Compiler Design

Unit IV Storage allocation & Error Handling

Any 3
- Symbol table (Q1, Q2, Q3) -
- Common errors during lexical analysis phase (Q4) -
- YACC (In short) (Q5) -

- Short Note
  - → Storage Allo
  - → Activation

+ and (Copy Numericals)

Unit V Code Optimization

- Block + Flowgraph + DAG (Numericals) [Copy]
- DAG + BLOCK (Theory) (Q7 & Q8)

Any 1
- Peephole -
- Loop Optimization

Unit VI Code Generation

Any 2
- Heuristic ordering (DAG) (Q12)
- Labelling algorithm (Q13)
- Simple Code generator (Q14 & Q16) [In short]

(Numericals in Copy if any)

Unit 4

Q Ans

Symbol Table

Definition :- The symbol table is defined as the set of Name & Value pair.

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 instances of various entities such as variable & 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 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 check) 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.

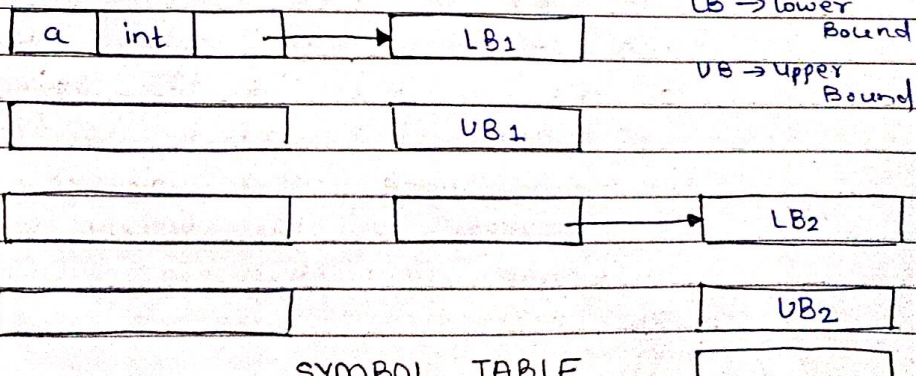
> Information used by the compiler from symbol table

- Data type & 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 & 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 & free storage of symbol table
• lookup	- to search for a name and return pointer to its entry
• insert	- to insert a name in symbol table and return pointer to its entry
• set_attribute	- to associate an attribute with a given entry
• get_attribute	- to get an attribute associate with a given entry.



LB → lower Bound
UB → Upper Bound

> Various Approaches to symbol table organization.

① The Linear List
- It is the easiest way to implement a symbol table.
- The new names are added to the table in the order that they arrive
- Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the same name is already present in the table
- If the name is not present, then the record for new name is created and added to the list at a position specified by the avaliable pointer

② Binary search tree
- A search tree is more efficient approach to symbol table.
- We add 2 links, left & right, in each record and these links point to the records in the search tree.
- All names a created as child of the root node that always follows the property of the binary search tree
- Insertion & lookup are O(log₂n) on average.

③ Hash table
- In hash scheme, 2 tables are maintained - a hash table & symbol table and are the most commonly used methods
- A Hash table is an array with an index range : 0 to table size -1.
- These entries and pointers pointing to the names of the symbol table.
- Insertion and lookup can be made very fast - O(1).
→ Adv :- Quick to search
→ Dis :- Hashing is complicated to implement

Advantages
- Improved code reuse
- Faster Code execution
- Better Coding structure

Disadvantages
- Increased memory consumption
- Increased processing time
- Complexity
- Limited Scalability & functionality.

Applications
- Resolution of variable & function names
- Resolution of scope issues
- Error checking & Code debugging

Q 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 occuring in lexical analysis phase
  - Long Identifiers
  - Too long numerical literals
  - Badly formed numerical literals
  - Input characters that are not in source code
  - Spelling mistakes
  - Unmatched string or comments.

⇒ Error recovery methods for lexical errors including.

1. Panmic Mode Recovery
   - Involves skipping input untill a designated synchronized token is found
   - Enabeling the compiler to recover from lexical errors & continue parsing

Example: Consider the following Python Code with an illegal character.

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

The presence of 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
   - Involves analyzing the input (entire i/p) to identify and correct lexical errors.

