THIRD ASSESSMENT ASSIGNMENT

MAHI CHHANCHHARIA

220103029 SECTION- B

CS3044- COMPILER DESIGN

1. The concept of backpatching in an intermediate code generator is a technique used to modify the flow of control within the generated intermediate code. It involves creating a list of statement labels (or jump addresses) that need to be filled in later, and then updating these labels once the final control flow structure is determined.

Backpatching Process involves 3 parts:

i) Label Creation - when jump or conditional jump instruction is encountered, a new label is created & added to a list of labels that need to be backpatched.

ii) Label Filling - After control flow is determined, the labels in the list are filled with the appropriate jump targets.

iii) Code Generation - The intermediate code is generated, incorporating the updated labels.

Example: C-code

```
if(a>b) { x=y+z; }
else { x=y-z; }
```

Intermediate code (before backpatching)

```
1. if a > b goto L1
2. goto L2
3 L1: x = y+z
4 goto L3
5 L2: x = y-z
6 L3:
```

P.T.O

L1 & L2 needs to backpatched.
So after backpatching, code is -

```
1 if a > b goto 3
2 goto 5
3 x = y+z
4 goto 6
5 x = y-z
6
```

2. code Optimization is a critical phase in compiler design that aims to improve the quality of the generated machine code. It involves transforming the intermediate code into equivalent code that executle more efficiently in terms of speed or memory usage.

Key role is in -

i) Reduce the execution time of generated code
ii) Minimize memory footprint of the program
iii) Decrease energy consumption of the program

2 common optimization techniques:-

i) Machine-dependent-optimizations: Targets machine architecture. eg:- pipeline, cache etc.

ii) Machine -independent-optimizations: Focuses on improving the code structure & reducing redundancy.

Peephole optimization is a simple but effective technique that examines a small window of instructions (3-5) at a time & replaces them with equivalent sequences that are more efficient. It is often performed as a final optimization pass after other global optimization. ~~Example :~~ Redundant inst^n elimination, flow-of-control-opt^n, algebric simpli., machine idioms are characteristics of peephole opt^n.

Algorithm for Peephole Optimization :
Step 1 – Select a window (peephole)
Step 2 – Analyze the window & replace the inefficient sequence
Step 3 – Repeat the process by shifting peephole

Example :

Assembly code

```
MOV  R0, A
ADD  R0, #0
MOV  A, R0
```

peephole optimized will eliminate ADD R0, #0 ∵ it is redundant.

Optimized code

```
MOV  R0, A
MOV  A, R0
```

3. Code generator is a final phase of a compiler that takes optimized intermediate code and converts it into target machine code (assembly or machine language). It produces efficient & correct code while preserving the semantics of the source program. It also ensures that $inst^n$ are suitable for architecture of target machine.

DESIGN ISSUES of a CODE GENERATOR :-

i) Target Machine Architecture
Instruction Set : code generator must understand the available $inst^n$ & their semantics for target processors.
Addressing Modes : It needs to select appropriate addressing modes to access data efficiently.
Registers : The allocation & use of registers are crucial for efficient code generation.

ii) Intermediate Code Representation (Input to code generator)
3-Address Code - common representation that consists of inst^{n} with at most 3 operands
Quadruples & Triples. The design must handle different types of these formats ensuring translation is correct.

iii) Instruction Selection, Scheduling & Register Allocation.

Appropriate machine inst^{n} must be selected for the operations in intermediate code. Design should consider: optimal inst^{n} sets, complexity of these sets (RISC VS CISC) & availability of specialized inst^{n}. Determining evaluation order & deciding which values to store in register is also considered.

~~iv)~~
iv) Code Generation Techniques which require design to be complex.

Table driven code generation (using table to map intermediate code to machine code).
Pattern Matching and template based code genration requires a complex code generator design.

v) Barriers and benchmarks requirements in code Quality is high, therefore, design is really difficult. Since codegenerator design should ~~at~~ keep efficiency, correctness and readability under ~~cor~~ considerations.

---

4. A symbol table is a data structure used by a compiler to store information about identifiers (variables, func^{n}, procedures etc) encountered during the lexical analysis and parsing phases.
This information is essential for subsequent phases like semantic analysis, code generation & type checking. Information stored are - identifier name, type (int, float etc), scope,

attributes and location.

Data structures used for implementing the symbol Table :-

i) Linear list (Linked List / Array)

ii) Hash table

iii) BST (Binary search tree)

iv) AVL Tree (Self-Balancing BST)

v) Trie (Prefix tree)

Key actions on symbol table are - insertion, lookup, deletion.

Data str.	Insertion	Lookup	Deletion.
Linearlist	O(1)	O(n)	O(n)
Hash table	O(1)	O(1)	O(1)
BST	O(log n)	O(log n)	O(log n)
AVL	O(log n)	O(log n)	O(log n)
Trie	O(k)	O(k)	O(k).

5. a. if a < b then x = y+z else p = q+r.

(optimised one)

```
1. if a < b goto(5)
2. t1 = q+r
3. p = t1
4. goto(7)
5. t2 = y+z
6. x = t2
7.
```

OR.

```
1. if a < b goto(3)
2. goto(6)
3. t1 = y+z
4. x = t1
5. goto(8)
6. t2 = q+r
7. p = t2
8.
```

P.T.O

b. for (i=1 ; i <= 20 ; i++)
   x = y+z

(optimized code) OR.

```
1.   i = 1
2.   if i > 20 goto (7)
3.   t1 = y+z
4.   x = t1
5.   i = i+1
6.   goto (2)
7.
```

c. while (A<C and B >D) do
   if A=1 then C = C+1
   else
   while A <= D
    do A = A+B.

```
1.  if A< C  goto (3)
2.  goto (15)
3.  if B>D  goto (5)
4.  goto (15)
5.  if  A=1  goto(7)
6.  goto (10)
7.  t1 = C+1
8.  C = t1
```

```
9.  goto (1)
10.   if A <= D goto (12)
11.  goto(1)
12.   t2 = A+B
13.   A = t2
14.   goto(10)
15.
```