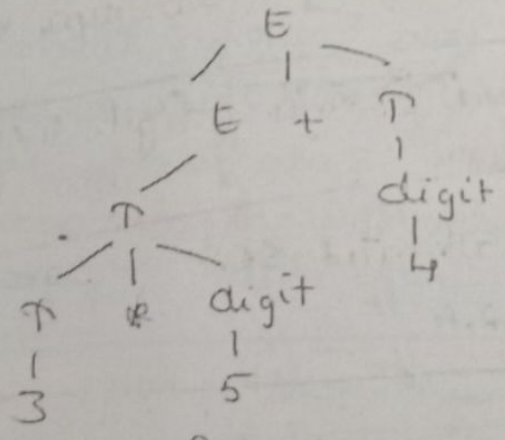
3) $L \rightarrow L_1, B$	1) L.lhs = L_1.lhs + (2 ^ lhs_exponent * B.val) 2) L.rhs = L_1.rhs + (2 ^ L.rhs_exponent * B.val) 3) L.lhs_exponent = L_1.lhs_exponent + 1 4) L.rhs_exponent = L_1.rhs_exponent + 1
4) $L \rightarrow B$	1) L.lhs = 2 ^ lhs_exponent * B.val 2) L.rhs = 2 ^ L.rhs_exponent * B.val 3) L.lhs_exponent = 0 4) L.rhs_exponent = -1
5) $B \rightarrow 0 \mid 1$	B.Val = digit.lexval

Application Of Syntax-Directed Translation

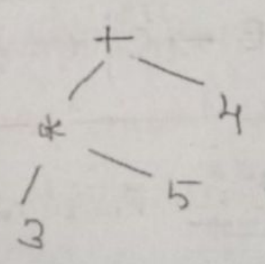
1. Construction Of Syntax Tree

SDD's are useful for construction of syntax tree.

A syntax tree is condensed form of parse tree



Parse tree



Syntax tree

* Syntax trees are useful for representing programming language constructs like expressions & statements.

* They help compiler design by decoupling parsing from translat^n.

* Each node of a syntax tree represent a construct; the children of the node represent the meaningful components of a construct

  Eg: A syntax tree node representing an expression E1 + E2 has label + & 2 children representing the sub expression E1 & E2

* Each node is implemented by objects with suitable no of fields; each object will have an op field that is the label of node with additional fields as follow:

  a) If the node is a leaf, an additional field holds the lexical value for the leaf. This is created by function Leaf(op, val).

  b) If the node is an interior node, there are as many fields as the node has children in syntax tree. This is created by function Node(op, c1, c2, ... ck)

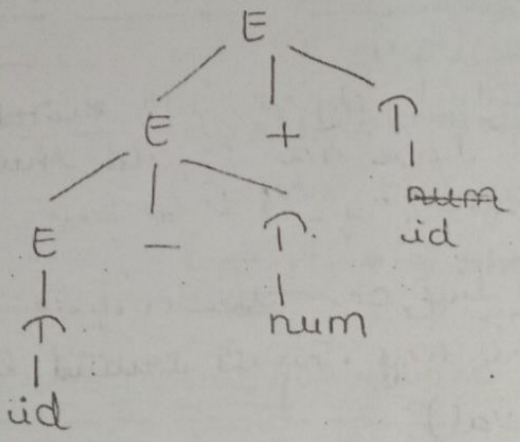
Example: The S-attributed definat^n in fig below constructs syntax trees for a simple expr grammar involving only binary operators + & -. As usual these operators are at the same precedence level & are jointly left associative. All nonterminals have one synthesized attr node, which represents a node of the syntax tree.

a-4+c

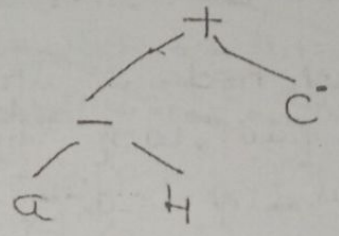
	PRODUCTION	SEMANTIC RULES
1.	$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
2.	$E \longrightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
3.	$E \longrightarrow T$	E.node = T.node

T → (E)	T.node = E.node
T → id	T.node = newLeaf(id, id.entry)
T → num	T.node = newLeaf(num, num.val)