Example: Consider the following C code with a mis-spelling variable:

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

The mis-spelling function 'pirntf' instead of 'printf' triggers a lexical error. With global correction, the compiler might analyze the entire code & suggest corrections. After correction, the code look like this:

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

3. Interactive Error Handling
- Involves interacting with the user or programmar to resolve lexical errors
- Providing a high level of user involvement in error resolution

Example : Consider the following user input
Enter on arithmetic expression : 2*(3+4))-5
An extra closing parenthesis is seen ")" causing lexical error, to resolve it the user might be asked to correct the expression using ~~an~~ interactive environment.

Corrected expression :-

2*(3+4)-5

9 Provide an overview of YACC (Yet Another Compiler with labelled diagram in the context of compiler design, 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.

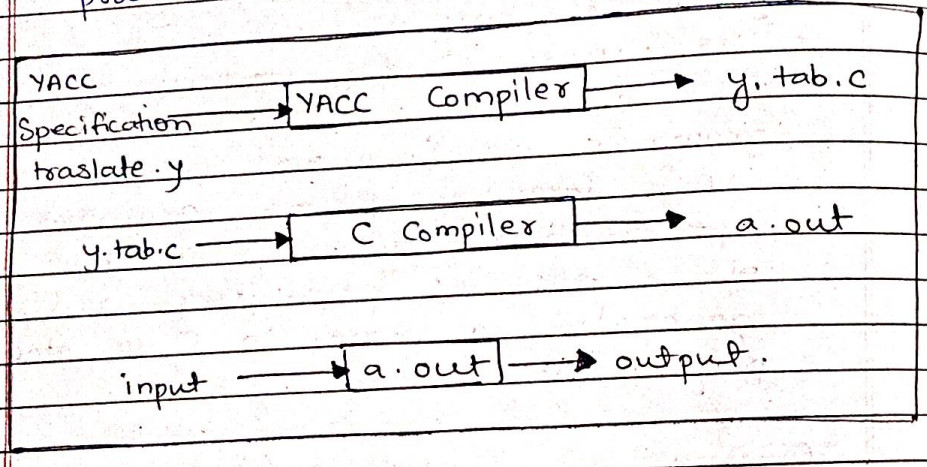
Overview :-

1. Grammar Specification
- First step in using YACC is to define the grammar of the language to be parsed
- Typically specified using Backus-Naur Form (BNF) or Extended BNF (EBNF)

2. YACC Input File
- The grammar specification is written in a file with a '.y' extension, serves as input to the YACC tool

4. Parser Integration
- The generated parser code is then integrated into the compiler project.

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



* Purpose
- Automated Parser Generation
- Language Independence
- Efficiency
- Error Handling

* Utilization
→ Compiler Construction
→ Interpreter Development
→ Language Processing Tools
→ Educational Purposes

Automatic Error recovery in YACC

- The tool YACC can generate a parser with the ability to automatically recover from the errors.
- Major non terminals, such as those for program blocks or statements, are identified; and then error productions of the form A → error a are added to the grammar, where a is usually ε.
- When YACC generated parser encounters an error, it finds the top-most state on its stack, whose underlying set of items includes an item of the form A → error.
- ∴ the parser shifts the token error and a reduction to A is immediately possible
- The parser then invokes a semantic action associated with production A → error, and this semantic action takes care of recovering from the error.

Q Short Note on Storage Allocation.

Ans - 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 data objects such as integers and real variables,
- Data structures such as arrays & strings, are represented by several words of machine memory

There are mainly 3 types of Storage Allocation Strategies:

1. Static Allocation → Simple to understand
2. Heap Allocation → Most Flexible
3. Stack Allocation → Allocation of memory at run-time

—x—

Unit V

Q Elaborate Code Optimization. Discuss with suitable examples about m/c independent & machine-dependent optimization.

Ans 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 m/c code will result.

