📘 Compiler Design – UNIT IV

🔥 Code Generation & Intermediate Code Generation

🎯 2 Days Before Exam Study Notes (RGPV Exam Oriented)

🌟 UNIT IV SYLLABUS

📌 Intermediate Code Generation

- Declarations
- Assignment Statements
- Boolean Expressions
- Case Statements
- Backpatching
- Procedure Calls

📌 Code Generation

- Design Issues of Code Generator
- Basic Block
- Flow Graph
- Register Allocation & Assignment
- DAG Representation
- Peephole Optimization
- Code Generation from DAG

🔥 MOST IMPORTANT TOPICS

Topic	Importance
Intermediate Code Generation	⭐⭐⭐⭐⭐
Backpatching	⭐⭐⭐⭐⭐
Basic Block & Flow Graph	⭐⭐⭐⭐⭐
DAG Representation	⭐⭐⭐⭐⭐
Peephole Optimization	⭐⭐⭐⭐
Register Allocation	⭐⭐⭐⭐
Boolean Expressions	⭐⭐⭐⭐
Procedure Calls	⭐⭐⭐

🔥 1. INTERMEDIATE CODE GENERATION

📌 Definition

Intermediate Code Generation is the compiler phase in which:

```
Machine Independent Code
```

is generated.

This code lies between:

```
Source Code
        &
```

```
Machine Code
```

🌟 Advantages

✅ Machine independent
✅ Easy optimization
✅ Simplifies code generation

🌟 Example

Expression:

```
a = b + c
```

Intermediate Code:

```
t1 = b + c
a = t1
```

📌 Types of Intermediate Code

Type	Example
Three Address Code	t1 = a+b
Quadruples	(+,a,b,t1)
Triples	(0) + a b

📌 Conclusion

Intermediate Code simplifies translation from source program to machine code.

🔥 2. DECLARATIONS

📌 Definition

Declarations specify:

- Variable name
- Data type
- Memory requirement

🌟 Example

```
int a;
float b;
```

🌟 Purpose

✅ Memory allocation
✅ Type checking
✅ Symbol table entry

📌 Conclusion

Declarations help compiler identify variables and their properties.

🔥 3. ASSIGNMENT STATEMENTS

📌 Definition

Assignment statement assigns value to a variable.

🌟 Example

```
a = b + c;
```

🌟 Intermediate Code

```
t1 = b + c
a = t1
```

🌟 Advantages

✅ Simplifies evaluation

✅ Easy optimization

📌 Conclusion

Assignment statements are converted into intermediate representation for further processing.

🔥 4. BOOLEAN EXPRESSIONS

📌 Definition

Boolean Expressions produce:

```
True or False
```

🌟 Example

```
if(a>b)
```

🌟 Relational Operators

Operator	Meaning
>	Greater than
<	Less than
==	Equal to

🌟 Intermediate Code Example

```
if a>b goto L1
goto L2
```

📌 Conclusion

Boolean expressions are mainly used in conditional statements and loops.

🔥 5. CASE STATEMENTS

📌 Definition

Case Statement selects one option from multiple choices.

🌟 Example

```
switch(x)
{
 case 1:
 break;
}
```

🌟 Intermediate Code

```
if x==1 goto L1
if x==2 goto L2
```

🌟 Advantages

✅ Multiple selection

✅ Better readability

📌 Conclusion

Case statements simplify multi-way decision making.

🔥 6. BACKPATCHING 🔥 VERY IMPORTANT

📌 Definition

Backpatching is a technique used to fill incomplete jump addresses later.

🌟 Need of Backpatching

During intermediate code generation:

```
Target labels may not be known immediately
```

🌟 Example

```
goto ___
```

Later replaced by:

```
goto L1
```

🌟 Advantages

✅ Simplifies code generation
✅ Useful in loops and conditionals

📌 Conclusion

Backpatching helps resolve unknown jump targets during compilation.

🔥 7. PROCEDURE CALLS

📌 Definition

Procedure Call transfers control to another function or procedure.

🌟 Example

```
sum(a,b);
```

🌟 Activities During Procedure Call

✅ Parameter passing
✅ Return address storage
✅ Local variable allocation

📌 Conclusion

Procedure calls support modular programming.

🔥 8. CODE GENERATION

📌 Definition

Code Generation is the final compiler phase that converts intermediate code into machine code.

🌟 Input and Output

Input	Output
Intermediate Code	Machine Code

🌟 Goals of Code Generator

✅ Efficient code
✅ Fast execution
✅ Less memory usage

📌 Conclusion

Code Generation produces executable machine instructions.

