Unit-3

Syntax directed Translation and Intermediate code generation

SDT (syntax directed translation): SDT is nothing but translation of Abstract syntax tree into Decorated abstract syntax tree

Decorated Abstract syntax tree is adding attributes to the nodes of the syntax tree.

SDD (syntax directed definition):

SDD is a context free grammar along with attributes and Rules. Attributes are associated with grammar symbols and Rules are associated with the productions of the grammar.

Attributes are anything which we want to define about a grammar symbols. For Eg. value, type, length, size, ~~etc~~. string, etc.

Synthesized attributes and Inherited attributes:-

Synthesized attributes:- An attribute at a non-terminal 'A' can be defined by itself or by its childrens.

Inherited attributes:- An attribute at a non-terminal 'A' can be defined by its parent, by itself or by its siblings.

Terminal symbols always have synthesized attribute and non-terminals can have both synthesized and Inherited attributes.

S-attributed definition and L-attributed definition:-

An SDD is said to be S-attributed definition if it contains only synthesized attributes.

For Eg:

	Production	Semantic Rule
1	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
2	$E \rightarrow T$	$E.val = T.val$
3	$T \rightarrow T_1 * f$	$T.val = T_1.val * F.val$
4	$T \rightarrow F$	$T.val = F.val$
5	$F \rightarrow (E)$	$f.val = E.val$
6	$F \rightarrow digit$	$F.val = digit.lexval$

fig:- SDD for simple Desk Calculator.

Above SDD contains only synthesized attributes so it is called as S-attributed definition. (Above table is also an example for SDD).

L- Attributed Definition:-

An SDD is said to be L-attributed definition it contains both

1. Synthesized attribute
2. Inherited attribute with following conditions
   (i) Its value can be calculated by its parent
   (ii) Its value can be calculated by itself.
   (iii) Its value can be calculated by its left side sibilings.

For Eg:-

	Productions	Semantic Rule
1.	$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2.	$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
3.	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4.	$f \rightarrow digit$	$F.val = digit.lexval.$

Fig:- Indicated L-attributed definition

Above SDD contains both synthesized and Inherited attributes so it is called as L-attributed definition.

* Annotated Parse tree :—

A parse tree showing the values of attributes is called as annotated parse tree.

For eg 1:- Annotated parse tree for S-attributed definition for 3*5+4.

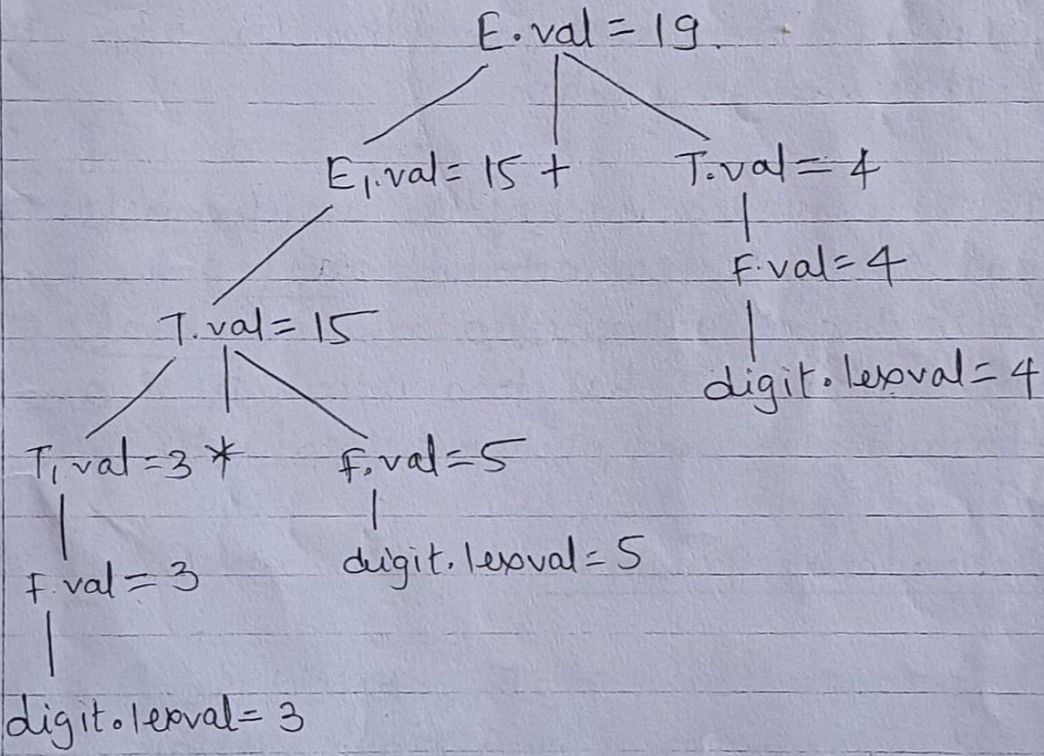


Fig: Annotated parse tree for 3*5+4 using S-attributed definition.

Eg 2:- Annotated parse tree of 3*5 using L-attributed definition.

P.T.O

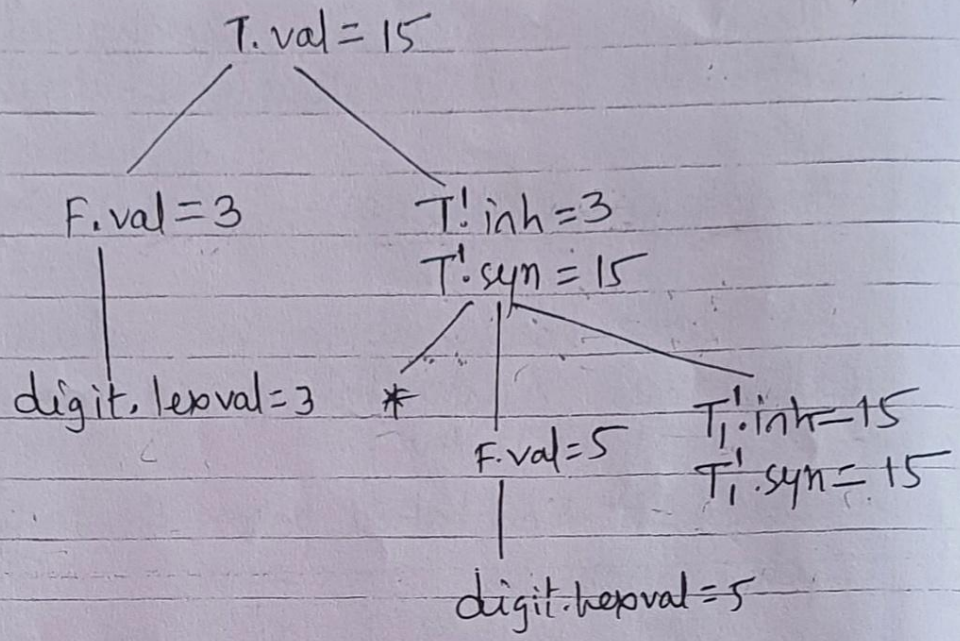


Fig:- Annotated parse tree for 3*5 using L-attributed definition.

