Page No.
Date 26 9 24

Open-Book Test

Que-1) a) State the role of syntax directed translation in compiler.

b) How do inherited and synthesized output facilated the process of translation.

→

Que-2) Given the following CFG for arithmatic expression.

$E \rightarrow E + E \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$.

~~Que-3~~ a) Design the syntax directed translation scheme to compute the value of arithmetic expression.

b) Draw annotated parse tree for expression $id_1 + id_2 * id_3$

Que-4) a) Discuss the advantage of using intermediate code in compilation.

b) Compare and contrast three different intermediate code representation.-

- Three address code.
- Quadruples.
- Triples.

Ques 3/4) Write intermediate code in 3-address format for following program format:-

```
if (a+b*c>0)
  {
      x=a+b;
  } else { x=d-c; }
```

Ques 5) a) Explain peephole optimization in code generation phase.

b) What types of optimization are commonly performed in peephole optimization and how do they improve final code.

Ques 6) Apply common subexpression elimination and loop invariant code motion optimize the following code segment:-

```
for (i=0; i<n; i++)
  { x=a+b;
    y= 2*a+b;
    z= x*y;
  }
```

Ques 7) a) Describe the process of register allocation in code generation.

b) Discuss two common algorithm for register allocation such as graph coloring and greedy approach and compare them.

Que - 8 given the following expression
$x = (a+b) * (a-b) + c$

1) translate it into post-fix notation using syntax directed translation
2) generate intermediate code using three address code.
3) perform code optimization by eliminating redundant computation and applying strength reduction were possible.

Answers:-

Ans - 1) -Syntax-directed translation is a key mechanism in compiler design that associates a set of grammar rules with semantic actions.

- This approach helps in translating source code into an intermediate representation or target code by embedding semantic rules directly into the syntax of the language

- Role of syntax-directed Translation:-
1) Grammar-Action association: Syntax-directed translation links syntax rules with semantic actions. Each production in the grammar can have associated

actions that are executed when that production is used in parsing

2) Semantic Analysis : it helps in performing semantic checks during parsing, ensuring that the code adheres to the language's rule (e.g. type checking)

3) Intermediate code generation :- it can produce intermediate representations (IR) directly from the syntax rules, which can later be optimized and translated to target code

4) Error handling :- it allows for the integration of semantic error checking, facilitating better error messages related to context.

* Inherited and Synthesized attributes :-

- Synthesized Attributes :
  - These are attributes that are computed from the attributes of child nodes in a parse tree.
  - They facilitate bottom-up processing, where the semantic information flows from leaves to the root.

- Inherited attributes:
  - These are attributes that are passed down from parent nodes to child nodes.
  - They support top-down processing, allowing children to access information defined in their parent.

* how they facilitate translation:-

1) efficient information flow:- Synthesized attributes ensure that information about the computation can be propagated up the tree efficiently, allowing for the gradual building of complex structure.

2) context awareness:- inherited attributes provide child nodes with contextual information, ensuring correct type checks and maintaining the program's semantic during parsing.

3) modularity:- using synthesized and inherited attributes makes the compiler modular, allowing for easy addition or modification of semantic actions without altering the grammar structure.

4) facilitating optimizations:- by gathering semantic information during parsing, the compiler can optimize code based on context and types, leading to more efficient generated code.

* example :-

$E \rightarrow E1 + E2$

$E \rightarrow E1 - E2$

$E \rightarrow E1 * E2$

$E \rightarrow E1 / E2$

$E \rightarrow num$

- attributes :-
  - Synthesis attribute : E.val
  - Inherited attribute : E1.type , E2.type.
- Translation steps :-
  1) for $E \rightarrow num$:
     - Action : set E.val = num.value
     - Type : E.type = int :
  2) for $E \rightarrow E1 + E2$:
     - Action : check types : if E1.type and E2.type are both int.
     - set E.val = E1.val + E2.val.
     - type : E.type = int.
  3) for $E \rightarrow E1 - E2$
     - Action : E.val = E1.val - E2.val
     - Type : E.type = int.

Ans-2) To design a syntax-directed translation scheme for the given context-free grammar (CFG) for arithmetic expressions, we will include synthesized attributes to compute the value of the expression as we parse it.

- CFG :-

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

- Syntax directed translation scheme.

Attributes

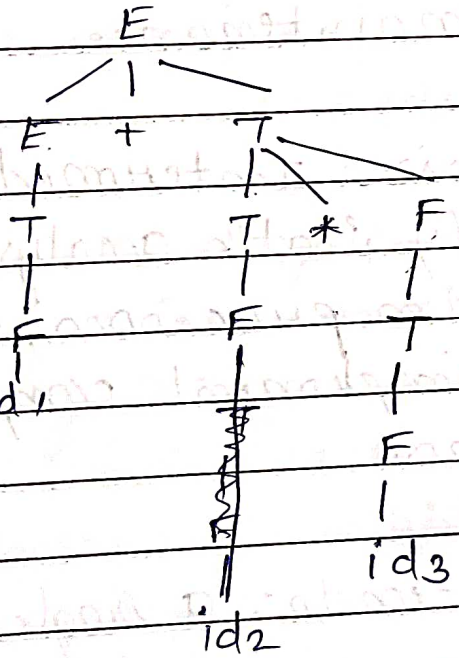
- E.val : the value of the expression
- T.val : the value of the term.
- F.val : The value of the factor.
- id.val : The value associated with the identifier.