🔥 9. ISSUES IN DESIGN OF CODE GENERATOR

📌 Important Issues

Issue	Description
Instruction Selection	Choosing proper machine instruction
Register Allocation	Efficient use of registers
Memory Management	Efficient storage usage
Code Quality	Faster execution

🌟 Objectives

✅ Generate efficient code

✅ Minimize execution time

✅ Reduce memory usage

📌 Conclusion

Proper code generator design improves program efficiency.

🔥 10. BASIC BLOCK 🔥 VERY IMPORTANT

📌 Definition

A Basic Block is a sequence of statements with:

```
Single Entry
Single Exit
```

🌟 Example

```
a = b + c
d = a - e
f = d * g
```

🌟 Characteristics

✅ No branching inside block

✅ Executes sequentially

📌 Conclusion

Basic Blocks simplify optimization and flow analysis.

🔥 11. FLOW GRAPH

📌 Definition

Flow Graph represents control flow between basic blocks.

🌟 Diagram

```
B1 → B2 → B3
```

🌟 Uses

✅ Program analysis

✅ Optimization

✅ Detecting loops

📌 Conclusion

Flow Graph shows execution flow between blocks.

🔥 12. REGISTER ALLOCATION & ASSIGNMENT

📌 Definition

Register Allocation assigns variables to CPU registers.

🌟 Example

```
R1 = a+b
```

🌟 Advantages

✅ Faster execution

✅ Reduces memory access

🌟 Register Assignment

Assigning specific registers to variables.

📌 Conclusion

Register Allocation improves execution speed.

🔥 13. DAG REPRESENTATION OF BASIC BLOCK 🔥 VERY IMPORTANT

📌 Definition

DAG (Directed Acyclic Graph) represents expressions in a basic block.

🌟 Example

Expression:

```
a = b+c
d = b+c
```

🌟 DAG

```
   +
  / \
```

```
b   c
```

Both expressions share same node.

🌟 Advantages

✅ Eliminates repeated expressions

✅ Supports optimization

📌 Conclusion

DAG helps identify common subexpressions and optimize code.

🔥 14. PEEPHOLE OPTIMIZATION

📌 Definition

Peephole Optimization examines small sets of instructions and improves them.

🌟 Example

Before:

```
a = a + 0
```

After:

```
a = a
```

🌟 Techniques

✅ Remove redundant instructions

✅ Constant folding

✅ Strength reduction

🌟 Advantages

✅ Faster code

✅ Smaller code size

📌 Conclusion

Peephole Optimization improves local instruction efficiency.

🔥 15. GENERATING CODE FROM DAG

📌 Definition

Machine code is generated from DAG representation.

🌟 Steps

1. Construct DAG
2. Identify common expressions
3. Generate optimized code

🌟 Advantages

✅ Eliminates redundancy
✅ Produces efficient code

📌 Conclusion

DAG-based code generation produces optimized machine code.

🔥 MOST IMPORTANT 14-MARK QUESTIONS

⭐ Very Very Important

1. Explain Intermediate Code Generation.
2. Explain Backpatching with example.
3. Explain Basic Block and Flow Graph.
4. Explain DAG Representation of Basic Block.
5. Explain Peephole Optimization.
6. Explain Register Allocation and Assignment.
7. Explain Boolean Expressions in Intermediate Code Generation.
8. Explain Issues in Design of Code Generator.
9. Explain Code Generation from DAG.

⚡ SMART 2-DAY STUDY PLAN

🌟 DAY 1

MUST STUDY FIRST

✅ Intermediate Code Generation
✅ Backpatching
✅ Basic Block
✅ DAG Representation

THEN STUDY

✅ Boolean Expressions
✅ Assignment Statements
✅ Flow Graph

🌟 DAY 2

MUST STUDY FIRST

✅ Peephole Optimization
✅ Register Allocation
✅ Code Generator Design Issues

LAST REVISION

✅ Procedure Calls
✅ Case Statements

✅ Generating Code from DAG

⚡ ONE NIGHT REVISION

✔ Intermediate code is machine independent
✔ Three Address Code is important
✔ Backpatching fills jump addresses later
✔ Basic Block = Single Entry + Single Exit
✔ Flow Graph shows control flow
✔ DAG removes repeated expressions
✔ Register allocation improves speed
✔ Peephole optimization removes redundant instructions
✔ Code generation converts intermediate code to machine code