In the above eg, each of the non-terminals T and F has a synthesized attribute val; the terminal digit has a synthesized attribute lexval. The non-terminal T' has two attributes: an inherited attribute inh and a synthesized attribute syn

* Dependency graph:—

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree. An edge from one attribute instance to an other means that the value of the first is needed to compute the second.

For eg 1:- Dependency graph for 3*5+4 using S-attributed definition.

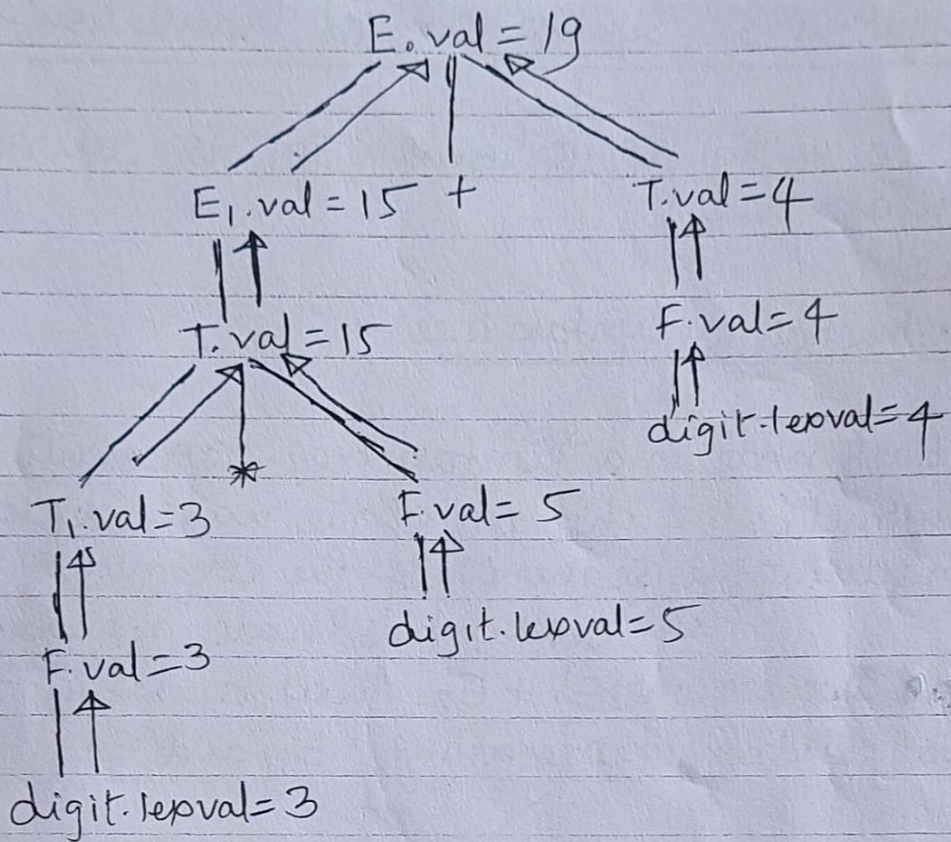


Fig:- Dependency graph for 3*5+4 using S-attributed definition.

In the above graph the dotted line indicates actual parse tree and solid edge line indicates dependency graph.

Eg: 2. Dependency graph for 3*5 using L-attributed def?

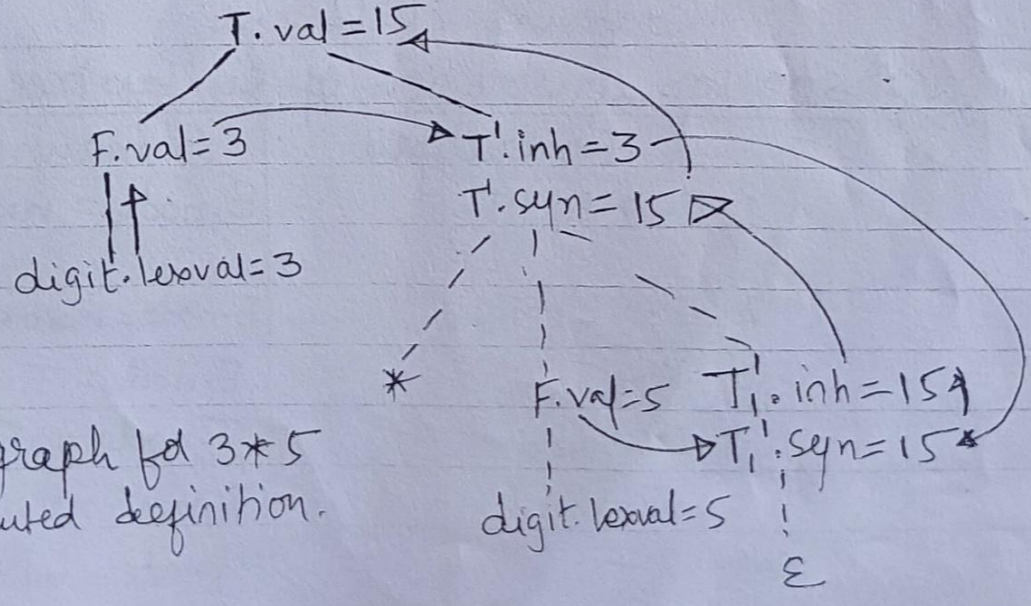


Fig:- Dependency graph for 3*5 using L-attributed definition.

Applications of Syntax-Directed Translation:-

One of the application of SDT is construction of syntax tree.

Construction of syntax tree:-

Each node in a syntax tree represents a construct; the children of the node represents meaningful sequence of the construct.

A syntax tree node representing an expression $E_1 + E_2$ has a label + and two children representing E_1 and E_2.

We shall implement the nodes of a syntax tree by objects with suitable number fields.

Each object will have an op field that is the label of the node. The objects will have other fields as follows.

