1) Explain the concept of syntax trees and how they are used in the intermediate code generation with an example.*

A:- Syntax trees, also known as a parse trees (or abstract syntax trees (AST)), are a hierarchical representation of the syntatic structure of a program's source code. They capture the relationships between the various components of the code such as expressions, statements and declarations, based on the grammar rules of the programming language.

The process of generating a syntax tree involves parsing the source code using a parser, which analyzes the code's tokens and applies grammer rules to construct the tree structure. Each node in the syntax tree represents a syntactic element, such as an operator, identifier, or control structure, the tree's structure reflects the code's organization.

Syntax trees are commonly used in compiler construction and intermediate code generation. Here's an example to illustrate their usage in intermediate code generation:

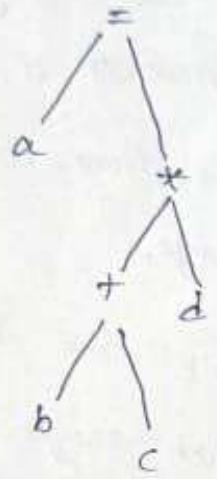
Ex:- a = (b+c)*d

1) Lexical Analysis:- The source code is first tokenized, identifying

individual tokens such as identifiers (a, b, c, d), operators (+, *), and parentheses.

2) parsing :-

The parser analyzes the tokens based on the language's grammar rules to construct a syntax tree. The resulting syntax tree for the above expression would look like this :



Intermediate Code Generation :-

The syntax tree is then used to generate intermediate code. By using syntax trees during intermediate code generation, the compiler can analyze the program's structure, apply optimizations, and transform the code into a more suitable format for subsequent stages.

Syntax tree provide a structured representation of the code's syntax, allowing the compiler to reason about the program's structure, perform semantic analysis, and generate efficient intermediate code for further processing.

learn VLSI fun

Compiler Design

(2) Discuss the translation of expression into three Address code Providing an example

(A) → In three address code, almost three addresses are used to Represent any statement

→ General Form of three Address code

$a = b \ op \ c$

Here a, b, c are operands

op Indicates operators

Eg: $a = b + c$

$c = a * b$

→ Three address code is a type of Intermediate code which is easy to generate and can be easily converted to machine code.

→ It makes use of al most three addresses and one operator to represent an expression and value computed at each Instruction is stored in temparory variable generated by compiler

→ Common Three Address Instruction Forms

① Assignment statements

a) $x = y \ op \ z$ → { op may be +, −, /, * e.t.c; }

b) $x = op \ y$ → { op represents Increment or decrement operator }

② Copy statement

$x = y$ (Operator is =)
(Assignment)

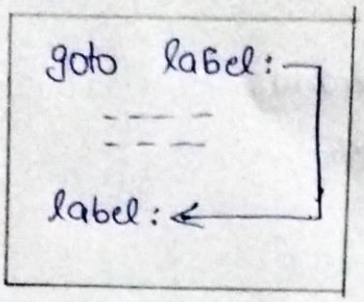
③ Conditional Jump

→ if x relop y goto L

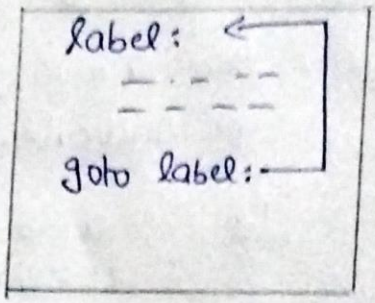
relop indicates relational operators (<, >, <=, >= etc)

goto → It represents transfer to that location specified by Label ('L')

Eg:



fanward goto



Backward goto

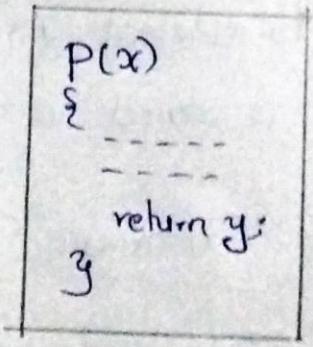
④ Unconditional Jump

goto L

⑤ Procedure call

Param x call P

return y



ξ :: 'x' is a parameter calling 'P' function and returns 'y' }

→ Generation of Three Address Code for expression

Eg: 1 : a = b+c+d

Operands = a, b, c, d

Operators = +, = (+ High Precedence than =)

