on Symbol Table :

Following operations can be performed on symbol table-

1. Insertion of an item in the symbol table.

2. Deletion of any item from the symbol table.

3. Searching of desired item from symbol table.

Implementation of Symbol table –
Following are commonly used data structures for implementing symbol table:-

1. **List –**

we use a single array or equivalently several arrays, to store names and their associated information ,New names are added to the list in the order in which they are encountered . The position of the end of the array is marked by the pointer available, pointing to where the next symbol-table entry will go. The search for a name proceeds backwards from the end of the array to the beginning. when the name is located the associated information can be found in the words following next.

id1	info1	id2	info2	……..	id_n	info_n

- In this method, an array is used to store names and associated information.
- A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from the beginning of the list till available pointer and if not found we get an error "use of the undeclared name"
- While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e. "Multiple defined names"
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- The advantage is that it takes a minimum amount of space.

1. **Linked List –**
    - This implementation is using a linked list. A link field is added to each record.
    - Searching of names is done in order pointed by the link of the link field.
    - A pointer "First" is maintained to point to the first record of the symbol table.
    - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

2. **Hash Table –**

    - In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
    - A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
    - To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
    - Insertion and lookup can be made very fast – O(1).
    - The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

3. **Binary Search Tree –**

    - Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
    - All names are created as child of the root node that always follows the property of the binary search tree.
    - Insertion and lookup are $O(\log_2 n)$ on average.

Advantages of Symbol Table

1. The efficiency of a program can be increased by using symbol tables, which give quick and simple access to crucial data such as variable and function names, data kinds, and memory locations.
2. better coding structure Symbol tables can be used to organize and simplify code, making it simpler to comprehend, discover, and correct problems.
3. Faster code execution: By offering quick access to information like memory addresses, symbol tables can be utilized to optimize code execution by lowering the number of memory accesses required during execution.
4. Symbol tables can be used to increase the portability of code by offering a standardized method of storing and retrieving data, which can make it simpler to migrate code between other systems or programming languages.

5. Improved code reuse: By offering a standardized method of storing and accessing information, symbol tables can be utilized to increase the reuse of code across multiple projects.

6. Symbol tables can be used to facilitate easy access to and examination of a program's state during execution, enhancing debugging by making it simpler to identify and correct mistakes.

Disadvantages of Symbol Table

1. Increased memory consumption: Systems with low memory resources may suffer from symbol tables' high memory requirements.

2. Increased processing time: The creation and processing of symbol tables can take a long time, which can be problematic in systems with constrained processing power.

3. Complexity: Developers who are not familiar with compiler design may find symbol tables difficult to construct and maintain.

4. Limited scalability: Symbol tables may not be appropriate for large-scale projects or applications that require o the management of enormous amounts of data due to their limited scalability.

5. Upkeep: Maintaining and updating symbol tables on a regular basis can be time- and resource-consuming.

6. Limited functionality: It's possible that symbol tables don't offer all the features a developer needs, and therefore more tools or libraries will be needed to round out their capabilities.

Applications of Symbol Table

1. Resolution of variable and function names: Symbol tables are used to identify the data types and memory locations of variables and functions as well as to resolve their names.

2. Resolution of scope issues: To resolve naming conflicts and ascertain the range of variables and functions, symbol tables are utilized.

3. Symbol tables, which offer quick access to information such as memory locations, are used to optimize code execution.

4. Code generation: By giving details like memory locations and data kinds, symbol tables are utilized to create machine code from source code.

5. Error checking and code debugging: By supplying details about the status of a program during execution, symbol tables are used to check for faults and debug code.

6. Code organization and documentation: By supplying details about a program's structure, symbol tables can be used to organize code and make it simpler to understand.

4 .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 : Common errors that can occur during the lexical analysis phase of a compiler include:

1. Exceeding the length of identifiers or numeric constants.

2. Appearance of illegal characters.

3. Unmatched strings or comments.

Error recovery methods for lexical errors include:

1. **Panic Mode Recovery:**

Theory: Panic mode recovery involves skipping input until a designated synchronizing token is found, enabling the compiler to recover from lexical errors and continue parsing.

Example: Consider the following Python code snippet with an illegal character:

```python
def hello():
    print("Hello, world!$");
```

The presence of the illegal character `$` at the end of the string triggers a lexical error. Using panic mode recovery, the compiler would discard characters until it reaches a synchronizing token, such as a semicolon `;`. After recovery, the code might look like this:

```python
def hello():
    print("Hello, world!"); // $ discarded
```

2. **Global Correction:**

Theory: Global correction involves analyzing the entire input to identify and correct lexical errors, providing a more comprehensive approach to error handling.