1. If node is a leaf, an additional field holds the lexical value for the leaf.
2. If the node is interior node, there are as many additional fields in which node contains childrens.

SDD for construction of syntax tree

Productions	Semantic Rule
1. $E \to E_1 + T$	E.node = new Node('+', E_1.node, T.node
2. $E \to E_1 - T$	E.node = new Node('-', E_1.node, T.node
3. $E \to T$	E.node = T.node
4. $T \to (E)$	T.node = E.node
5. $T \to id$	T.node = new Leaf(id, id.entry
6. $T \to num$	T.node = new Leaf(num, num.val

Eg:-

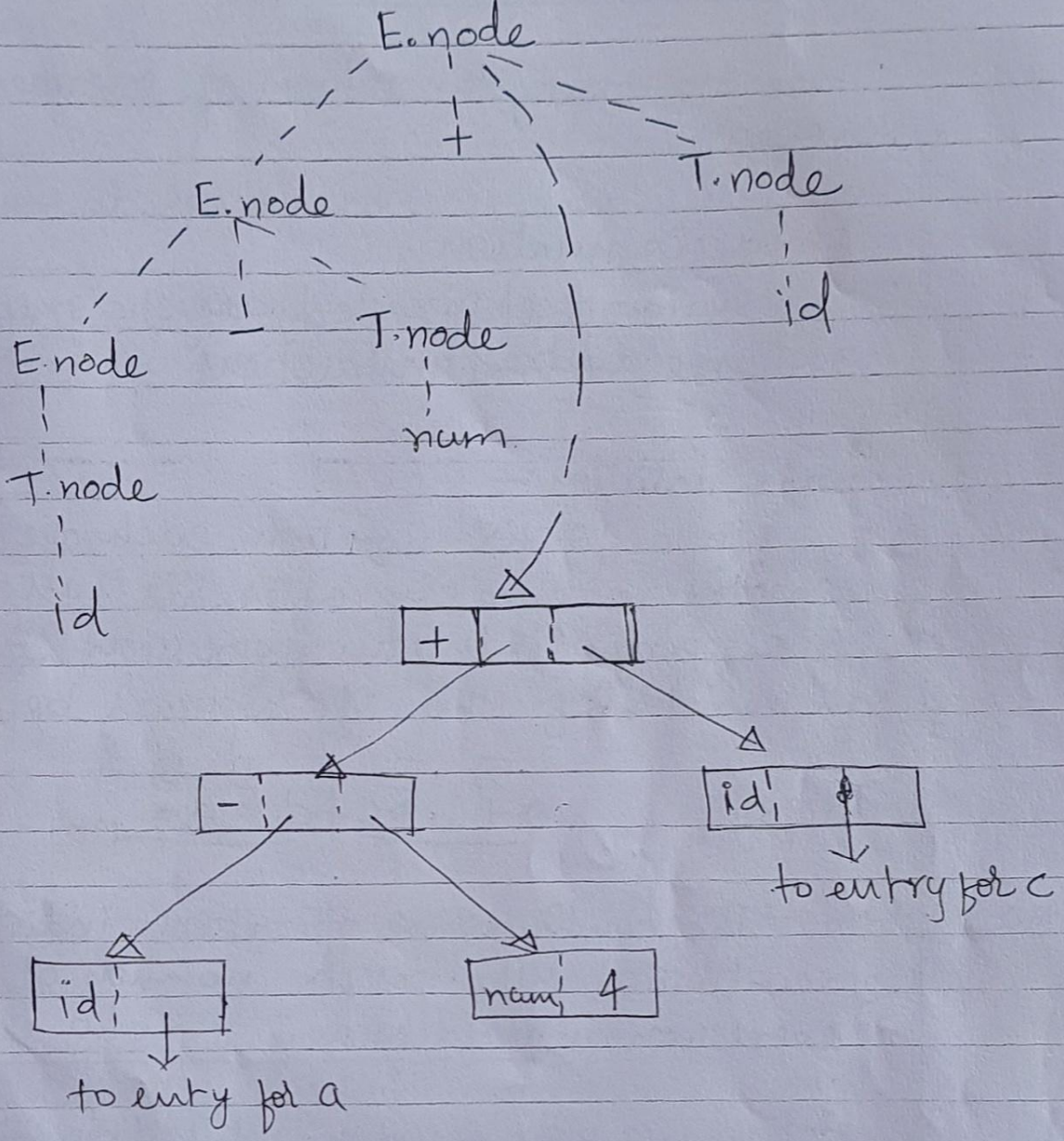


Fig: syntax tree for a-4+c

Date __/__/__
Page

Intermediate languages :-

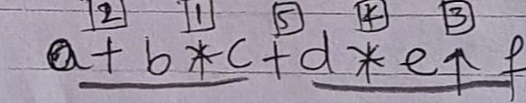
There are mainly three types of intermediate languages

1. Postfix notation
2. Syntax tree / Directed acyclic graph
3. Three address instructions.

1. Postfix notation :-

In a postfix notation each operator 'opi' appears immediately after its last operand. If the operand itself is an expression containing an operator 'opj' then opj appears before opi

For eg :-



The number above operators indicates order of evaluation. The postfix notation of the above expression is as follows.

a + bc * + d * e ↑ f
abc * + + d * e ↑ f
abc * + + d * ef ↑
abc * + + def ↑ *
abc * + def ↑ * + → (postfix notation of the above given expression).

2. Directed Acyclic graph :- DAG (Directed acyclic graph) is a special version of syntax tree). Both represents an syntactical structure of an expression.

i) ↑ * / + -

Some of the differences of DAG with respect to syntax tree are.

(i) DAG is helpful in identifying common subexpression
(ii) DAG is more compact than syntax tree. It means it takes less space than syntax tree.