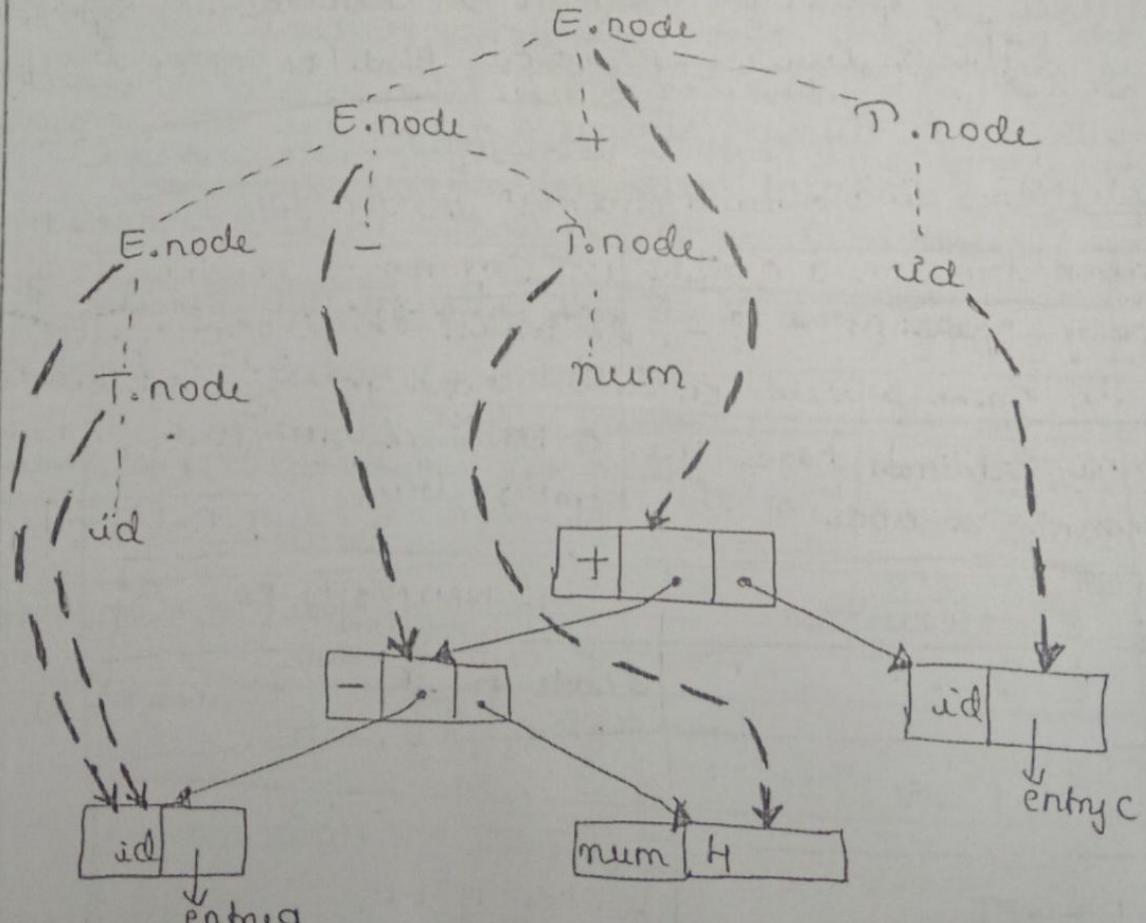
Step : Parse tree



Syntax tree



Step3: Syntax tree for a − b + c using above SDD :



Steps in Construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reduction during a bottom up parse, then the sequence of steps shown below ends with p_5 pointing to the root of the constructed syntax tree.

1) P_1 = new Leaf (id, entry-a)
2) P_2 = new Leaf (num, 4)
3) P_3 = new Node ('-', P_1, P_2)
4) P_4 = new Leaf (id, entry-c)
5) P_5 = new Node ('+', P_3, P_4)

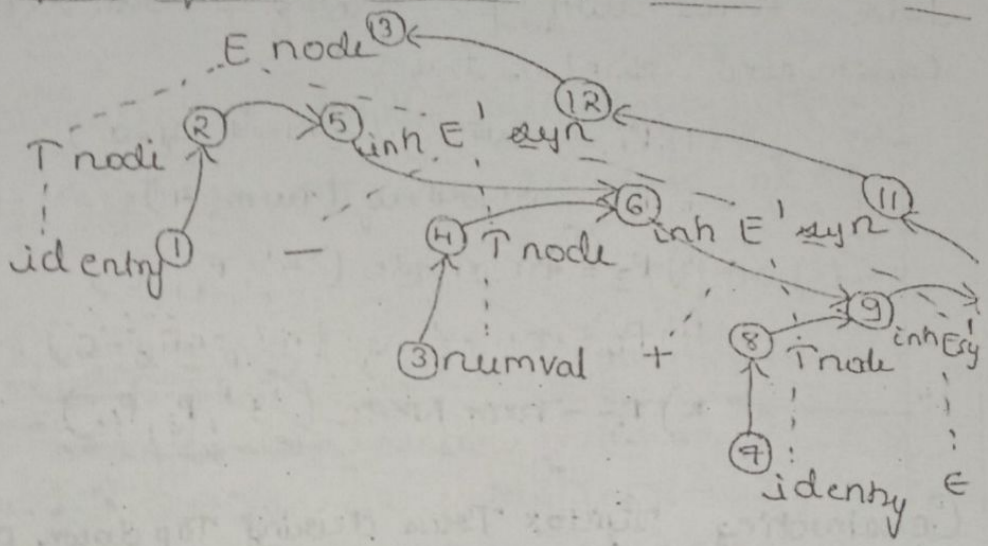
Constructing Syntax Trees during Top down parsing

With a grammar designed for top-down parsing, the syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees. The L-attributed definition below performs the same translation as the S attributed definition shown before.

1)	$E \to TE'$	E.node = E'.syn E'.inh = T.node
2)	$E' \to +TE_1'$	E_1'.inh = ~~E'.inh + P_3~~ new Node('+', E'.inh, T.node) E'.syn = E_1'.syn
3)	$E' \to -TE_1'$	E'.inh = new Node('-', E'.inh, T.node) E'.syn = E_1'.syn
4)	$E' \to \epsilon$	E'.syn = E'.inh
5)	$T \to (E)$	T.node = E.node

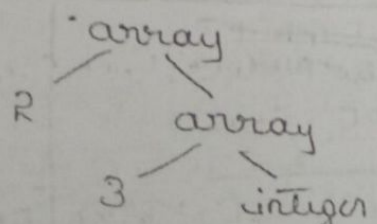
6) T → id	T.node = new Leaf (id, id.entry)
7) T → num	T.node = new Leaf (num, num.val)

Dependency Graph for a - 4 + c with attributed SDD



STRUCTURE OF A TYPE

This is an example of how inherited attribute can be used to carry info from one part of the parse tree to another. In C, the type int[2][3] can be read as "array of 2 arrays of 3 integer". The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below.



product^n	Semantic rules
1) T → BC	T.t = C.t C.b = B.t
2) B → int	B.t = integer
3) B → float	B.t = float
4) C → [num] C_1	C_0.t = array (num.val, C_1.t) C_1.b = C.b
5) C → ϵ	C.t = C.b

→ The non terminals B & T have a synthesized attribute t representing a type

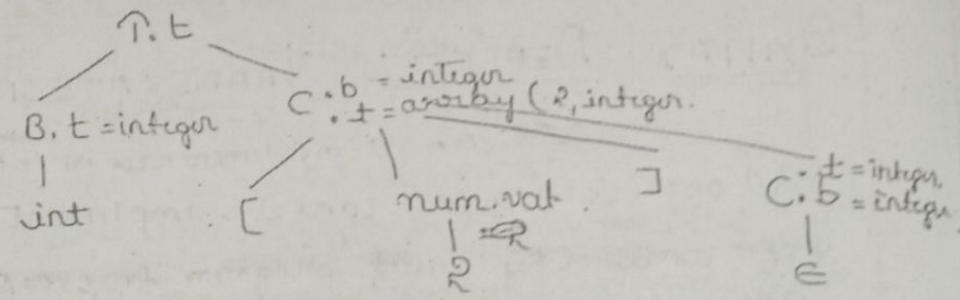
→ The non terminal C has 2 attributes: an inherited attr (b) & a synthesized attr (t).

→ The inherited attribute, b pass a basic type down the tree

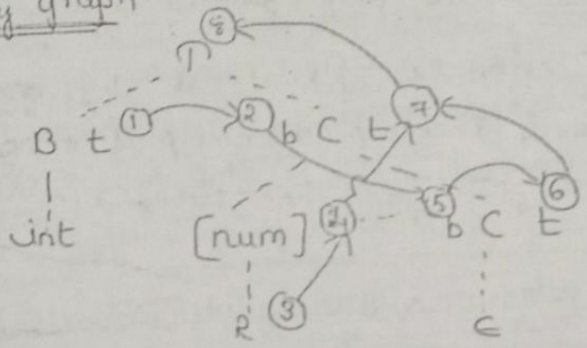
→ The synthesized attribute, t accumulate the result.