Example: Consider the following C code snippet with a misspelled variable:

```c
#include <stdio.h>
int main() {
    int num = 10;
    pirntf("The number is: %d\n", num);
    return 0;
}
```

The misspelled function `pirntf` instead of `printf` triggers a lexical error. With global correction, the compiler might analyze the entire code and suggest corrections. After correction, the code could look like this:

```c
#include <stdio.h>
```

```
int main() {
    int num = 10;
    printf("The number is: %d\n", num); // Correction applied
    return 0;
}
```

3. **Token Rejection:**

Theory: Token rejection involves discarding tokens containing errors while allowing the parsing process to continue with the remaining input, minimizing disruption.

Example: Consider the following JavaScript code snippet with an unmatched string:

```javascript
function greet() {
    console.log('Hello, world!);
}
```

The missing closing single quote `'` at the end of the string triggers a lexical error. With token rejection, the lexer would discard the incomplete string token and continue processing the rest of the code:

```javascript
function greet() {
    console.log('Hello, world!); // Unmatched string discarded
}
```

```

**4. **Interactive Error Handling**:**

**Theory**: Interactive error handling involves interacting with the user or programmer to resolve lexical errors, providing a high level of user involvement in error resolution.

**Example**: Consider a simple calculator program where the user inputs an arithmetic expression:

```

Enter an arithmetic expression: 2 * (3 + 4)) - 5

```

The extra closing parenthesis `)` at the end of the expression is a lexical error. In an interactive environment, the compiler might prompt the user to correct the expression:

```

Enter an arithmetic expression: 2 * (3 + 4) - 5

```

Here, the user corrected the expression by removing the extra closing parenthesis, allowing the compiler to proceed with evaluation.
```

5 .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 : YACC (Yet Another Compiler Compiler) is a tool used in compiler design for generating the syntax analyzer (parser) of a compiler. Its purpose is to automate the process of generating a parser for a given grammar specification. YACC takes a formal grammar specification, typically in the form of a set of production rules, and generates a parser in C or other programming languages.

****Overview of YACC:****

1. **Grammar Specification**: The first step in using YACC is to define the grammar of the language to be parsed. This grammar is typically specified using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).

2. **YACC Input File:** The grammar specification is written in a file with a `.y` extension, which serves as input to the YACC tool.

3. **YACC Compilation:** The YACC tool processes the input grammar file and generates a parser code in C or other target programming languages.

4. **Parser Integration:** The generated parser code is then integrated into the compiler project, typically alongside other components such as the lexer and semantic analyzer.

5. **Compilation:** The compiler project, including the generated parser, is compiled to produce an executable compiler.

****Labelled Diagram:****

```
+-------------------------+
|     YACC Input File     |
| (Grammar Specification) |
+-------------------------+
           |
           V
      YACC Compiler
           |
           V
```

Generated Parser Code

|

V

Compiler Project

(Integrated with Lexer,

Semantic Analyzer, etc.)

|

V

Executable Compiler

```

YACC, or Yet Another Compiler Compiler, serves the purpose of automating the generation of syntax analyzers (parsers) for compilers or other language processing tools. It simplifies the process of creating parsers by allowing developers to specify the grammar of a language using formal notation and automatically generating parser code based on this specification.

****Purpose of YACC:****

**1. ****Automated Parser Generation****: YACC automates the process of generating parsers for compilers or other language processing tools. Developers can specify the grammar of a language, and YACC handles the task of generating parser code from this specification.

**2. ****Language Independence****:** YACC is language-independent, meaning it can be used to generate parsers for a wide range of programming languages or domain-specific languages. Developers can define grammars for different languages and use YACC to generate parsers tailored to those languages.

**3. ****Efficiency****:** YACC-generated parsers are typically efficient in terms of memory usage and parsing speed. The generated parsers are often implemented using techniques such as LR parsing, which provides fast and efficient parsing of input code.

**4. ****Error Handling****:** YACC provides mechanisms for error handling and recovery during parsing, allowing parsers to gracefully handle syntax errors in the input code and continue parsing as much of the code as possible.
```

Utilization of YACC:

1. **Compiler Construction**: YACC is commonly used in the construction of compilers for programming languages. Developers can use YACC to generate parsers for the syntax analysis phase of a compiler, which is responsible for recognizing the structure of the input code according to the language's grammar.

2. **Interpreter Development:** YACC can also be utilized in the development of interpreters for scripting languages or domain-specific languages. Interpreters often require parsers to analyze and execute the input code, and YACC can automate the generation of these parsers.

3. **Language Processing Tools:** YACC is used to develop various language processing tools beyond compilers and interpreters. These tools may include source code analyzers, code generators, syntax highlighters, and more, where parsing of the input code is necessary.

4. **Educational Purposes:** YACC is frequently used in educational settings to teach compiler construction concepts. Students learn about formal grammars, parsing algorithms, and language design principles by using YACC to implement parsers for simple languages.

Automatic Error Recovery in YACC:

YACC provides a mechanism for automatic error recovery during parsing. This is achieved through error productions, which are special rules in the grammar designed to handle common syntax errors gracefully.

1. **Error Productions:** Error productions are rules that specify how to recover from a syntax error encountered during parsing. These rules are triggered when the parser encounters an unexpected input that does not match any valid production.