Semantic Actions:

- for $E \rightarrow E + T$:-
  - Action : E.val = E1.val + T.val

- 2) for $E \rightarrow T$:
  - Action : $E.val = T.val$
- 3) for $T \rightarrow T * F$:
  - Action : $T.val = T.val * F.val$
- 4) for $T \rightarrow F$:
  - Action : $T.val = F.val$
- 5) for $F \rightarrow (E)$:
  - Action : $F.val = E.val$
- 6) for $F \rightarrow id$:
  - Action : $F.val = id.val$ (where $id.val$ is fetched from a symbol table)

- Annotated parse tree for the expression: $id_1 + id_2 * id_3$:-



Ans-3] Advantages of using intermediate code in compilation:-

- platform independence: intermediate code abstracts away machine-specific details, making it easier to generate code for different architecture.

- optimization opportunities: it provides a stage where various optimizations can be applied without being tied to the source or target languages.

- Simplified compiler structure: compiling in stages allows for modular development and easier maintenance.

- Ease of Analysis: Intermediate code can simplify static analysis and transformation processes, making it easier to implement compiler optimizations.

- multiple back-ends: a single intermediate representation can support code generation for different target architectures, reducing redundancy.

- Support for debugging : it can carry additional information useful for debugging and profiling, aiding developers in performance analysis.

* Comparing different intermediate code representation :-

1) Three - address code
   - description : This representation uses instructions that have at most three addresses or operands. Typically, it follows the form $x = y$ op z, where op is a binary operator
   - example :

$$t_1 = id_1 + id_2$$

$$t_2 = t_1 * id_3$$

   - advantages :-
     - Simple and straightforward, making it easy to generate machine code.
     - facilitates optimization by allowing temporary variables to store intermediate results.
   - disadvantage :
     - more verbose than other forms, as it uses temporary variables explicitly.

2) quadruples:

- description: This representation organizes code into tuples of four fields, usually formatted as (op, arg1, arg2, result).
- example:
  (+, id1, id2, t1)
  (*, t1, id3, t2)
- advantages:
  - clearly separates operations and their operands, making it easy to understand and manipulate.
  - facilitates the application of optimization and code generation.
- disadvantages:
  - required more space than three-address code due to the explicit representation of each operand and result.

3) Triples:

- description: Similar to quadruples, but it uses three fields, omitting the result field. instead, results are represented by an index.
- example:
  (+, id1, id2) → t1
  (*, t1, id3) → t2

- Advantages:
  - more compact than quadruples, as it eliminates the need for an explicit result field.
  - still retains a clear structure for understanding the operation.

- disadvantages:
  - slightly less intuitive, as users must interpret indices to determine results
  - can complicate optimizations and code generation, as tracking results can become cumbersome.

Ans-4) given: if (a+b*c) { x = a+b; }
else { x = d-c; }

* Three - address code representations:

1) evaluate the expression: Start by calculating b*c and a+(result).
2) conditional check: We will use a label to represent the branching.
3) Assignments: Assigns values to x based on the conditional result.

→ the intermediate code in three-address form:—

```
t1 = b * c    // step 1: Calculate b*c
t2 = a + t1   // step 2: Calculate a+(b*c)
if t2 != 0 goto L1  // step 3: if(a+b*c) is true, go to L1
t3 = d - c    // step 4: else part: Calculate d-c
x = t3        // step 5: Assign result to x
goto L2       // step 6: Jump to end.

L1:           // label for if part
t4 = a + b    // step 7: Calculate a+b
x = t4        // step 8: Assign result to x
L2:           // end label
```

Qu-5> Ans- Peephole optimization is a local optimization technique used during the code generation phase of a compiler. it involves examining a small window of consecutive instructions in the generated code and replacing them with more efficient instructions or patterns. This approach is often applied after the initial code generation to improve performance without needing a complete re-evaluation of the

entire code structure.

* Types of optimizations performed in peephole optimization :-

1) Redundant instruction elimination:
- example : removing unnecessary instructions that do not affect the programs outcome.
- improvement : reduces the number of instructions, leading to faster execution and smaller code size.

2) constant folding :
- example : evaluating constant expressions at compile time rather than runtime.
- for instance, converting.
x = 3+4 to x = 7
Before : MOV R1, 3+4
After : MOV R1, 8
- improvement : reduces the need of runtime computation, improving execution speed.

3) Strength reduction:
Description : replaces expensive operations with equivalent but cheaper operation
- example: Before : MUL R1, R2, 4
After : SHL R1, R2, 2
- improvement : Reduce code size and unwanted computation

4) Algebraic simplifications:

Descriptions: Simplifies arithmetic expressions using algebraic identities, such as eliminating operations that have no effect or simplifying expressions involving constants.

example: Before: ADD R1, R1, 0

After: (No operation, since adding 0 does not change the value)

improvement: minimizes unnecessary operations, which improves both code size and execution time.