→ An annotated parse tree for i/p : int [2]

(i) int [2]

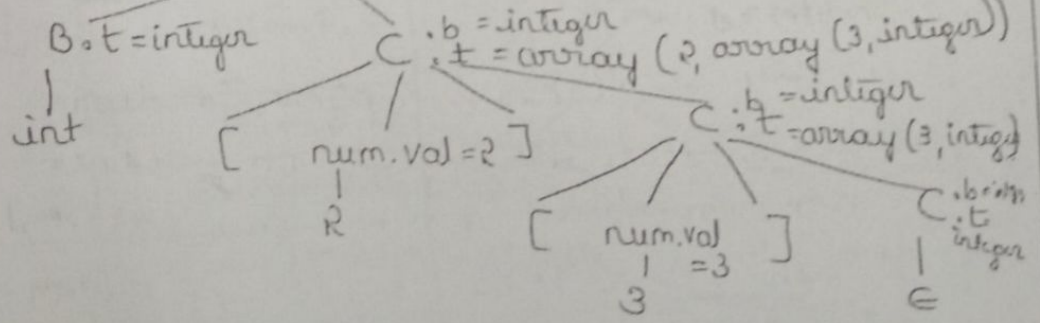


Dependency graph

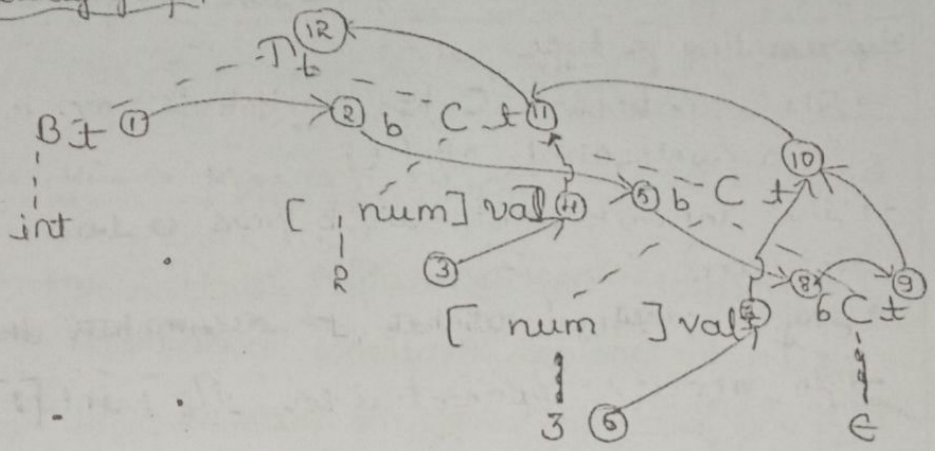


(ii) int [2][3]

Annotated parse tree: T.t = array (2, array (3, integer))



Dependency graph



SYNTAX DIRECTED TRANSLATION SCHEME:

SDT is a complementary notation to SDD.

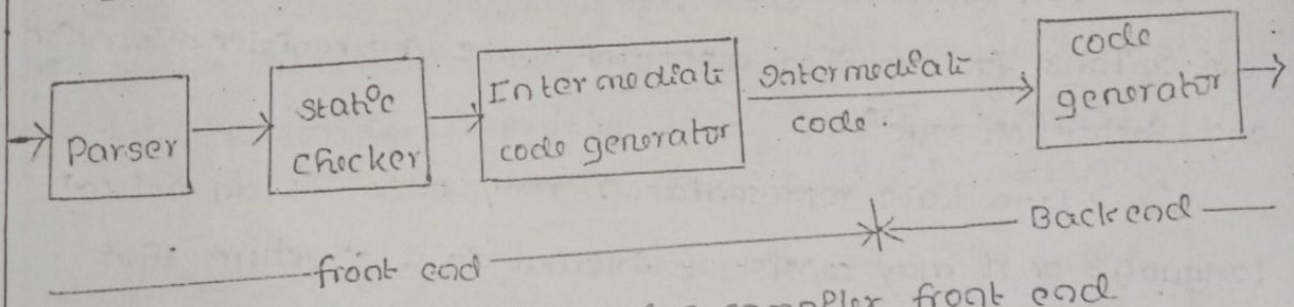
* All applicatn of SDD can be implemented using SDT.
* SDT is a CFG with program fragments called semantic actions embedded with productn bodies.
* Any SDT can be implemented by first building a parse tree & then performing the actions in a left to right, depth first order i.e, during preorder traversal.
* Typically SDT's are implemented during parsing without building parse tree. During parsing an action in a production body is executed as soon as all the grammar symbols to the left of action have been matched.
* SDT's that can be implemented during parsing can be characterized by introducing distinct marker non terminals in place of each embedded action.
* Each marker M has only one production $M \rightarrow \epsilon$.
* If grammar with marker non terminals can be parsed by a given method, then SDT can be implemented

UNIT-6

INTERMEDIATE CODE GENERATION

Intermediate Code generation.

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.



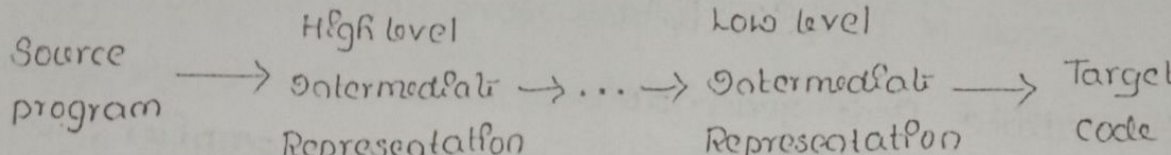
Logical structure of a compiler front end.

Parsing, static checking and intermediate code generation are done sequentially; sometimes they can be combined and folded into parsing.

Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing.

Ex: It ensures asserts that a break-statement in C is enclosed within a while-, for- or switch-statement; an error is reported if such an enclosing statement does not exist.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representation as



High level representations are close to the source language and are well suited to tasks like static type checking.

Ex: Syntax tree

Low level representations are close to the target machine & are suitable for machine dependent tasks like register allocation and instruction selection.

An intermediate representation may either be an actual language or it may consist of internal data structure that are shared by phases of the compiler.

Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.

A directed acyclic graph (DAG) for an expression identifies the common subexpressions of the expression.

Directed Acyclic Graphs for Expressions.