2. **Error Tokens:** YACC allows the definition of special error tokens that represent unexpected input. When the parser encounters such tokens, it can apply error productions to recover from the error.

3. **Error Handling Actions:** Error productions include semantic actions that define how the parser should recover from the error. These actions may involve discarding input tokens, inserting missing tokens, or other strategies to synchronize the parser and continue parsing.

4. **Error Messages:** YACC can also be configured to generate meaningful error messages to aid developers in diagnosing and fixing syntax errors in the input source code.

Overall, automatic error recovery in YACC enhances the robustness of the generated parser by enabling it to gracefully handle syntax errors and continue parsing, even in the presence of erroneous input.

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

ANS : Error recovery plays a significant role in the overall efficiency and performance of LR parsing, which is a parsing technique commonly used in the construction of parsers generated by tools like YACC (Yet Another Compiler Compiler). LR parsers are powerful because they can handle a wide range of grammars efficiently, but their performance can be affected by the presence of errors in the input code. Let's discuss the impact of error recovery on LR parsing efficiency and performance:

1. **Minimizing Disruption:** Error recovery mechanisms help LR parsers to minimize disruption caused by syntax errors in the input code. Instead of halting parsing at the first error encountered, error recovery techniques allow the parser to recover from errors and continue parsing as much of the code as possible. This helps in maintaining the efficiency of the parsing process by reducing the need for repeated parsing attempts.

2. **Maintaining Parsing Speed:** Efficient error recovery techniques ensure that LR parsers can maintain parsing speed even in the presence of errors. By quickly identifying and recovering from errors, LR parsers can continue parsing subsequent portions of the input code without significant slowdowns. This is crucial for applications where parsing speed is a priority, such as in compilers handling large codebases.

3. **Reducing Memory Usage:** LR parsing algorithms typically require maintaining a parse stack to track the parsing state and facilitate parsing decisions. Inefficient error recovery strategies may lead to excessive stack usage, increasing memory consumption. However, effective error recovery mechanisms can help LR parsers recover from errors without excessively growing the parse stack, thereby reducing memory usage and improving overall efficiency.

4. **Ensuring Robustness:** Robust error recovery techniques enhance the resilience of LR parsers to handle various types of errors effectively. LR parsers with robust error recovery capabilities can gracefully recover from common syntax errors, such as missing semicolons or unmatched parentheses, without compromising the integrity of the parsing process. This ensures that the parser remains reliable in real-world scenarios where input code may contain errors.

5**. **Maintaining Error Reporting Accuracy**:** While error recovery focuses on enabling the parser to continue parsing despite errors, it is also essential to ensure accurate error reporting to aid developers in debugging their code. Efficient error recovery mechanisms should provide informative error messages that accurately pinpoint the location and nature of syntax errors in the input code. This helps developers identify and rectify errors efficiently, ultimately improving the development process.

In conclusion, error recovery techniques have a significant impact on the overall efficiency and performance of LR parsing. By enabling LR parsers to handle syntax errors effectively while maintaining parsing speed, minimizing disruption, reducing memory usage, ensuring robustness, and providing accurate error reporting, efficient error recovery mechanisms contribute to the reliability and effectiveness of LR parsing in various applications.

- **Short Note on :**
- **Storage allocation:**

STORAGE ALLOCATION

- One of the important tasks that a compiler must perform is to allocate the resources of the target machine to represent the data objects that are being manipulated by the source program.
- That is, a compiler must decide the run-time representation of the data objects in the source program.
- Source program run-time representations of the data objects, such as integers and real variables, usually take the form of equivalent data objects at the machine level;
- whereas data structures, such as arrays and strings, are represented by several words of machine memory

❖ **Activation Record:**

- **The activation record contains the following information:**
- 1. Temporary values, such as those arising during the evaluation of the expression.
- 2. Local data of a procedure. The information about the machine state (i.e., the machine status) just before a procedure is called, including PC values and the values of these registers that must be restored when control is relinquished after the procedure.
- 3. Access links (optional) referring to non-local data that is held in other activation records. This is not required for a language like FORTRAN, because non-local data is kep in fixed place. But it is required for Pascal.
- 4. Actual parameters (i.e., the parameters supplied to the called procedure). These parameters may also be passed in machine registers for greater efficiency.
- 5. The return value used by called procedure to return a value to calling procedure. Again, for greater efficiency, a machine register may be used for returning values.