For Eg: DAG for the expression
a + a * (b - c) + (b - c) * d

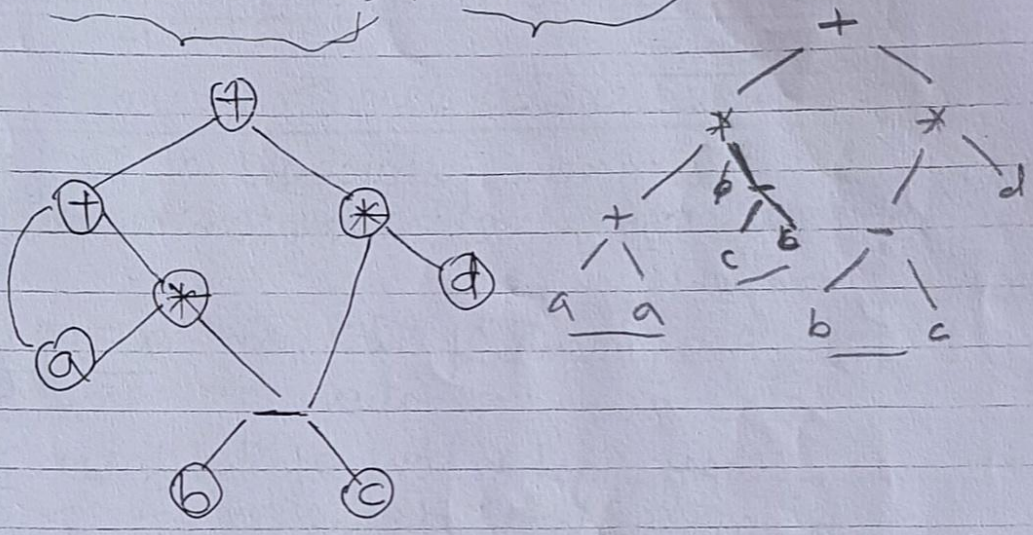


fig:- Dag for the expression a + a * (b-c) + (b-c) * d

3. Three address Instruction:-

Properties of three address instruction.

1. Every three address contains maximum three operands. (Less then three operands is accepted)
2. Right side of three address instruction contains atmost one operator.
3. Compiler may generate some of the temporary variables to hold the partial results.

For Eg:- Z = a + b * C

The above expression is not an three address code as it does not obeys the properties of three address code.

It can be converted into three address code as follows

$t_1 = b * c$
$t_2 = t_1 + a$
$x = t_2$

} obeys properties of three address instructions.

* Some of the list of common three-address instruction forms:-

1. Assignment instructions of the form $x = y$ op z, where op is a binary arithmetic or logical operations.
2. Assignments of the form $x = $ op y, where op is a unary operation.
3. copy instructions of the form $x = y$, where x is assigned the value of y.
4. An unconditional jump goto L. The three address instruction with label 'L' is the next to be executed.
5. conditional jump if x goto L and if False goto L are also considered as three address instruction.
6. conditional jump such as if x relop y goto L is also three address code form.
7. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$ are also considered as three address instruction.
8. Address and pointers assignments of the form $x = \&y$, $x = *y$ and $*x = y$ are also kind of three address instructions.

t5
t2 t4
t1 t3

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

convert above expression into three address code.

Ans:- $t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$.

* Implementation of three address instruction:-

Three address instruction can be implemented using
(i) postfix Triples
(ii) Quadruples.

(i) Triple:- A triple has only three fields.

op	operand1	operand2

note that triple may be operand1 | operand2 or it may be result of some other triple.

For eg:- Triples for $a = b * -c + b * -c$

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

Triples ⟹

	op	op1	op2
1	minus	c	--
2	*	b	(1)
3	minus	c	--
4	*	b	(3)
5	+	t2	t4
6	=	(5)	a

fig:- Three address code

fig:- Triples

(ii) Quadruples:- A quadruple has 4 fields

op	operand1	operand2	result name

For Eg: quadruple for $a = b * -c + b * -c$

t_2 t_1 t_4 t_3 t_5

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

fig:- Three address code

op	op1	op2	result
-	c	- X	t_1
*	b	t_1	t_2
-	c	- X	t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5	- X	a

fig:- Quadruple.

* Assignment Statements: (Refer Explaination written at 6th chapter)

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider a grammar

$S \rightarrow id := E$
$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow (E_1)$
$E \rightarrow id$

The translation scheme of above grammar is given below.

Production	Semantic Rule
$S \rightarrow id = E$	{ p = look-up(id.name); if p ≠ nil then Emit(p = E.place) Else Error; }
$E \rightarrow E_1 + E_2$	{ E.place = newtemp(); Emit(E.place = E_1.place + E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place = newtemp(); Emit(E.place = E_1.place * E_2.place }
$E \rightarrow (E_1)$	{ E.place = E_1.place }
$E \rightarrow id$	{ p = look-up(id.name); if p ≠ nil then emit(p = E.place) Else Error; }

In the above SDD, the function look-up(id.name) searches the id in the symbol table, if it finds the mentioned id & the p returns not nil and if it is equals to nil then it will be an error

'emit' is nothing but generate. It generates code according to the appropriate rule.

(Question to be asked on the above question
*- Give syntax directed translation for assignment statements).

Backpatching:-

A key problem when generating code for boolean expression and flow-of-control statements is that of matching a jump instruction with the target of the jump.

For Eg consider below boolean expression

if (B) S

in the above boolean statements, compiler get confuse where to control go when 'B' is true and where the control should go when 'B' is false.

Backpatching is process of where list of jumps are passed as synthesized attributes. specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when proper label can be determined. All the jumps on a list have the same target field.

Backpatching can be used to generate code for boolean expression and flow of control statements in one pass.

Date __/__/__
Page

Backpatching for Boolean expression:

Here we use synthesized attributes truelist and falselist of non-terminal B are used to manage labels. In particular, B.truelist will be a list of jump or conditional jump instruction into which we must insert the label to which control goes if B is true. B.falselist likewise is the list of instructions that eventually get the label to which control goes when B is false. Similarly, a statement 'S' has synthesized attribute S.nextlist, denoting a list of jump to the instruction immediately following the code for S.

We use three functions

1. makelist(i): creates a new list containing only 'i', an index into the array of instructions; makelist returns a pointer to the newly created list.
2. merge(P_1, P_2): concatenates the lists pointed to by P_1 and P_2 and return pointer to the concatenated list.
3. backpatch(P, i) inserts 'i' as the target label for each of the instruction on the list pointed to by P.

Translation scheme for boolean expression using backpatching:

Consider a grammar

$B \rightarrow B_1 \,||\, MB_2 \mid B_1 \,\&\&\, MB_2 \mid B_1 \mid (B_1) \mid$ true | false

The translation scheme as follows.

1. $B \rightarrow B_1 \mid\mid M B_2$	{ Backpatch (B_1.falselist, M.instr); B.truelist = merge (B_1.truelist, B_2.truelist); B.falselist = B_2.falselist; }
2. $B \rightarrow B_1 \&\& M B_2$	{ backpatch (B_1.truelist, M.instr); B.truelist = B_2.truelist; B.falselist = merge (B_1.falselist, B_2.falselist); }
3. $B \rightarrow !B_1$	{ B.truelist = B_1.falselist; B.falselist = B_1.truelist; }
4. $B \rightarrow B_1$	{ B.truelist = B_1.truelist; B.falselist = B_1.falselist; }
5. $B \rightarrow$ true	{ B.truelist = makelist (nextinstr); gen ('goto _'); }
6. $B \rightarrow$ false	{ B.falselist = makelist (nextinstr); gen ('goto _'); }

fig:- Translation scheme for boolean expression.