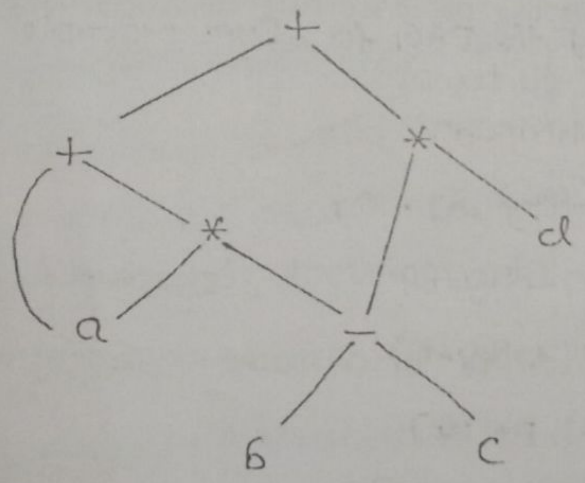
In DAG leaves represents the atomic operands and interior nodes represents the operators. as in the syntax tree.

A node N in a DAG has more than one parent if N represents a common subexpression; But in the syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

DAG gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Ex: DAG for the expression

$$a + a * (b - c) + (b - c) * d$$



↳ The leaf for 'a' has 2 parents, because 'a' appears twice in the expression

↳ The 2 occurance of the common subexpression b-c are represented by one node, the node labeled '−'

SDD to produce DAG.

PRODUCTION	SEMANTIC RULES
(1) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
(2) $E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
(3) $E \rightarrow T$	E.node = T.node
(4) $T \rightarrow (E)$	T.node = E.node
(5) $T \rightarrow id$	T.node = new Leaf (id, id.entry)
(6) $T \rightarrow num$	T.node = new Leaf (num, num.val)

It will construct a DAG, before creating a new node, these functions first check whether an identical node already exists.

If a previously created identical ~~ex~~ node exists, the existing node is returned.

Steps for constructing the DAG for above example.

(1) P_1 = Leaf (id, entry-a)

(2) P_2 = Leaf (id, entry-a) = P_1

(3) P_3 = leaf (id, entry-b)

(4) P_4 = Leaf (id, entry-c)

(5) P_5 = Node ('-', P_3, P_4)

(6) P_6 = Node ('*', P_1, P_5)

(7) P_7 = Node ('+', P_1, P_6)

(8) P_8 = Leaf (id, entry-b) = P_3

(9) P_9 = Leaf (id, entry-c) = P_4

(10) P_{10} = Node ('-', P_3, P_4) = P_5

(11) P_{11} = Leaf (id, entry-d)

(x11) P_{12} = Node ('*', P_7, P_{11})

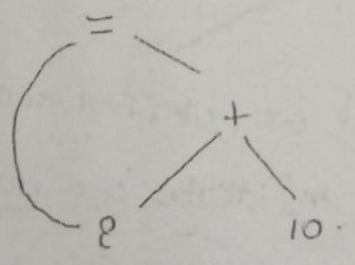
(x12) P_{13} = Node ('+', P_7, P_{12})

When the call to Leaf (id, entry-a) is repeated at step 2, the node created by the previous call is returned, so $P_2 = P_1$.

The Value-Number Method for Constructing DAG's

* The nodes of a DAG are stored in an array of records
* Each row of array represents one record & therefore one node.
* In each record, the first field is an operation code, labeling the label of the node. Leaves have one additional field which holds the lexical value and interior nodes have 2 additional fields indicating the left and right children.

Ex: DAG for $a = b + 10$ allocated in an Array



DAG

1	id		to entry for b
2	num	10	
3	+	1	2
4	=	1	3
5			

Array

* In this array, we refer to nodes by giving the integer index of the record for that node within the array.
* This integer historically has been called the "value number" for the node or for the expression represented by the node
* For above example node labeled + has value number 3 & its left & right children have value numbers 1 & 2 respectively.

Suppose that nodes are stored in an array & each node is referred to by its value numbers. Let the signature of an interior node be the triple $\langle op, l, r \rangle$ where op is the label, l is its left child's value number & r its right child's value number. A unary operator may be assumed to have $r = 0$.

ALGORITM: To construct the nodes of a DAG using value number method.

INPUT: Label op, node l and node r

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$

METHOD: Search the array for a node M with label op, left child l and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l and right child r & return its value number.

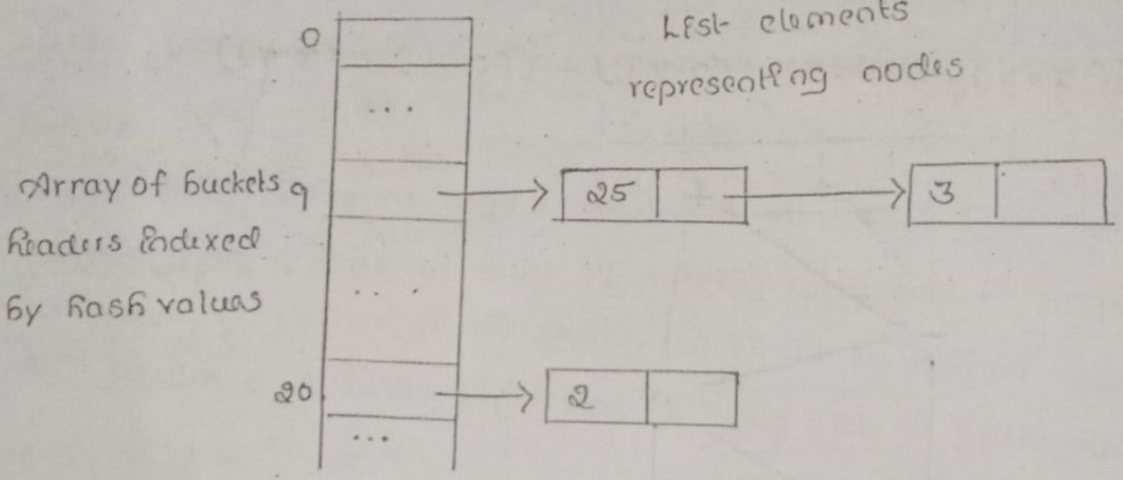
Above algorithm yields the desired output, but searching the entire array every time we are asked to locate one node is expensive.

A more efficient approach is to use a hash table, in which the nodes are put into "buckets" each of which typically will have only a few nodes. It supports dictionaries. which is an abstract data type that allows us to insert & delete elements of a set & to determine whether a given element is currently in the set.

To construct a hash table for the nodes of a DAG, we need a hash function h that computes the index of the bucket for a signature <op, l, r>.

The bucket index h(op, l, r) is computed deterministically from op, l & r so that we may repeat the calculation & always get to the same bucket index for node <op, l, r>

The buckets can be implemented as linked list as,



An array indexed by hash value, holds the bucket headers, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node <op, l, r> can be found on the list whose header is at index h(op, l, r) of the array.