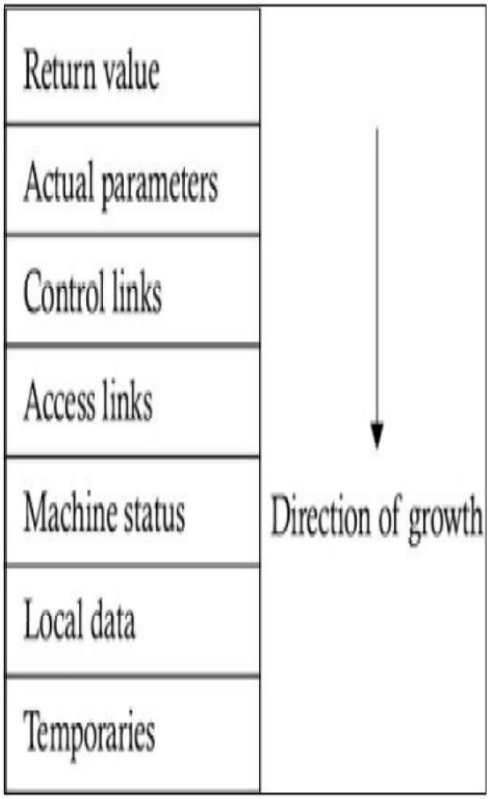


Figure 8.2: Typical format of an activation record.

ACTIVATION OF THE PROCEDURE AND THE ACTIVATION RECORD

- Each execution of a procedure is referred to as an activation of the procedure.
- This is different from the procedure definition, which in its simplest form is the association of an identifier with a statement; the identifier is the name of the procedure, and the statement is the body of the procedure.
- If a procedure is non-recursive, then there exists only one activation of procedure at any one time.
- Whereas if a procedure is recursive, several activations of that procedure may be active at the same time.
- The information needed by a single execution or a single activation of a procedure is managed using a contiguous block of storage called an "activation record"

UNIT-05

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

ANS : The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Machine Dependent Code Optimization	**Machine Independent Code Optimization**
Machine dependent code optimization refers to optimizing code specifically for a particular type of computer or hardware platform	Machine independent code optimization refers to optimizing code for use on any type of hardware.
The purpose of machine dependent code optimization is to take advantage of specific features or characteristics of a particular hardware platform.	The purpose of machine independent code optimization is to make the code as efficient as possible across all hardware platforms.
Machine dependent code optimization is limited to a specific hardware platform.	Machine independent code optimization is applicable to any hardware platform.
Machine dependent code optimization may result in code that is incompatible with other hardware platforms	Machine independent code optimization produces code that is compatible with all hardware platforms
Machine dependent code optimization may make it more difficult to port the	Machine independent code optimization makes it easier to port the

code to other hardware platforms.Machine-independent code optimization aims	code to other platforms.
Machine dependent code optimization may require more maintenance and updates to keep the code optimized for different hardware platforms.	Machine independent code optimization requires fewer updates and is easier to maintain.
Machine dependent code optimization may take longer to develop, as it requires specialized knowledge of the hardware platform and may require testing on multiple platforms.	Machine independent code optimization can be developed more quickly.
Machine dependent code optimization may result in better performance on the specific hardware platform.	Machine independent code optimization may result in more consistent performance across all platforms.
Machine dependent code optimization may require more complex code, as it needs to take into account the specific features of the hardware platform.	Machine independent code optimization can use simpler, more generic code.
Machine dependent code optimization may not be reusable on other hardware platforms.	Machine independent code optimization can be easily reused on any platform.

EXAMPLES :

Machine Independent Optimization:

Example 1: Constant Folding Consider the following expression:

int result = 10 * 5 + 3;

During machine-independent optimization, the compiler can evaluate constant expressions at compile time. In this case, the expression 10 * 5 can be calculated to 50, and then 50 + 3 can be evaluated to 53. So, the optimized code would be:

int result = 53;

Machine Dependent Optimization :

Example: Instruction Selection

The x86 architecture provides various instructions for performing arithmetic operations, such as add, sub, mul, etc. During machine-dependent optimization, the compiler selects the most efficient instructions for the target architecture.

In our example, the compiler might choose to use the add and imul instructions for addition and multiplication, respectively, as they are optimized for integer arithmetic on x86 processors.

So, the optimized machine code might look like this:

```
mov eax, x    ; Load x into register eax

add eax, y    ; Add y to eax

imul eax, z   ; Multiply eax by z

mov a, eax    ; Store the result in variable a
```

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

OR

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

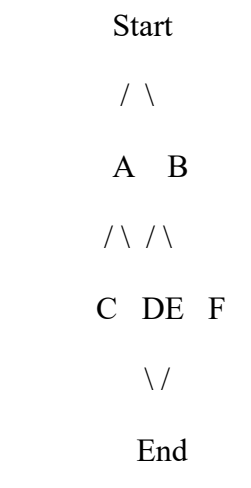
ANS : **Directed Acyclic Graph (DAG):**

A Directed Acyclic Graph (DAG) is a finite directed graph with no directed cycles. In other words, it's a graph where edges have a direction and there are no cycles, meaning you cannot follow a sequence of edges to return to a vertex you've already visited. DAGs have vertices (nodes) connected by directed edges, and each edge has a direction from one vertex to another.