(Questions to be asked on above topic.

* Explain Backpatching for boolean expression

Or

what is backpatching? Explain backpatching process for boolean expression.

Or

Give translation scheme for boolean expressio

Date __/__/__
Page _____

* Procedure calls:-

Procedure is an important and frequently used programming construct for a compiler.

The translation for a call occurs includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence.

→ when a procedure call occurs then space is allocated for activation record.
→ Evaluate the arguments of the called procedure
→ Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
→ save the state of the calling procedure so that it can resume execution after the call.
→ Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
→ Finally generate a jump to the beginning of the code for the called procedure.

Let us consider grammar

S → call id (Elist)
Elist → Elist, E
Elist → E

A suitable translation scheme for above grammar is as follows

P.T.O.

Production	Semantic Rule
S → call id (Elist)	For each item P on queue do Gen (param p) Gen (call .id place)
Elist → Elist, E	append E.place to the end of queue.
Elist → E	initialize QUEUE to contain only E.place.

Fig: Translation Schame for procedure call.

Queue is used to store the list of parameters in procedure calls.

END

Unit-5

Code optimization:

code optimization :- is a process of improving the efficiency of program execution.

or

Elimination of unnecessary instructions in object code of the replacement of one sequence of instructions by a faster instructions that does the same thing.

VVIMP Principle sources of optimization

There are number of ways in which a compiler can improve a program without changing the function it computes.

1. common sub expression elimination
2. copy propogation
3. Dead code elimination
4. Constant folding
5. Frequency reduction
6. Strength reduction.

1. Common sub-expression elimination :- An occurrences of an expression E is called a common sub expression if E was previously computed, and the values of variable in E have not changed since the previous computation. we can avoid recomputing the expression if we can use the previously computed value.

For Eg:
$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_4 = 4 * i$
$t_5 = n$
$t_6 = b[t_4] + t_5$

In the above three address code $t_3 = 4*i$ is eliminated as its computations is already in t_1 and the value of i is not been changed.

i.e:
$t_1 = 4*i$
$t_2 = a[t_1]$
$t_3 = 4*j$
$t_5 = n$
$t_6 = b[t_1] + t_5$

2. copy propogation :-

Assignments of the form $x := y$ called copy statements. The idea behind the copy-propogation transformation is to use 'y' for x, whenever possible.

For eg

$x := y$
$z = x*1*1$

in the above expression $z = x*1*1$ instead of x, we can directly use y

i.e $z = y*1*1$.

3. Dead code elimination :-

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is a dead code at that point.

A related idea is dead or useless code, statements that compute values that never get used

For eg.

```
i=0;
if(i=1)
{
  a=b+5;
}
```

Here 'if' statement is dead code because this condition will never get satisfied. So, we eliminate both the test & printing from the object code.

4 Constant folding:-

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

For eg: $a = \frac{4}{2}$

above expression yeilds value $a=2$ and it will remain same as it is constant.

5. Frequency reduction:-

Frequency reduction is a process of moving a peice of code from a place where it is executing larger/more times to a place where it executes very fewer times.

For eg:

```
z=3;
for(i=0; i<10; i++)
{
  a=4+z;
  P=a+i;
}
```

In the above for loop 'a' value remain same for all the 10 iteration so, we can move it outside the loop so that it can execute only once.

i.e

```
z=3
a=4+z
for(i=0; i<10; i++)
{
    p= a+i;
}
```

6 **Strength reduction:-**

Strength reduction is a process of replacing expensive (more time consuming operations) by equivalent but less expensive operations

For Eg:- $a = 2^2$ is replaced by $a=2+2$.

square root always takes more time to evaluate than addition. So, we can replace 2^2 by 2+2 as, result will be same

(Question to be asked on above ~~qtn~~ topic
* Explain in detail sources of optimization)

VVImp * **Peephole optimization:-**

Peephole is nothing but a small sliding window, peephole optimization is done by examining a sliding window of target instructions (called the peephole) and replacing

instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

The peephole is a small sliding window on a program.

characteristics of peephole optimization

→ Redundant instruction elimination
→ Flow of control optimization
→ Algebric simplifications
→ use of machine idioms.

① (i) Redundant instruction elimination

Consider the below instruction below

```
LD   R0, a
ST   a, R0
```

in the above instructions, we can delete the store instruction because whenever it is executed the value of a is store into R0, but in the first instruction also 'a' will be stored in R0 so we can delete redundant instruction that is ST a, R0.

(ii) Eliminating unreachable code

Unreachable code, a part of the source code that will never be executed due to inappropriate exit points/control flow.

For Eg:

```
function add(x y)
{
    return x+y;
    int z= x*y;
}
```

in the above function int z = x * y is unreachable code and can be eliminated.

2. Flow-of-control optimizations:-

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimization.

We can replace the sequence

```
    goto L1
     :
L1: goto L2
```

by the sequence

```
    goto L2
     :
L1: goto L2
```

3. Algebric simplification and Reduction in strength

In algebric simplification, we can eliminate instructions like

$$x = x + 0$$

or

$$x = x * 1$$

similary, reduction in strength transformations can be applied to replace expensive operations by equivalent cheaper ones on the target machine.

4. Use of machine idioms:-

Here we use some machine hardware to perform code optimisations. For eg some machines have auto increment and auto decrement addressing modes.

(* What is peephole optimisation? Explain characteristics of peephole optimisation.

vvimp *** Basic blocks & Flow graphs

Basic block and flow graphs can be constructed as follows.

1. Partition the intermediate code into basic blocks with the properties that
   @ The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
   (b) control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks becomes the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

Basic blocks:-

First job is to partition a sequence of three address instructions into basic blocks. We begin new basic block with the first instruction and keep adding instruction until we met either jump, a conditional jump or a label on the following instruction.

This idea is formalized in the following algorithm.

Algorithm: Partitioning three address instruction into basic blocks.

Input:- A sequence of three address instruction

output: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

Method:- First we determine those instructions in the intermediate code that are leaders, that is, the first instruction of each basic blocks. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader
2. Any instruction that is target of a conditional or unconditional jump is a leader
3. Any instruction that immediately follows a conditional or unconditional jump is a leader

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

For Eg: Consider following sequence of three address instruction.