Thus, given the output input node op, l & r we compute the bucket index h(op, l, r) & search the list of cells in this bucket for the given input node.

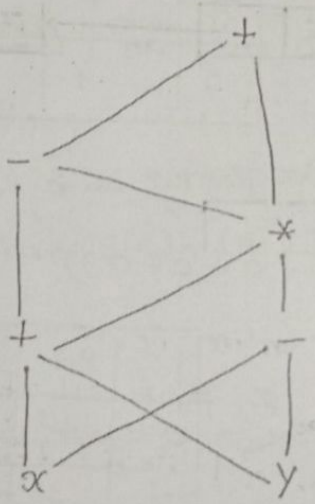
For each value number 'v' found in a cell, we must

check whether the signature <op, l, r> of the input node matches the node with value number v in the list of the cell. If we find a match, we return v. If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket-index h(op, l, r) & return the value number in that new cell.

Problems.

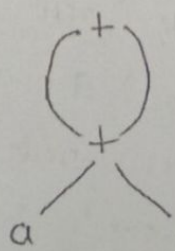
Construct the DAG for the expression.

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



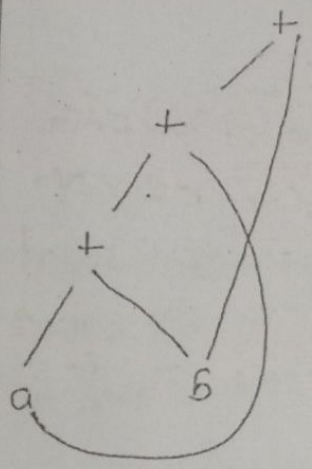
Construct the DAG & identify the value number for the sub expressions of the following expressions, assuming + associative from the left.

(a) $a+b+(a+b)$



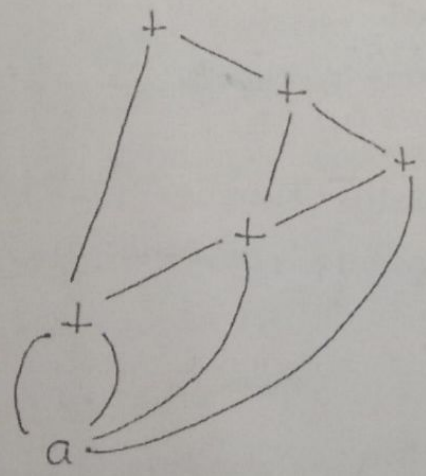
1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

(b) a + b + a + b



1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

(c) a + a + (a + a + a + (a + a + a + a))



1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

Three - Address Code

In 3-address code, there is atmost one operator on the right R side of an instruction, i.e, no built up arithmetic expression are permitted.

Thus a source-language expression like $x+y*z$ might be translated into the sequence of 3 address instructions

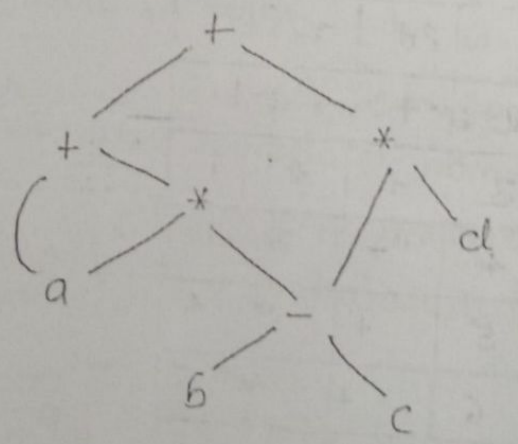
$t_1 = y * z$

$t_2 = x + t_1$

where t_1 & t_2 are compiler generated names.

3 address code is a linearized representation of a DAG in which explicit names correspond to the interior nodes of the graph.

Ex: Write DAG & its corresponding 3 address code for the expression $a + a * (b-c) + (b-c) * d$.



DAG

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$.

3-address code

Addresses and Instructions

3-address code is built from 2 concepts : address & instructions.

An address can be one of the following.

↳ A name : For convenience, we allow source program names to appear as addresses in 3-address code. In an implementation, a source name is replaced by a pointer to its symbol table entry, where all information about the name is kept.

↳ A constant : A compiler must deal with many different types of constants and variables.

↳ A compiler generated temporary : It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a 3-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching"

Here is a list of the common 3-address instruction forms

(i) Assignment instructions of the form $x = y$ op z where op is a binary arithmetic or logical operation & x, y & z are addresses.

(ii) Assignments of the form $x = $ op y, where op is a unary operation

(iii) Copy instructions of the form $x = y$, where x is assigned the value of y.

(iv) An unconditional jump goto L. The 3-address instruction with label L is the next to be executed.

(v) Conditional jumps of the form if x goto L and if False x goto L. These instructions execute the instruction with label L next if x is true and false respectively.

(vi) Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >= etc) to x & y & execute the instruction with label L next if x stands in relation relop to y. If not, the 3 address instruction following if x relop y goto L is executed next, in sequence

(vii) Procedures calls & returns are implemented using the following instructions: param x for parameters; call p,n & y = call p,n for procedure & function calls respectively & return y, where y representing a returned value, is optional

```
param x1
param x2
...
param xn
call p,n
```

The integer n indicating the no. of actual parameters in call p,n is not redundant because calls can be nested.

(viii) Indexed copy instructions of the form x = y[i] and x[i] = y. The instruction x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i] = sets the contents of the location i units beyond x to the value of y.

(ix) Address & pointer assignments of the form $x = \&y$, $x = *y$ and $*x = y$.

Ex: Consider the statement

do $i = i+1$; while $(a[i] < v)$;

2 possible translations of this statement are

L : $t_1 = i+1$	100 : $t_1 = i+1$
$i = t_1$	101 : $i = t_1$
$t_2 = i*8$	102 : $t_2 = i*8$
$t_3 = a[t_2]$	103 : $t_3 = a[t_2]$
if $t_3 < v$ goto L	104 : if $t_3 < v$ goto 100
(a) Symbolic labels	(b) position number.

The translation in (a) uses a symbolic label L, attached to the first instruction. The translation in (b) shows position number for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication $i*8$ is appropriate for an array of elements that each take 8 units of space.

Quadruples.

A quadruples has 4 fields, which we call op, arg1, arg2, & result. The op field contains an internal code for the operator.

Ex: $x = y + z$ is represented by placing + in op, y in arg1, z in arg2 & x in result.

The following are some exceptions to this rule.

(i) Instructions with unary operators like x = minus y or x = y do not use arg2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

(ii) Operators like param use neither arg2 nor result.

(iii) Conditional & unconditional jumps put the target label in result.