****Example of DAG**:**

Consider a project management scenario where tasks need to be completed in a specific order without any circular dependencies. Each task can be represented as a node, and directed edges represent dependencies between tasks. For example:

```
```




```

In this DAG:

- Nodes represent tasks or activities.

- Directed edges indicate the order of completion or dependency between tasks.

- There are no cycles, so it's a Directed Acyclic Graph.

**Role of DAG in Code Generation**:

1. **Expression Representation**: In compilers, expressions are represented using DAGs to optimize code generation. Each node represents an operation or value, and edges represent dependencies between them.

2. **Optimizations**:

   - **Common Subexpression Elimination (CSE)**: DAGs help identify and eliminate redundant computations by recognizing common subexpressions and reusing their results.

   - **Dead Code Elimination**: DAGs aid in removing unreachable or redundant code segments from the intermediate representation.

   - **Constant Folding and Propagation**: DAGs facilitate evaluating constant expressions at compile-time and propagating constant values through the graph.

   - **Register Allocation**: DAGs assist in allocating registers efficiently by analyzing data flow within the graph.

3. **Instruction Selection**: DAGs are used to select machine instructions during code generation. By analyzing the structure of the DAG and target architecture, compilers choose appropriate machine instructions for operations represented in the DAG.
```

4. **Loop Optimization:** DAGs help optimize loops by identifying loop-invariant computations and facilitating loop unrolling and other optimizations.

****Applications of DAG**:**

1. **Data Flow Analysis:** DAGs are used in data flow analysis to represent dependencies between data values and optimize data flow within programs.

2. **Dependency Resolution:** DAGs are employed in dependency resolution problems, such as in project management or software build systems, to ensure tasks are executed in the correct order without circular dependencies.

3. **Task Scheduling:** DAGs are used in task scheduling algorithms to schedule tasks efficiently while satisfying precedence constraints.

4. **Circuit Design:** In digital circuit design, DAGs are used to represent circuit structures and optimize circuit layouts.

5. **Genetic and Evolutionary Algorithms:** DAGs are utilized in genetic and evolutionary algorithms for representing solutions and optimizing genetic operations.

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

Basic Blocks

A basic block is a **sequence of consecutive statements** in which flow of control **enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.**

The following sequence of three-address statements forms a basic block

```
t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
```

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
   - The first statement is a leader.
   - Any statement that is the target of a conditional or unconditional goto is a leader.
   - Any statement that immediately follows a goto or conditional goto statement is a leader.

1. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Construct the Basic Blocks for the following statements

```
I = 1
While (I < 10)
{
        a = b[i] + c;
        i++;
}
Result = a + c
```

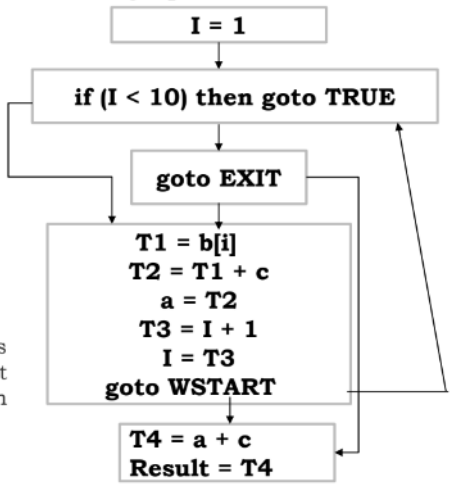
```
            I = 1
WSTART:     if (I < 10) then goto TRUE
            goto EXIT
TRUE:       T1 = b[i]
            T2 = T1 + c
            a = T2
            T3 = I + 1
            I = T3
            goto WSTART
EXIT:       T4 = a + c
            Result = T4
```

```
L1:     I = 1
L2:     if (I < 10) then goto TRUE
L3:     goto EXIT
L4:     T1 = b[i]
        T2 = T1 + c
        a = T2
        T3 = I + 1
        I = T3
        goto WSTART
L5:     T4 = a + c
        Result = T4
```

Flow Graphs

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks.

```
L1:     I = 1
L2:     if (I < 10) then goto TRUE
L3:     goto EXIT
L4:     T1 = b[i]
        T2 = T1 + c
        a = T2
        T3 = I + 1
        I = T3
        goto WSTART
L5:     T4 = a + c
        Result = T4
```



- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.

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

ANS : Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

Loop Optimization is a machine independent optimization. Whereas Peephole optimization is a machine dependent optimization technique.

Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

Loop Optimization Techniques

In the compiler, we have various loop optimization techniques, which are as follows:

1. Code Motion (Frequency Reduction)

In frequency reduction, the amount of code in the loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

Example:

Before optimization:

```
while(i<100)
{
 a = Sin(x)/Cos(x) + i;
 i++;
}
```

After optimization:

```
t = Sin(x)/Cos(x);
while(i<100)
{
 a = t + i;
 i++;
}
```

2. Induction Variable Elimination

If the value of any variable in any loop gets changed every time, then such a variable is known as an induction variable. With each iteration, its value either gets incremented or decremented by some constant value.