*Compiler optimization process should meet the following objectives:
> It must be correct, it must not, in any way, change the meaning of the program
> Should increase the speed & performance of the program
> The compilation time must be reasonable

Machine dependent C.O	Machine Independent C.O
- Refers to optimizing code specifically for a particular type of computer or hardware	- Refers to optimizing code for use on any type of hardware.
- Limited to a specific hardware platform	- Applicable to any hardware platform
- Better Performance on specific hardware platform	- Consistent performance across all platforms
- Not useable in any other platform	- Easily reusable in any other platform

* Example

Machine Independent Optimization : (MIO)

Ex: Consider the following expression. (Constant Folding)

int result = 10 * 5 + 3

During MIO, the compiler can evaluate the constant expressions at compile time. In this case the expression 10 * 5 calculated to 50 followed by 50 + 3 —" —— 53. So the optimized code would be

int result = 53

Machine dependent Optimization : (MDO)

Ex: Instruction selection

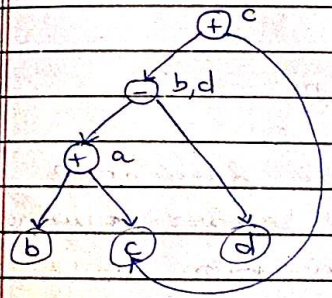
If the source code contains a loop that increments a variable, the compiler might select a single machine instruction that efficiently increments the variable instead of generating multiple instructions to achieve the same result.

Q Ans DAG (Directed Acyclic Graph)

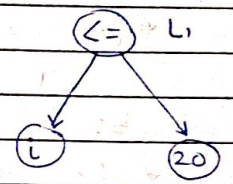
- A DAG is a finite directed graph with no directed cycles.
- In other words, you cannot follow a sequence of edges to return to a vertex you've already visited.
- DAGs have vertices (nodes) connected by directed edges, and each edge has a direction from one vertex to another.

Example | x = y OP z |

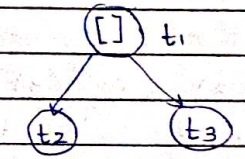
① a = b + c
b = a - d
c = b + c
d = a - d



② if (i <= 20) goto L1



③ t1 = t2 [t3]



* Applications
① Data Flow analysis
② Dependency Resolution
③ Task Scheduling
④ Circuit Design
⑤ Genetic & Evolutionary Algorithms

* Role of DAG in Code Generation
① Expression Representation
② optimizations
   (a) Common Subexpression Elimination (CSE)
   (b) Dead Code Elemenation
   (c) Constant Folding & Propagation
   (d) Register Allocation
③ Instruction selection. ④ Loop Optimization

Q8. Basic Block

Ans 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.

Let us see simple example :-

3 Address Code

(1) f = 1;
(2) i = 2
(3). if i <= x goto 8
(4) f = f * i
(5) t_1 = i + 1
(6) i = t_1
(7) goto (3)
(8) goto calling program

Step① First Identify the leader statements
(1), (3), (8), (4) → Are leader statements
Following is the explination why.

(1) → first statement
(3) → Target goto
(4) → immediately follows a conditional statement
(8) → because it is a target of conditional goto statement

Step② Building Blocks

Block B1 :

f = 1
i = 2

Block B2 :

if i <= x goto 8

Block B3 :

f = f * i
t_1 = i + 1
i = t_1
goto (3)

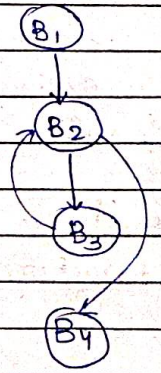
Block B4 :

goto calling program

Note :

1. Blocks :- Include all the statement starting from leader & up to, but not including the next leader
2. No of leader = No of Blocks
   → Blocks ≈ Nodes
3. if x < 3 goto 100 → Conditional goto
   goto 800 → Unconditional goto
4. Edges keep an eye on the Blocks like which block comes after which Block

⇒



B2 Follows B1
B3 Follows B2
B4 Follows B2
B2 Follows B3

8 Peephole

- Simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replaceing these instructions by a shorter or faster sequence, whenever possible.

Characteristics of Peephole optimization:
- → Redundant- instructions elemination
- → Flow- of - control optimizations
- → Algebric simplifications
- → Use of machine idioms
- → Unreachable

* Redundant Loads & stores

(1) MOV R0, a
(2) MOV a, R0

we can delete instruction (2) because whenever ~~whether~~ (2) is executed, (1) will ensure that the value of a is already in register R0

* Strength Reduction

// Before strength Reduction // After S-R

```
int a = 2                int a = 2
int b = a*3              int b = a+a+a
```

* Unreachable Code

Ex:-

```
define debug 1
if (debug)
{
    print debugging inf
}
if debug = 1 goto L1 goto L2
L1 : print debugging inf L2 : ....
```

optimized code

→
```
if debug ≠ 1 goto L2
Print debugging inf
L2 : ------ (b)
```

* Flow of Control

goto L1 goto L2 ⟹ L1: goto L2 L1: goto L2

* Algebric Simplification

x = x + 0 or x = x * 1

This can be easily eliminated using peephole optimization.