Steps: ① Perform b+c and store in temporary variable

$T_1 = b + c$

② Add to d and store Result in new Temporary variable

$T_2 = T_1 + d$

③ Now Assign to 'a'

$\boxed{a = T_2}$ so, this is the three Address code generation

Ex-2: $-(a*b) + (c+d) - (a+b+c+d)$

Here it contains unary minus, binary plus, binary minus we already know unary minus Precedence is high

$T_1 = a * b$

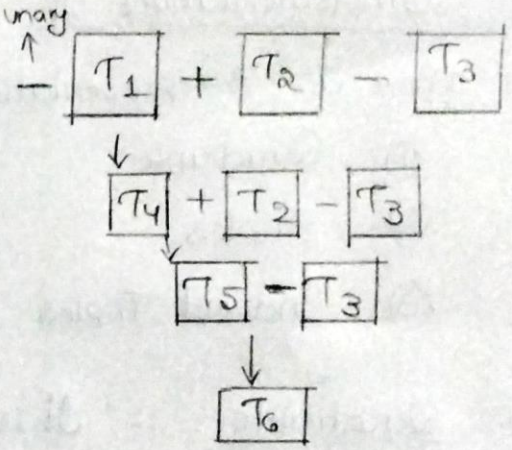
$T_2 = (c+d)$

$T_3 = (a+b+c+d)$

$T_4 = -(a*b)$

$T_5 = T_1 + T_2$

$T_6 = T_5 - T_3$



Ex-3

if A<B
then 1
else 0

Sol

Memory location	Statements
(1)	if (A<B) goto 4
(2)	$T_1 = 0$
(3)	goto (5)
(4)	$T_1 = 1$
(5)	(end)

{ :: if (A<B) is true it goto (4) and displays $T_1 = 1$ If it is not true then display $T_1 = 0$ }

goto (5) indicates end state

Ex-4: if a<b & c<d then t=1, else t=0

Memory location	statements
(1)	if (a<b) goto (3)
(2)	goto (4)
(3)	if (c<d) goto (6)
(4)	T=0
(5)	goto (7)
(6)	T=1
(7)	(end)

→ Implementation of Three Address code

There are 3 representations namely

① Quadruple
② Triples
③ Indirect Triples

→ Quadruple :- It is structure which consists of 4 fields namely OP, arg1 & arg2 and Result.

OP → Operator, arg1, arg2 → Operands

Ex:- consider expression a = b* - c + b* - c

Sol

t1 = Uminus c
t2 = b * t1
t3 = Uminus c
t4 = b * t3
t5 = t2 + t4
a = t5

memory location #	OP	Arg1	Arg2	Result
(0)	Uminus	c		t_1
(1)	*	t1	b	t2
(2)	Uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple Representation

→ **Triples** This Representation doesn't make use of extra temporary variable to represent a single operation

Instead when a Reference to another triples value is needed a pointer to that triple is used.

→ It consists of OP, arg1, arg 2

Ex: $a = b * -c + b * -c$

#	OP	Arg 1	Arg 2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples Representation

→ **Indirect Triples** This Representation makes use of Pointer to the listing of all references to computations which is made Separately and stored

Ex: $a = b * -c + b * -c$

#	OP	arg 1	arg 2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of Pointers to table	
#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triple Representation.

3) Describe the process of translating Boolean expression in control flow statement considering short circuited code flow?

When translating a Boolean expression into a control flow statement, especially in a language that supports short-circuited evaluation, the process involves converting the Boolean expression into a series of conditional statements that reflect the short-circuit behavior. Here's a step-by-step explanation of the process:

1. Identify the Boolean expression:

Start by identifying the Boolean expression that you want to translate into a control flow statement. It typically consists of logical operators (such as AND, OR) and relational operators (such as <, >, ==) combined with variables, constants, or function calls.