L1	1.	i = 1
L2	2.	j = 1
L3	3.	$t_1 = 10 * i$
	4.	$t_2 = t_1 + j$
	5.	$t_3 = 8 * t_2$
	6.	$t_4 = t_3 - 88$
	7.	$a[t_4] = 0.0$
	8.	j = j + 1
	9.	if j <= 10 goto (3)
L4	10	i = i + 1
	11.	if i <= 10 goto (2)
L5	12.	i = 1
L6	13.	$t_5 = i - 1$
	14	$t_6 = 88 * t_5$
	15	$a[t_6] = 1.0$
	16	i = i + 1
	17	if i <= 10 goto (13)

For the above sequence of three address instruction we can generate basic blocks as follows.

P.T.O.

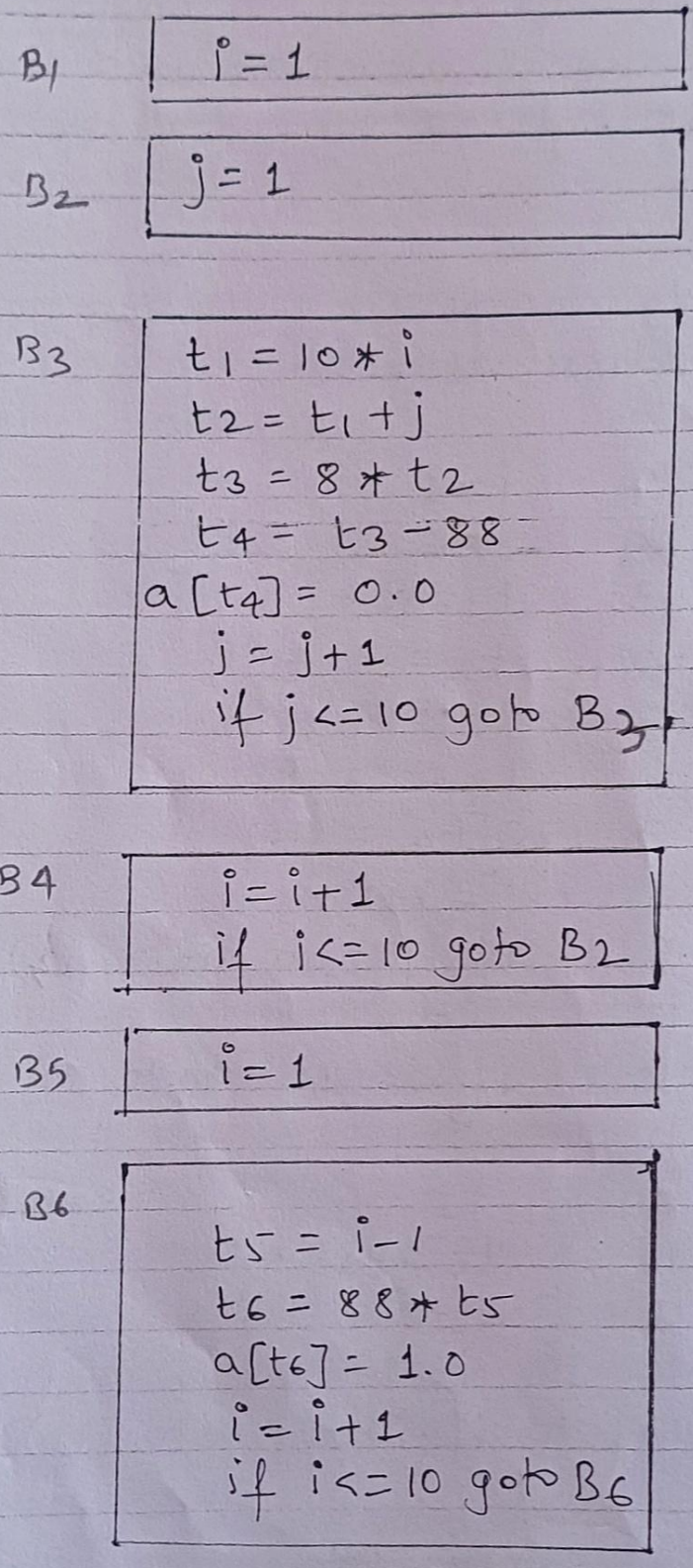


Fig :- Sequence of Basic blocks.

Date __/__/__
Page _____

* Flow graphs:-

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in Block B. There are two ways that such an edge could be justified:

→ There is a conditional or unconditional jump from the end of B to the beginning of C.

→ C immediately follows B in the original order of the three address instructions and B does not end in an unconditional jump.

often we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions.

There is an edge from the entry to the first executable ~~intermediate instruction~~ node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code.

Following flow graph shows how basic blocks are converted into flow graph.

P.T.O.