Example:

Before optimization:

```
B1
i:= i+1
x:= 3*i
y:= a[x]
if y< 15, goto B2
```

In the above example, i and x are locked, if i is incremented by 1 then x is incremented by 3. So, i and x are induction variables.

After optimization:

```
B1
i:= i+1
x:= x+4
y:= a[x]
if y< 15, goto B2
```

3. Strength Reduction

Strength reduction deals with replacing expensive operations with cheaper ones like multiplication is costlier than addition, so multiplication can be replaced by addition in the loop.

Example:

Before optimization:

```
while (x<10)
  {
   y := 3 * x+1;
   a[y] := a[y]-2;
   x := x+2;
  }
```

After optimization:

```
t= 3 * x+1;
    while (x<10)
       {
         y=t;
         a[y]= a[y]-2;
         x=x+2;
         t=t+6;
       }
```

4. Loop Invariant Method

In the loop invariant method, the expression with computation is avoided inside the loop. That computation is performed outside the loop as computing the same expression each time was overhead to the system, and this reduces computation overhead and hence optimizes the code.

Example:

Before optimization:

```
for (int i=0; i<10;i++)
t= i+(x/y);
...
end;
```

After optimization:

```
s = x/y;
for (int i=0; i<10;i++)
t= i+ s;
...
end;
```

5. Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Example:

Before optimization:

```
for (int i=0; i<5; i++)
    printf("Pankaj\n");
```

After optimization:

```
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
```

11. What are some common peephole optimization techniques used to eliminate redundant or unnecessary instructions?

OR

15. What are the main goals or objectives of peephole optimization?

ANS : Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.
The small set of instructions or small part of code on which peephole optimization is performed is known as ***peephole*** *or* ***window****.*
It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

Objectives of Peephole Optimization:

The objective of peephole optimization is as follows:

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

Peephole Optimization Techniques

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

Initial code:

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

Optimized code:

```
y = x + 5;
w = y * 3; //* there is no i now

//* We've removed two redundant variables i & z whose value were just being copied from one another.
```

B. 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:

```
x = 2 * 3;
```

Optimized code:

x = 6;

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

y = x * 2;

Optimized code:

y = x + x; or y = x << 1;

Initial code:

y = x / 2;

Optimized code:

y = x >> 1;

D. Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.

a := a + 0;

a := a * 1;

a := a/1;

a := a - 0;

E. Combine operations: Several operations are replaced by a single equivalent operation.

F. Deadcode Elimination:- Dead code refers to portions of the program that are never executed or do not affect the program's observable behavior. Eliminating dead code helps improve the efficiency and performance of the compiled program by reducing unnecessary computations and memory usage.

Initial Code:-

```
int Dead(void)
{
   int a=10;
   int z=50;
   int c;
   c=z*5;
   printf(c);
```

```
   a=20;
   a=a*10; //No need of These Two Lines
   return 0;
   }
```

Optimized Code:-

```
int Dead(void)
{
   int a=10;
   int z=50;
   int c;
   c=z*5;
   printf(c);
   return 0;
   }
```

GOALS:

1. **Enhance Code Quality:**

Peephole optimization aims to improve the quality of generated code by eliminating redundant or unnecessary instructions. By analyzing small sequences of instructions, the compiler can identify opportunities for optimization and transform the code to make it more concise, efficient, and maintainable. Enhanced code quality leads to better readability, easier debugging, and overall higher performance of the compiled program.

2. **Reduce Execution Time:**

One of the primary objectives of peephole optimization is to reduce the execution time of the compiled program. By eliminating redundant computations, unnecessary memory accesses, and inefficient instruction sequences, peephole optimization helps streamline the execution path of the program, leading to faster runtime performance. Reduced execution time translates to quicker program execution, improved responsiveness, and enhanced user experience.

3. **Optimize Resource Usage:**

Peephole optimization also focuses on optimizing the utilization of hardware resources such as CPU registers, memory, and cache. By minimizing the number of instructions executed and reducing unnecessary memory accesses, peephole optimization helps conserve valuable resources and improve overall system efficiency. Optimized resource usage results in better scalability, reduced power consumption, and enhanced performance on resource-constrained environments.

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

ANS : In the context of Directed Acyclic Graphs (DAGs) and code generation, heuristic ordering refers to a strategy or algorithm used to determine the order in which nodes (representing operations or expressions) should be processed or evaluated. Heuristic ordering is essential for optimizing code generation and improving the efficiency of generated code.

Here's how heuristic ordering relates to code generation in the context of DAGs:

1. **Expression Evaluation:**

In many cases, expressions represented by DAGs involve multiple nodes representing operations such as addition, multiplication, and function calls. Heuristic ordering determines the sequence in which these operations should be performed to minimize redundant computations and maximize efficiency. For example, performing computations involving constants or common subexpressions early in the evaluation process can reduce the overall number of instructions needed.