Ex: write quadruples for a = b * -c + b * -c.

t_1 = minus c	
t_2 = b * t_1	
t_3 = minus c = t_1	
t_4 = b * t_3	
t_5 = t_2 + t_4	
a = t_5	

(a) 3-address code

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

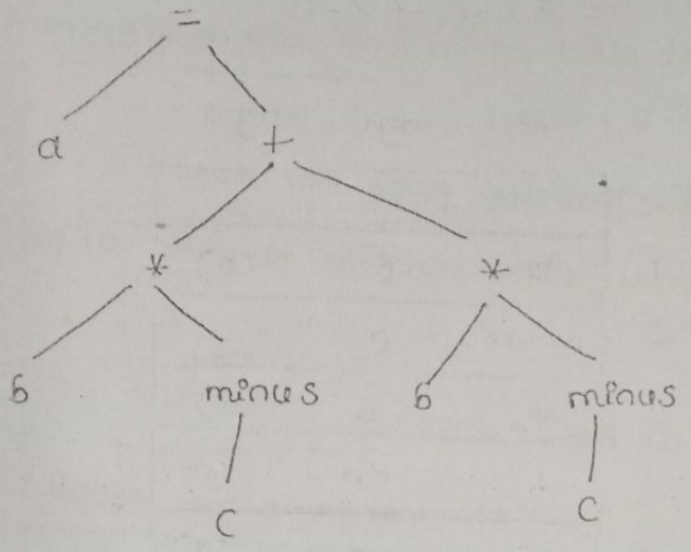
(b) Quadruples.

The special operator minus is used to distinguish the unary minus operator.

Triples

A triples has only 3 fields, op, arg1 & arg2. Note that the result field in quadruples is used primarily for temporary names. Using triples, we refer to the result of the operation x op y by its position, rather than by an explicit temporary names.

Ex: Write triples for $a = b * -c + b * -c$



(a) Syntax tree

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(b) triples.

A ternary operator like x[i] = y requires 2 entries in the triple structure, for example, we can put x & i in one triple & y in the next.

A benefits of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

Indirect triples.

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

With indirect triples, an optimizing compiler can move an

instruction by reordering the instruction list without affecting the triples themselves.

Ex: write indirect triples for $a = b * -c + b * -c$

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5.	=	a	(4)
	.	. . .	.

Static Single-Assignment Form:

Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.

2 distinctive aspects of SSA that distinguish SSA from 3-address code

(1) All assignments in SSA are to variables with distinct names.

Ex: 3-address code

$p = a + b$

$q = p - c$

$p = q * d$

$p = e - p$

$q = p + q$

Static single assignment form

$p_1 = a + b$

$q_1 = p_1 - c$

$p_2 = q_1 * d$

$p_3 = e - p_2$

$q_2 = p_3 + q_1$

the same variable may be defined in 2 different control flow paths in a program. For example, the source program

```
if (flag) x = -1; else x = 1;
y = x * a;
```

If we use different names for x in the true part & false then conflict arises which name should use in y = x * a.