* Use Machine Idioms

i = i + 1 → i++
i = i - 1 → i--

- Peephole optimization is type of code Optimization performed on a small part of the code
- The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

Objective
- To improve performance
- To reduce memory footprint
- To reduce code size.

Goals
- Enhance Code Quality
- Reduce Execution time
- Optimize Resource Usage.

—x—

④ Optimization Technique
- Heuristic ordering strategies aim to prioritize nodes that are likely to yield the most significant optimization benefits, thereby improving the overall quality & efficiency of the generated code

UNIT VI

Q Heuristic ordering in the context of DAG.

Ans
- In context of DAGs and code generation heuristic ordering refers to a strategy or algorithm used to determine the order in which nodes should be processed or evaluated
- Heuristic ordering is essential for optimizing code generation & improving the efficiency of generated code.

Here's How heuristic ordering related to code generation in context to DAGs:-

① Expression evaluation
- Expressions represented by DAGs involve multiple nodes representing operations such as +, x, and function calls.
- Heuristic ordering determines the sequence in which these operations should be performed.

② Instruction Selection
- By analyzing the structure of DAG and considering factors such as instruction latencies resource constraints, and target architecture characteristics, a suitable order for generating machine instructions can be determined.

③ Register Allocation
- When performing register allocation, heuristic ordering helps determine the order in which variables are assigned to register

Q Labelling algorithm

Ans
- Labelling 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.
- Labelling algorithm works in bottom-up fashion.
- We will start labeling firstly child nodes & then interior nodes.

Advantages
- Efficient code generation
- Flexibility
- Easy to implement

Disadvantages
- Limited optimization
- Increased code size

Rules

→ if 'n' is a leaf node
- If n is left child its value is 1
- If n is right child its value is 0

→ if 'n' is interior node

Assume L_1 & L_2 is left & right child of interior node

If $L_1 == L_2$ then value of $n = L_1 + 1$ or $(L_2 + 1)$

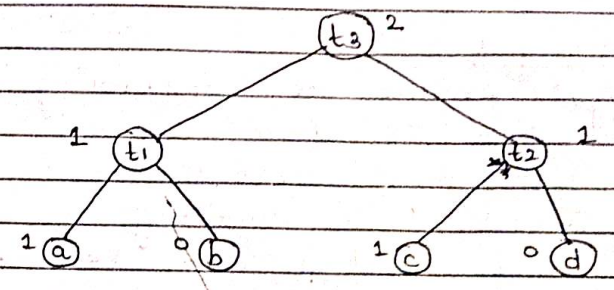
If $L_1 != L_2$ then value of n is $MAX(L_1, L_2)$

Example

$t_1 = a + b$

$t_2 = c + d$

$t_3 = t_1 + t_2$



Q Simple Code Generator.

Ans A simple code Generator is a component of a compiler responsible for translating intermediate representations (such as Abstract Syntax Trees or 3-Address code) into executable m/c code or assembly language instructions.

Overview of how it works :-

1. Intermediate Representation (IR) Parsing :
- The code receives (IR) of program generated by Front-end of the compiler

2. Traversal : The code generator traverse the (IR), using depth-first or top-down traversal algorithm, to visit each node and generate corresponding code.

3. Code Generation :
For each node m/c code sequence is generated.

4. Output Generation:

The generated code is written to an output file or memory buffer, ready for further processing or execution

Challenges

- Target Architecture Compatibility
- Register Allocation
- Instruction Selection
- Control Flow Handling
- Error Handling and Recovery

* Best Practices for designing & implementing a simple code generator (steps)

Understand Target Architecture

↓

modular Design

↓

Use an Intermediate Representation (IR)

↓

Traversal Algorithm

↓

Code Generation Rules

↓

Optimization Techniques

↓

Test & Validation

↓

Documentation & Comments

Continous Improvement

↑

Performance Profiling