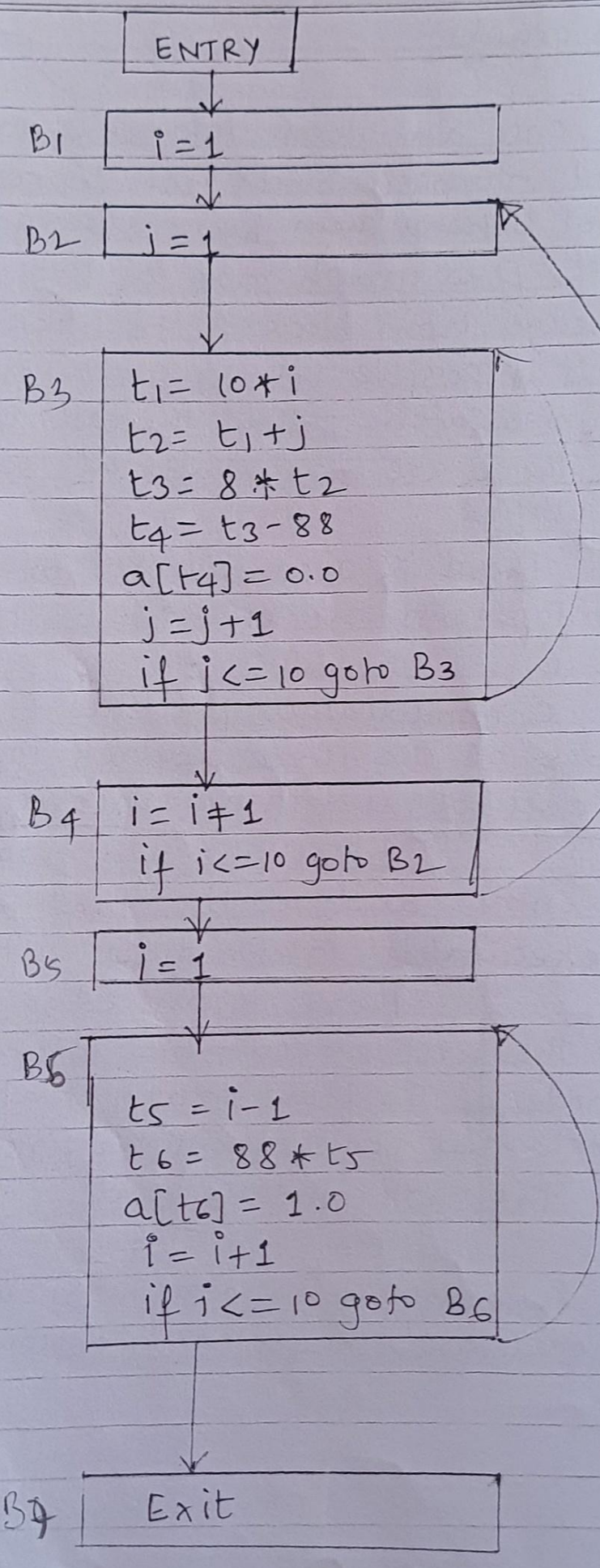


fig:- indicate flow graph.

Date ___/___/___
Page ___

Loops in flow graphs:-

Loops are important because programs spend most of their time executing them, and optimizations that improve the performance of loops can have a significant impact.

Dominators:-

We say node 'd' of a flow graph dominates node 'n', written d dom n, if every path from the entry node of the flow graph to 'n' goes through 'd'.

Note that under this definition, every node dominates itself.

Eg:- Consider the flow graph of below with entry node 1. The entry node dominates every node. Node 2 dominates only itself, since control can reach any other node along a path that begins with $1 \to 3$. Node 3 dominates all but not 1 and 2. Node 4 dominates all but not 1, 2 and 3, since all paths from 1 must begin with $1 \to 2 \to 3 \to 4$ or $1 \to 3 \to 4$. Node 5 and 6 dominate only themselves, since flow of control can skip around either by going through the other. Finally 7 dominates 7, 8, 9 and 10; 8 dominates 8, 9 and 10; 9 and 10 dominates only themselves.

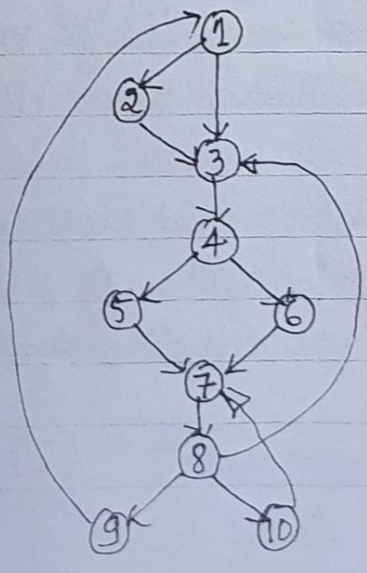


Fig: A flow graph.

A useful way of presenting dominator information in a tree, called the dominator tree, in which the entry node is the root, and each node d dominates only its descendants in the tree. For eg below fig shows the dominator tree for the above flow graph.

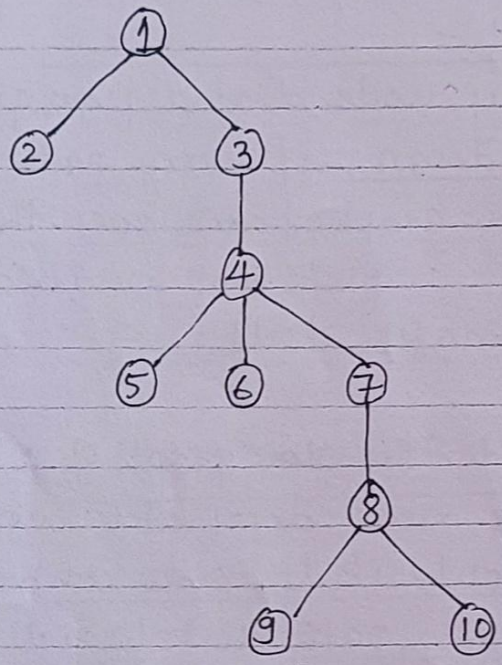


Fig: Dominator tree for the flow graph above

Natural loops and Inner loops:

Natural loops:- Natural loops is that in a flow graph if there exist a back edge $n \rightarrow b$, then the natural loop of the edge 'b' is given along with a set of nodes that do not go through 'b' to reach 'n'. In the edges $n \rightarrow b$, b is the head and n is the tail.

Inner loops:- A loop that does contain any other loop is called an inner loop. For eg, the flow graph given below has an inner loop $4 \rightarrow 2$ i.e the path from 2-3-4.

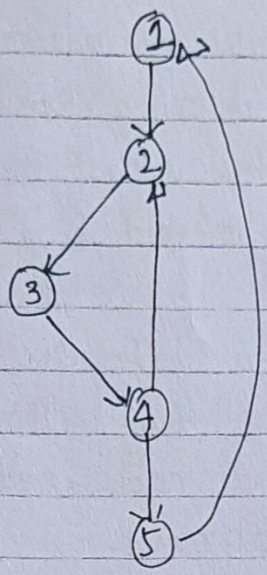


fig: Flow graph with an inner loop 4→2.

Data flow analysis and Equations

Data flow analysis collects information regarding the data flow in a program. The information includes how variables are assigned and referenced, what variables are live on exit from each block, what definitions of variables are available at a particular point in the program and so on.

The data-flow information is useful to an optimizing compiler for the purposes of code optimization. For example, the information about live variables at the end of each block helps to efficiently utilize registers.

Data flow information is collected by setting up and solving the dataflow equations. These equations for each data flow concepts like available expressions, reaching definitions, live variables and busy expressions etc.,

The general form of data flow equation is

$$END[S] = GEN[S] \cup (BGN[S] - KILL[S])$$

The meaning of above equation is that the information available at the end of a statement 'S' is either generated by the statement or is available at the beginning and is not killed by S.

By Generating of a statement 'S' means assigning a value to a variable. By killing of 'S' means any one of its operand and redefined prior to its use in the subsequent program.