2. Evaluate the expression incrementally:

With short-circuit evaluation, the expression is evaluated incrementally from left to right, and the evaluation stops as soon as the final result can be determined. This means that some parts of the expression may not be evaluated if their evaluation is not necessary to determine the final result.

3. Translate each part of the expression:

Break down the Boolean expression into smaller sub-expressions or invidual conditions. consider the short-circuit behavior when determining the order

of evaluation, the logical operators involved in the expression are;

a. AND operator (&&): If the left-hand side of the AND operator evaluates to false, the right-hand side is not evaluated because the overall result will be false regardless of the right-hand side. Translate this behavior using an if statement where the left-hand side is the condition, and if it evaluates to false, the right-hand side is not executed.

b. OR operator (||): If the left-hand side of the OR operator evaluates to true, the right-hand side is not evaluated because the overall result will be true regardless of the right-hand side. Translate this behavior using an if statement where the left-hand side is the condition, and if it evaluates to true, the right-hand side is not executed.

4. Combine the translated conditions:

Combine the translated conditions using appropriate control flow statements (such as if-else, nested if, or switch case) based on the desired logic and structure of your program.

5. Handle the final result: Determine how the final result of the Boolean expression should affect the control flow of your program. For example, if the expression evaluates to true, certain code blocks may be executed, while if it evaluates to false, a different set of code block may be executed.

6. Implement the control flow: write the code that reflects the translation of the Boolean expressio and the desired control flow behaviour, make sure to include any necessary code blocks, loops, or other control flow constructs based on your specific requirements.

It's important to note that the exact implementation may vary depending on the programming language you are using, as different languages may have slightly different syntax or control flow constructs. Additionally, keep in mind any side effects or dependencies within the Boolean expression that may impact the overall behaviour of the translated code.

Issues in the design of a code generator:

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code Optimization and code-generation phases of a compiler, often referred to as the backend, may take multiple passes over the IR before generating the target program.

A code generator has three primary tasks: Instruction selection, register allocation and assignment, and Instruction ordering. Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

Issues that are going to be discussed are:

1) Input to the code generator
2) The target program
3) Instruction selection
4) Register Allocation
5) Evaluation order

Code generator converts the intermediate representation of a source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of the code generator should be done in such a way that it can be easily implemented, tested and maintained.

Input to the code generator: The input to the code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the runtime addresses of the data objects by the names in the intermediate representation! Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's etc; The code generation phase just proceeds on an assumption that the input is free from all syntactic and semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Target program: The target program is the output of the code generator

The output may be absolute machine language, relocatable machine language, or assembly language.

→ Absolute machine language as output has the advantages that it can be placed in a fixed memory location and can be immediately executed. For example, WATFIV is a compiler that produces the absolute machine code as output.

→ Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by a linking loader. But there is

added expense of linking and loading.

- Assembly language as output makes the code generation easier. We can generate symbolic instructions and use the macro-facilities of assembly in generating code. And we need an additional assembly step after code generation!

Instruction selection: Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered. But if we do not care at the efficiency of the target program then instruction selection is straightforward. For example the respective three address statements would be translated into the latter code sequence as shown below:

```
P := Q + R
S := P + T

MOV  Q, R0
ADD  R, R0
MOV  R0, P
MOV  P, R0
ADD  T, R0
MOV  R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code

Sequences with significant cost difference between the different implementations. Prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

Register allocation issues:

Use of registers make the computations faster in comparision to that of memory, so efficient utilisation of registers is important. The use of registers is subdivided into two subproblems:

→ During Register allocation: we select only that sets of variables that will reside in the registers at each point in the program.

→ During a subsequent Register assignment: the specific register is picked to access the variable

To understand the concept consider the following three address code sequence.