2. **Instruction Selection:**

Heuristic ordering is crucial for selecting machine instructions during code generation. By analyzing the structure of the DAG and considering factors such as instruction latencies, resource constraints, and target architecture characteristics, a suitable order for generating machine instructions can be determined. This ensures that instructions are generated in an optimal sequence, taking advantage of pipelining, instruction parallelism, and other hardware features.

3. **Register Allocation:**

When performing register allocation, heuristic ordering helps determine the order in which variables are assigned to registers. By prioritizing frequently used variables or variables with long lifetimes, heuristic ordering can minimize register spills and reloads, leading to more efficient register allocation and improved performance of the generated code.

4. **Optimization Techniques:**

Heuristic ordering is also relevant in various optimization techniques applied to DAGs during code generation. For example, during common subexpression elimination or constant propagation, the order in which nodes are processed can affect the effectiveness of these optimizations. Heuristic ordering strategies aim to prioritize nodes that are likely to yield the most significant optimization benefits, thereby improving the overall quality and efficiency of the generated code.

13. 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:

Traverse the control flow graph of the program and assign a label to the first instruction of each basic block.

For each jump or branch instruction, assign a label to the target basic block if it does not already have a label.

Repeat step 2 until all target basic blocks have labels.

Generate the code for each instruction, using the labels to specify the target addresses for jump and branch instructions.

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.

The advantages of the labeling algorithm include:

1. Efficient code generation: The labeling algorithm ensures that the code generated for jump and branch instructions is correct and efficient.
2. Flexibility: The labeling algorithm can be adapted to a wide range of programming languages and architectures.
3. Easy to implement: The labeling algorithm is a simple and easy-to-implement technique.

However, the labeling algorithm also has some disadvantages:

1. Limited optimization: The labeling algorithm does not optimize the code generated for jump and branch instructions.
2. Increased code size: The labels generated by the labeling algorithm increase the size of the code.

In summary, the labeling algorithm is a technique used in compiler design to generate labels for instructions during code generation. It ensures that the code generated for jump and branch instructions is correct and efficient, but it does not optimize the code and can increase the size of the code.

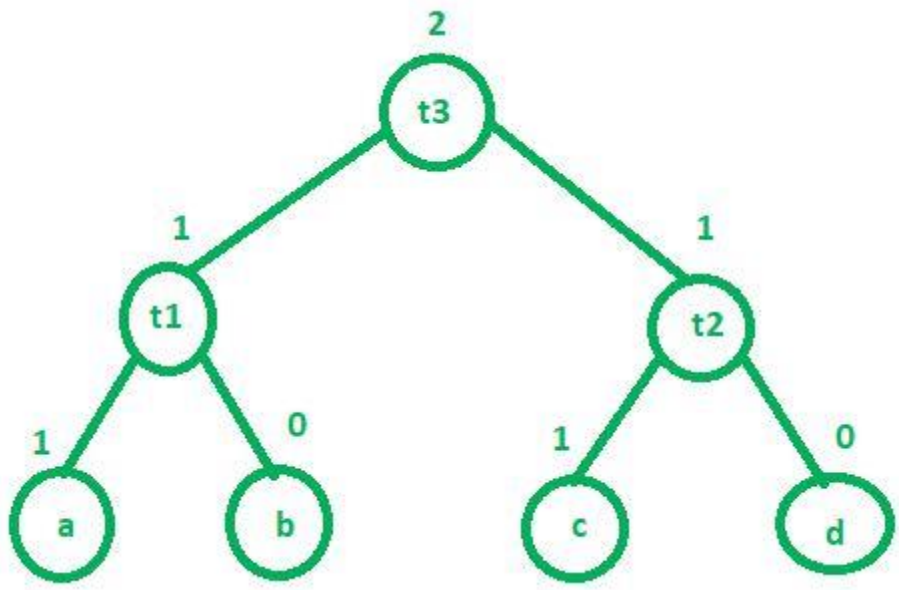
1. **If ‘n’ is a leaf node –**
    - **a.** If ‘n’ is a left child then its value is 1.
    - **b.** If ‘n’ is a right child then its value is 0.
2. **If ‘n’ is an interior node –** Lets assume L1 and L2 are left and right child of interior node respectively.
    - **a.** If L1 == L2 then value of ‘n’ is L1 + 1 or L2 + 1
    - **b.** If L1 != L2 then value of ‘n’ is MAX(L1, L2)

Example: Consider the following three address code:

```
t1 = a + b

t2 = c + d

t3 = t1 + t2
```



Above three address code will require maximum 2 registers to complete its execution. There is a function called getregister() which is used by compiler to decide where the result will get stored. There are 4 cases of this function which are as follows:

1. If there is a register R which is not holding multiple values then we can use this register to store the value of our result (in the above example, we can store t3 in R, provided that current value in R is not being used anywhere in the program.).
2. If first condition is not satisfied then compiler will search for any empty register to store the value of our result (t3).
3. If there are no empty registers then swap the contents of any registers into the memory and store the result (t3) in that register, provided that those contents do not have any next use.
4. If all the 3 conditions do not hold then store the result in any free memory location.