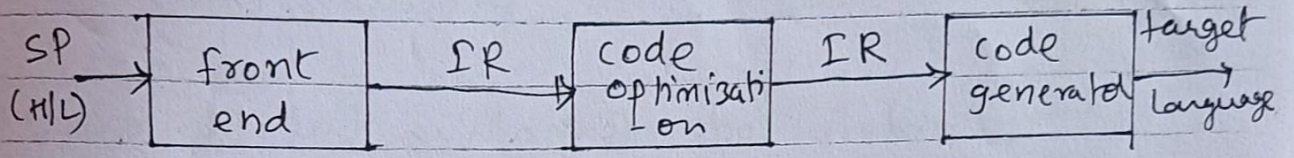
END

Unit - 6

code generation.

code generator takes input as a intermediate code of Intermediate Representation and generates output as a target code (Assembly / Machine Language).

The below figure shows function of code generator.



Issues in Design of code generator:

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

① Input to the code generator:-

Input to the code generator is the intermediate representation of the source program produced by the front end, along with the information in the symbol table. There are many choices for the IR - representation, for Eg. three-address-code representations such as Triples, quadruples, indirect triples, linear representation such as postfix notation and graphical representation such as syntax tree and DAG's. IR should be relatively low-level IR, so that target machine can directly manipulate IR and can convert into target program.

2. The target program:-

Final target program may be
- → Absolute code
- → Relocatable code
- → Assembly code.

Absolute code is nothing but the address of the instruction will be fixed so that it will be easy for target machine to execute the code.

Relocatable code is nothing but the address of code may change during run-time and some-times target may or may not support this kind of code. For Eg: Many architecture supports base addressing, relative addressing to add relocation processing. If relocation is not supported by target machine then compiler must insert code for ensure smooth relocation.

Assembly language code is generated so that assembler can further convert it into machine language.

3. Instruction selection:-

Code generator must map the IR program into a code sequence that can be executed by the target machine. If the IR is high level, the code generator may translate each IR-statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs further optimization. If the IR reflects some of the low-level details of the

underlying machine, then the code generator can use this information to generate more efficient code sequences.

4. Register allocation :-

A key problem in code generation is deciding what values to hold in what registers. Registers are fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two subproblems.

1. Register allocation, during which we select the set of variables that will reside in registers at each point in the ~~problem~~ program.
2. Register assignment, during which we pick the specific register that a variable will reside in.

5 Evaluation order :-

The order in which computations are performed can affect the efficiency of the target code. Some computations orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult, NP-complete problem.

vvimp
* Basic blocks, flow graphs and loops:

(Covered in 5th chapter)

* Next use information:

Next-use information is needed for dead-code elimination and register assignment.

Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register can be assigned to another variable will never be referenced subsequently, then that register can be assigned to another variable.

The use of a name in a three-address statement is defined as follows. Suppose three address statement 'i' assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening state assignments to x, then we say statement 'j' uses the value of x computed at statement i. We further say that 'x' is live at statement i.

Now we will see how next use and liveliness of a statement $x = y + z$, using following algorithm.

Date __/__/__
Page _____

Algo:- Determining the liveness and next-use information for each statement in a basic block.

Input:- A basic block 'B' of three address statements. We assume that the symbol table initially shows all non-temporary variables in B as being live on exit.

output:- At each statement i: $x = y + z$ in B, we attach to 'i' the liveness and next-use information of x, y and z.

Method:- We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following.

1. Attach to statement 'i' the information currently found in the symbol table regarding the next use and liveness of x, y and z.

2. In the symbol table, set x to "not live" and "no next use"

3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

* Sample example for how to liveness of a variable in a statement using next-use and liveness algorithm.

Note two points.

① Initially all temporary variables are dead and actual variables are live.

② As mentioned in the algorithm we need to start scanning from backward to beginning in a basic block for finding liveness.

Consider a block containing sequence of three address instruction.

1. $t_1 = 4 * i$
2. $t_2 = a[t_1]$
3. $t_3 = 4 * i$
4. $t_4 = b[t_3]$
5. $t_5 = t_2 * t_4$
6. $t_6 = Prod + t_5$
7. $Prod = t_6$
8. $t_7 = i + 1$
9. $i := t_7$
10. if $i <= 20$ goto ...

As mentioned above initially all temporary variables are dead and actual variables are live at line-0. Complete liveness information is shown in below table. 'D' indicates Dead and L(0) indicate Live at line '0', similarly L(1) Live at line '1' and so, on. First we start with line no 10, at line 10 'i' will live and rest other variables will be same as it was in the initial. And same idea will be applied to all the statements.

variables	Initial	For line 10	For line 9	For line 8	For line 7	For line 6	For line 5	For line 4	For line 3	For line 2	For line 1
T_1	D	D	D	D	D	D	D	D	D	L(2)	D
T_2	D	D	D	D	D	D	L(5)	L(5)	L(5)	D	D
T_3	D	D	D	D	D	D	D	L(4)	D	D	D
T_4	D	D	D	D	D	D	L(5)	D	D	D	D
T_5	D	D	D	D	D	L(6)	D	D	D	D	D
T_6	D	D	D	D	L(7)	D	D	D	D	D	D
T_7	D	D	L(9)	D	D	D	D	D	D	D	D
a	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(2)	L(2)
b	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(4)	L(4)	L(4)	L(4)	L(4)
Prod	L(0)	L(0)	L(0)	L(0)	D	L(6)	L(6)	L(6)	L(6)	L(6)	L(6)
i	L(0)	L(10)	D	L(8)	L(8)	L(8)	L(8)	L(8)	L(3)	L(3)	L(1)

At statement 9, t_7 is live and i becomes dead so it is updated in the table and rest other variable will be same as it was on line 10. The same process will be continued for all the statements.

END

(some other topics of 6th chapters are not important. Prepare only topics which I have written on the notes.

Difference between S-attributed & L-attributed Defn

	S-Attributed	L-Attributed
1	uses only synthesized attributes	(1) uses both synthesized & inherited attributes. Inherited attribute can inherit values either from parent or by left side siblings
2.	Semantic actions are placed at right end of production. Eg: $A \to BC \{ \ldots \}$	(2) Semantic actions are placed anywhere on RHS Eg: $A \to BC$ then $A \to \{ \ldots \} BC$ $A \to B \{ \ldots \} C$ $A \to BC \{ \ldots \}$
3.	Attributes are evaluated during bottom up parsing	(3) by traversing parse tree depth first, left to right