5) Jump optimization:

Description: Simplifies or removes unnecessary jump instruction, such as jumps that immediately lead to another jump to the next sequential instruction.

example: Before: JMP L1

L1: JMP L2

After: JMP L2

improvement: reduces instruction count, minimizing control flow complexity and improving runtime performance.

- How these optimization improve code is as follows:-

1) execution speed: peephole optimization reduces instruction count and replaces expensive operations with cheaper ones, speeding up program execution.

2) Code Size: it eliminates redundant & unnecessary code, reducing the overall size, which is useful in memory-limited systems

3) Resource utilization: by minimizing unnecessary memory access and calculations, it optimizes CPU and memory usage, enhancing overall efficiency.

Qu-6) Ans:- given:- for (i=0; i<n; i++) {

```
x = a+b;
y = 2 * a+b;
z = x*y;
```

- optimized code with common subexpression elimination and loop invariant code motion:

- common subexpression elimination:
  - a+b is a common subexpression in both x and y. we compute it once and reuse it

- loop invariant code motion:-
  - a+b and 2 * a+b donot depend on the loop variable i. Since they remain constant throughout the loop, they can be moved outside the loop.

optimized code:

```
t1 = a+b;
t2 = 2 * a+b;

for (i=0; i<n; i++){
    x = t1;
    y = t2;
    z = x * y;
}
```

explanation:- t1 = a+b and t2 = 2 * a+b are computed once outside the loop
- inside the loop, x and y reuse these computed values, reducing redundant calculation and improving performance

Ques-7) Ans- Register allocation is the process of assigning a limited number of CPU registers to hold the values of variables or intermediate results during program execution. Since the number of registers is limited, but the number of variables and temporary values can be large, efficient register allocation is crucial for generating optimized code. Poor register allocation can lead to excessive use of slower memory operations when values need to be stored and loaded from memory (spilling), which degrades performance.

* Two common algorithm for register allocation are as follows:-

1) Graph coloring approach:-

- ~~info~~ interference graph: The compiler constructs an interference graph, where:
  - each node represents a variable or temporary value.
  - An edge between two nodes indicates that the two variables interfere.

- Graph coloring: The goal is to assign registers such that no two adjacent nodes

are assigned the same register. This is analogous to coloring a graph where no two adjacent nodes share the same color, with the "colors" representing registers.

- Spilling: if the graph cannot be colored with the available registers some values are "spilled" to memory.

2) Greedy Approach:

process:- the greedy algorithms assigns registers on a first-come, first-served basis. it works by examining instructions in a linear fashion and trying to allocate a register to a variable when it is needed.

- when a register is needed but all registers are in use, the algorithm selects a variable to spill to memory.

- once a register is freed, it is reused for subsequent variables or instructions.

* Comparison :-

Aspect	Graph coloring	Greedy Approach
complexity	Higher (due to graph construction and coloring)	Lower (linear, simple to implement)
efficiency	Generally more efficient in minimizing spills.	more prone to excessive spills.
Global vs local	Global allocation (considers the whole program)	Local allocation (focus on immediate need)
suitability for small code	less ideal (overhead may not be justified)	well-suited (fast and simple)
Suitability for large code	effective for larger, more complex programs	suboptimal for complex, larger programs.
Spill Handling	fewer spills due to global consideration	more spills, handled on-the-fly

Q.11-2) Ans:-

1) Translate the expression into postfix notation using syntax-directed translation.

The given expression is:-

$x = (a+b) * (a-b) + c$

Using postfix notation :-

- first translate (a+b) and (a-b):
  - ab+ and ab-
- then translate the multiplication:-
  - ab+ ab- *
- lastly, add c and perform the final addition:-
  - ~~a+b+~~ ab+ ab- * c+

So, the postfix expression is

ab+ ab- * c+

2) Generate intermediate code using three-Address code:

To generate three-address code (TAC) we break the expression into smaller steps, assigning temporary variables for intermediate results:-

$t_1 = a + b$ (compute $a+b$)
$t_2 = a - b$ (compute $a-b$)
$t_3 = t_1 * t_2$ (multiply t_1 & t_2)
$t_4 = t_3 + c$ (add $t_3 + c$)
$x = t_4$ (assign result to x)

So, the three-address code is:-

$t_1 = a + b$
$t_2 = a - b$
$t_3 = t_1 * t_2$
$t_4 = t_3 + c$
$x = t_4$

3) perform code optimization:-

- Redundant computation elimination:-
  - in the expression $(a+b)$ and $(a-b)$, $a+b$ and $a-b$ are computed only once. There's no redundancy to eliminate in this step.

- strength reduction:-
  - there is no multiplication by powers of 2 so there's no direct opportunity for strength reduction here.

however, we can combine assignments directly where possible to streamline the code:-

$t_1 = a + b$
$t_2 = a - b$
$x = (t_1 * t_2) + c$