14. 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 responsible for translating intermediate representations (such as Abstract Syntax Trees or Three-Address Code) into executable machine code or assembly language instructions. It typically performs straightforward translation without sophisticated optimizations or complex analysis.

Here's a general overview of how a simple code generator works:

1. **Intermediate Representation (IR) Parsing:** The code generator receives an intermediate representation (IR) of the program generated by the front-end of the compiler.

2. **Traversal:** The code generator traverses the IR, typically using depth-first or top-down traversal algorithms, to visit each node and generate corresponding code.

3. **Code Generation:** For each node in the IR, the code generator emits assembly language instructions or machine code sequences that perform the equivalent computation or operation.

4. **Output Generation:** The generated code is written to an output file or memory buffer, ready for further processing or execution.

Despite its simplicity, code generation can face several challenges and problems:

1. **Target Architecture Compatibility:** Ensuring that the generated code is compatible with the target architecture, including its instruction set, memory model, and hardware

features, can be challenging, especially when targeting diverse architectures.

2. **Register Allocation:** Assigning variables to processor registers efficiently while minimizing spills and reloads poses a challenge, particularly in architectures with a limited number of registers or complex register interdependencies.

3. **Instruction Selection:** Choosing the most appropriate machine instructions to implement high-level language constructs while considering factors such as instruction latency, throughput, and resource constraints can be challenging, especially for architectures with diverse instruction sets.

4. **Control Flow Handling:** Generating efficient code for control flow constructs such as loops, conditionals, and function calls while maintaining correctness and minimizing overheads can be challenging, especially when dealing with complex control flow patterns or optimizing for speculative execution.

5. **Data Layout Optimization:** Optimizing data layout and memory access patterns to minimize cache misses, improve data locality, and exploit memory hierarchy efficiently can be challenging, particularly in applications with large datasets or irregular memory access patterns.

6. **Error Handling and Recovery:** Dealing with errors and exceptions during code generation, such as syntax errors, type mismatches, or out-of-memory conditions, while providing informative error messages and graceful recovery mechanisms, can be challenging, requiring robust error handling and recovery strategies.

Addressing these challenges often requires a combination of sophisticated algorithms, optimization techniques, and architectural insights tailored to the specific characteristics of the target platform and the requirements of the application being compiled.

16. 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 to ensure efficiency, maintainability, and correctness. Here are some best practices for designing and implementing a simple code generator:

1. **Understand the Target Architecture:** Gain a thorough understanding of the target architecture, including its instruction set, memory model, registers, and hardware features. This understanding will guide the code generation process and ensure compatibility with the target platform.

2. **Modular Design:** Design the code generator as a modular component of the compiler, separating concerns such as parsing, intermediate representation (IR) generation, code generation, and optimization. This modular design facilitates maintainability, extensibility, and code reuse.

3. **Use an Intermediate Representation (IR): Utilize an intermediate representation (IR) to represent the program's semantics and facilitate code generation. Choose a suitable IR that strikes a balance between simplicity and expressiveness, such as Abstract Syntax Trees (AST) or Three-Address Code (TAC).

4. **Traversal Algorithms:** Implement traversal algorithms, such as depth-first or top-down traversal, to traverse the intermediate representation (IR) and visit each node for code generation. Choose traversal algorithms that suit the structure of the IR and optimize for efficiency.

5. **Code Generation Rules:** Define clear and concise code generation rules for translating each node in the intermediate representation (IR) into corresponding assembly language instructions or machine code sequences. Ensure that code generation rules cover all language constructs and handle edge cases appropriately.

6. **Optimization Techniques:** Consider incorporating simple optimization techniques, such as constant folding, common subexpression elimination, and dead code elimination, into the code generation process. These optimizations can improve the efficiency and quality of the generated code without introducing significant complexity.

7. **Testing and Validation:** Develop comprehensive test suites to validate the correctness and performance of the code generator. Test various language constructs, control flow patterns, and optimization scenarios to ensure robustness and reliability.

8. **Documentation and Comments:** Document the design decisions, algorithms, and implementation details of the code generator thoroughly. Include comments in the code to clarify complex logic, edge cases, and optimization strategies for future reference and

maintenance.

9. **Performance Profiling:** Profile the performance of the code generator to identify bottlenecks and areas for optimization. Measure the compilation time, memory usage, and generated code quality to assess the effectiveness of the code generator and identify opportunities for improvement.

10. **Continuous Improvement:** Iterate on the design and implementation of the code generator based on feedback, performance metrics, and evolving requirements. Continuously refine the code generator to enhance efficiency, maintainability, and compatibility with new architectures or language features.

By following these best practices, you can design and implement a simple code generator that efficiently translates high-level language constructs into optimized machine code or assembly language instructions, contributing to the overall effectiveness and quality of the compiler.