(ii) SSA uses a notational convention called ϕ-function to combine the 2 definitions of x

```
if (flag) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

Here $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional & the value x_2 if the control flow passes through the false part.

Translate the arithmetic expression a + - (b+c) into

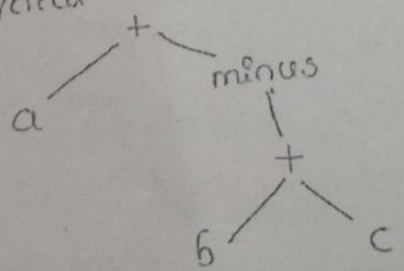
(a) A syntax tree

(b) Quadruples

(c) Triples

(d) Indirect triples

(a) Syntax tree



$t_1 = b + c$

$t_2 = \text{minus } t_1$

$t_3 = a + t_2$

(b) Quadruples

	op	arg 1	arg 2	result
0	+	b	c	t_1
1	minus	t_1		t_2
2	*	a	t_2	t_3

(c) triples

	op	arg 1	arg 2
0	+	b	b c
1	minus	(0)	
2	*	a	(1)

(d) Indirect triples.

	Instructions
35	(0)
36	(1)
37	(2)

	op	arg 1	arg 2
0	+	b	c
1	minus	(0)	
2	=	a	(1)

Translation of Expressions.

An expression with more than one operator, like a + b * c, will translate into instructions with almost one operator per instruction. An array reference A[i][j] will expand into a sequence of 3-address instructions that calculate an address for the reference.

Operations within Expressions

The following syntax-directed definition builds up the 3-address code for an assignment statement S using attribute code for S & attribute addr & code for an expression E. Attributes S.code & E.code denote the 3 address code for S & E respectively. Attribute E.addr denotes the address

QUESTIONS

1. Define quadruples, triples and static single assignment form.

A quadruples has 4 fields, op, arg1, arg2 and result. The op field contains an encremental code for the operator.

Ex: the quadruples for $a = (6 * -c) + (6 * -c)$

$t_1 = \text{minus } c$

$t_2 = 6 * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg1	arg2	result
0	minus	c		t_1
1	*	6	t_1	t_2
2	minus	c		t_3
3	*	6	t_3	t_4
4	+	t_0	t_4	t_5
5	=	t_5		a

A triples has only 3 fields op, arg1 & arg2

Ex: the triples for $a = 6 * -c + 6 * -c$.

	op	arg1	arg2
0	-minus	c	
1	*	6	(0)
2	minus	c	
3	*	6	(2)
4	+	(1)	(3)
5	=	a	(4)

Static single-statement assignment form is an intermediate representation that facilitates certain code optimizations

Ex:

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$
(a) 3-address code	(b) Static single assignment form.

2. Develop SDD to produce directed acyclic graph for an expression show the steps for constructing the DAG for the expression $a + a * (b - c) + (b - c) * d$.

Syntax directed definition is,

E → PRODUCTION	SEMANTIC RULES
(i) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
(ii) $E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
(iii) $E \rightarrow T$	E.node = T.node
(iv) $T \rightarrow (E)$	T.node = E.node
(v) $T \rightarrow id$	T.node = new Leaf (id, id.entry)
(vi) $T \rightarrow num$	T.node = new Leaf (num, num.val)

Steps for constructing the DAG1

(1) P_1 = Leaf (id, entry-a)

(2) P_2 = Leaf (id, entry-a) = P_1

(3) P_3 = Leaf (id, entry-b)

(4) P_4 = Leaf (id, entry-c)

(5) P_5 = Node ('-', P_3, P_4)

(6) P_6 = Node ('*', P_1, P_5)

(7) P_7 = Node ('+', P_1, P_6)

(8) P_8 = Leaf (id, entry-b) = P_3

(9) P_9 = Leaf (id, entry-c) = P_4

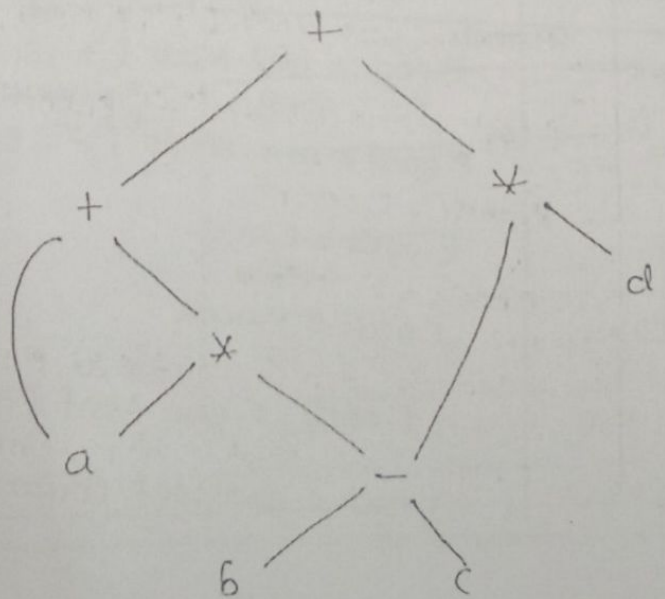
(10) P_{10} = Node ('-', P_3, P_4) = P_5

(11) P_{11} = Leaf (id, entry-d)

(12) P_{12} = Node ('*', P_5, P_{11})

(13) P_{13} = Node ('+', P_7, P_{12})

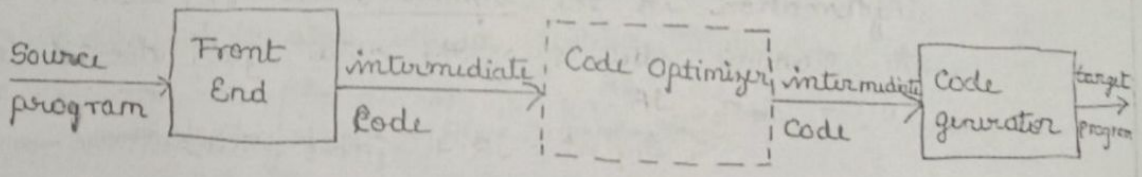
DAG1



UNIT -8
CODE GENERATION

INTRODUCTION :

* Code generation is the final phase in the compiler design.
* The code optimizer accepts intermediate code representation which is generated from the front end of the compiler & produces another intermediate code representation which is optimized.
* Code generator takes intermediate representation produced by code optimizer along with supplementary information in symbol table of the source program & produce as output an equivalent target program.



* Code generator has 3 main tasks:
  1) Instruction selection
  2) Register allocation & assignment
  3) Instruction Ordering

1) INSTRUCTION SELECTION :

Choose a appropriate target machine code instructions to implement the IR [intermediate representation] statements

2) REGISTER ALLOCATION & ASSIGNMENT :

Decide what values to keep in which registers

5) INSTRUCTION ORDERING

Decide in what order to schedule the execution of instructions.

8.1 ISSUES IN THE 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 repres of the source program produced by the front end along information in the Symbol table i.e., used to determ the runtime address of the data objects denoted by the names in IR.

< Input = IR + Symbol table >

* IR has several choices

(a) 3-address representation : quadruples, triples, indirect trip

(b) Virtual machine representat^n : byte codes & stack mach codes

(c) Linear representat^n such as postfix notat^n

(d) Graphical representat^n such as syntax trees & DAG's

* Assumptions made are

(i) Front end produces low-level IR, i.e., values of names in it can be directly manipulated by the machine instruct^n.

(ii) Syntatic & semantic errors have been already detected

2) The Target Program :

* The Output of code generator is target program.
* The instruⁿ set architecture of the target machine has a significant impact on the design of code generator.
* Most common architecture are:

(a) CISC : It has few registers, has maximum of 2 operands & variety of addressing mode, variable length instructions & instructⁿ with side effects.

(b) RISC : It has many registers, has maximum of 3 operands with simple addressing modes, & relatively simple instructⁿ set architecture.

* Output may take variety of forms.

a) Absolute machine language [Executable Code]
b) Relocatable machine language [object files for linker]
c) Assembly language [facilitates debugging]

a) Absolute machine language has advantage that it can be placed in a fixed location in memory & immediately executed.

b) Relocatable machine language program allows subprograms to be compiled separately.

c) Producing Assembly language program as output makes the process of code generatⁿ somewhat easier.

3) Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine.

* The complexity of performing this mapping is determined by the factors such as:

(i) the level of the IR

(ii) the nature of the instruction set architectures.

(iii) the desired quality of the generated code.

(i) the levels of the IR:

> If the IR is high level, use code templates to translate each IR statements into a sequence of machine instruction.

> produces poor code, needs further optimizat^n.

> If the IR is low level, use ~~(code, this)~~ low level information to generate more efficient code sequence.

(ii) the nature of the instruction set architectures has strong effect on difficulty of instruct^n select^n.

> Uniformity & completeness of the instruct^n set are imp factors.

> If we do not care about the efficiency of the target program, instruct^n select^n is straightforward.

> For eg:

```
x = y + z  ⇒  LD   R0, y
              ADD  R0, R0, z
              ST   x, R0
```

∴ produces redundant LD & store

eg2:

```
a = b + c  ⇒  LD   R0, b
d = a + b        ADD  R0, R0, c
                 ST   a, R0      ──→ REDUNDANT
                 LD   R0, a
                 ADD  R0, R0, c
                 ST   d, R0
```

(iii) the quality of the generated code is determined by its speed & size.

>For eg:

```
a = a+1  ⇒ LD  R0, a
           ADD R0, R0, #1      replaced by      INC a
           ST  a, R0
```

4) Register Allocation:

* Instruct^n involving register operands are usually shorter & faster than those involving operands in memory.
* 2 subproblems:
  (i) Register allocation: Select the set of variables that will reside in registers at each point in the program.
  (ii) Register assignment: Select specific register that a variable will reside in.
* Complications imposed by the hardware architecture
  Eg: Register pairs for multiplicat^n & division.
* Multiplicat^n inst^n is of the form

```
M  x, y
```

  where x → Multiplicand, is the odd register of an even/odd register pair.
  y → Multiplier, is a single register.
  ⇒ Product → occupies the entire even/odd register pair.

* Division inst^n is of the form