```
t := a + b
t := t * c
t := t / d
```

Their efficient machine code sequence is as follows:

```
MOV  a, R0
ADD  b, R0
MUL  c, R0
DIV  d, R0
MOV  R0, t
```

Evaluation order: The code generator decides the order in which the instructions will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

5 Explain the Concept of basic blocks and flow graphs and how they Contribute to Code generation optimization

Sol) The Basic blocks and flow graphs are fundamental Concepts that help optimize Code generation for efficient execution.

→ A basic block is a simple Combination of Statements. the basic blocks donot have any branches like in and out. It means that the flow of control enters at the beginning and it always leaves at the end without any halt. The execution of set of instructions of a basic block always takes place in the form of a Sequence. The first setup is to divide a group of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, the control will flow from one instruction to the next in sequential order.

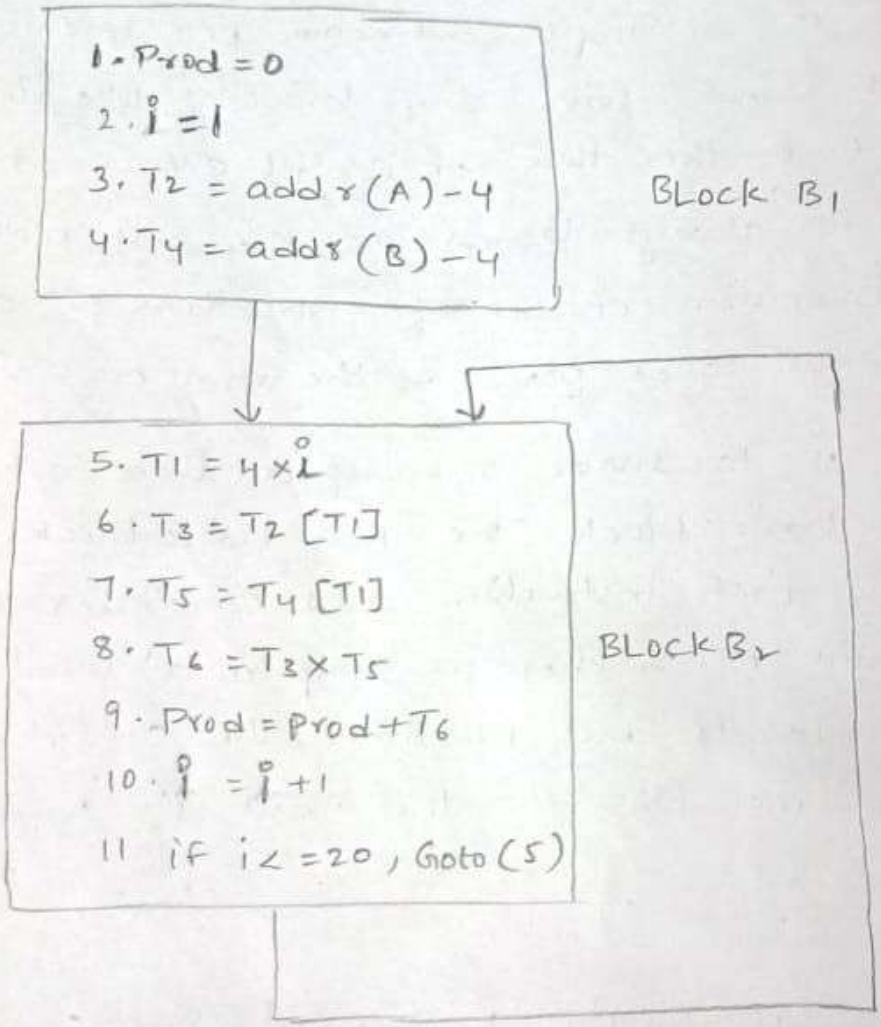
Flow Graph

→ A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of Control information. A Control flow graph is used to depict how the program Control is being passed among the blocks. A flow graph is used to illustrate the flow of Control between basic blocks once an intermediate Code has been partioned into basic blocks

~~once an intermediate Code has been partioned into~~

When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow one block x to another Y

Lets make the flow graph of the example that we used for basic block formation:



firstly, we Compute the basic blocks (which is already done above. Secondly. we assign the flow control information:

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