```
D  x, y
```

  where x → dividend, occupies even register
  y → divisor, occupies odd/even register
  ⇒ quotient → stored in odd register
  remainder → stored in even register

Eg: two-3 address code sequences

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

Optimal machine-Code Sequences

L R1, a	L R0, a
A R1, b	A R0, b
M R0, c	A R0, c
D R0, d	SRDA R0, 32
ST R1, t	D R0, d
	ST R1, t

5) Evaluation Order:

* The order in which computations are performed can effect the efficiency of the target code.
* When instructⁿ are independent their evaluation order can be changed.
* Some computatⁿ 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.

ADDITIONAL INFORMATION: Eg

a+b-(c+d)*e

t1 = a + b
t2 = c + d
t3 = e * t2
t4 = t1 - t3

Reordering:

t2 = c + d
t3 = e * t2
t1 = a + b
t4 = t1 - t3

```
MOV R0, a
ADD R0, b
MOV R1, R0
MOV R1, c
ADD R1, d
MOV R0, e
MUL R0, R1
MOV R1, t1
SUB R1, R0
MOV R1, t4, R1
```

```
MOV R0, c
ADD R0, d
MOV R1, e
MUL R1, R0
MOV R0, a
ADD R0, b
SUB R0, R1
MOV t4, R0
```

~~THE~~

8.2 THE TARGET LANGUAGE:

For designing a good code generator, we need to have familiarity with target machine & its instruct^{n} set. Instead of generating code on a specific target machine, a general machine consisting of many registers are considered.

A SIMPLE TARGET MACHINE MODEL:

The characteristics of target machine mode with instruct^{n} format & instruct^{n} set are shown below:

* Our hypothetical machine:

(i) It is a 3-address machine with the following format

OP destination, source1, source2

NOTE: A 3 address instruct^{n} can have 2 operands or 1 operands also but it can have max of 3 operands

(ii) The target machine is byte addressable i.e., it can access 8 bit of info from specific address

(iii) It has n no of registers denoted by $R_0, R_1, R_2, \ldots, R_{n-1}$

* Various types of instruct^{n} that are used by target m/c
  (i) Load Instruct^{n}
  (ii) Store Instruct^{n}
  (iii) Computational Instruct^{n}
  (iv) Unconditional Instruct^{n}
  (v) Conditional Instruct^{n}

(i) Load Instruct^{n}: Used to copy the data into distinct^{n} operand which must be a register.

SYNTAX: LD_{45} dst, addr

where addr operand → register or memory loc

(ii) Store instruction: used to copy the data into mem locatn specified in the destinatn operand.

SYNTAX: ST dst, sr

where dst → destination if it is a mem location

sr → register.

(iii) Arithmetic or computational operation instruction. They are performed using this instruction.

SYNTAX: OP dst, src1, src2

where 1st operand, dst → destination

2nd & 3rd operand → Operands where R values fetched for operatn to be p

Eg1: ADD R0, R1, R2 // R0 = R1 + R2

Eg2: SUB R0, R0, R1 // R0 = R0 - R1

Eg3: MUL R2, R0, R1 // R2 = R0 * R1

(iv) Unconditional Jumps: The branch instructn without any conditn are called unconditional jumps.

SYNTAX: BR label

where BR → BRanch instructn

(v) Conditional Jumps: Based on the value stored in a register i.e., whether it is +ve or zero or -ve, if branching takes place, then the branch instn are called Conditional jumps.

SYNTAX: Bcond sr, label

where B stands from Branch,

Cond can be LT, GT, LTE, GTE
less than, greater than, less than or equal, greater than or equal

R → register, contains value such as 0, +ve or -ve.

Eg1: BL R0, T1 // Branch to T1, if R0 contains -ve value

Eg2: BLTZ R1, TR // Branch to TR, if R1 contains either 0 or -ve value

* Diffrunt addressing modes supported by generalized target machine:

1) Direct addressing mode
2) Indexed —11—
3) Intiger Indexed —11—
4) Indirect —11—
5) Immediate —11—

(i) Direct A/M :

Address of the data to be accessed is directly present in the instruct^n, i.e., location is identified by a variable name x.

Eg: ~~LDP~~ LD R1, x // Load value stored in memory locat^n x into R1

(ii) Indexed A/M: The data can be accessed from a memory locat^n using index. This addressing mode is useful for accessing arrays, where a is the base address of the array & register holds the index value

Eg: LD R1, a(R2) // Accesses the data stored in R1 = Contents (a + contents (R2))

(iii) Indexed A/M where memory locat^n is intiger

It is same as previous one except that a memory locat^n is identified as intiger.

Eg: LD R1, 100(R2) // R1 = contents (100 + contents (R2))

(v) Undirect A/M : Contents of the data can be accessed by dereferencing using * operators as shown below:

LD R1, *(R2)	// R2 contains memory location the data stored in that memory location is copied in register R1.
LD R1, *100(R2)	// R1 = contents (contents(100+ contents(R

(iv) Immediate A/M : The data to be manipulated is directly present in the instruction & preceded by

LD R1, #100	// R1 ← 100

EXERCISE :

1. Generate code for 3 address statement for x = y - z

LD R1, y	// R1 = y
LD R2, z	// R2 = z
ADD R1, R1, R2	// R1 = R1 + R2
ST x, R1	// x = R1

2. Generate code for 3 address statement x = * p

LD R1, p	// R1 ⇐ p
LD R2, 0(R1)	// R2 = contents (0 + cont
ST x, R2	// x = R2

(iv) Indirect A/M: Contents of the data can be accessed by dereferencing using * operators as shown below:

```
LD   R1, *(R2)
```
// R2 = contains memory location the data stored in that memory location is copied in register R1.

```
LD   R1, *100(R2)
```
// R1 = contents(contents(100 + contents(R

(v) Immediate A/M: The data to be manipulated is directly present in the instruction & preceded by

```
LD   R1, #100
```
// R1 ← 100

EXERCISE:

1. Generate 3 address code for statement for $x = y - z$

```
LD   R1, y      // R1 = y
LD   R2, z      // R2 = z
ADD  R1, R1, R2 // R1 = R1 + R2
ST   x, R1      // x = R1
```

2. Generate 3 address code for statement $x = *p$

```
LD   R1, p      // R1 ⇐ p
LD   R2, 0(R1)  // R2 = contents(0 + cont
ST   x, R2      // x = R2
```

3. Generate code for 3 address statement *p = y

```
LD   R1, p          // R1 = p
LD   R2, y          // R2 = y
ST   0(R1), R2      // contents(0+contents(R1)) = R2
```

4. Generate m/c code for 3 address statement b = a[i]

```
LD   R1, i          // R1 = i
MUL  R1, R1, 8      // R1 = R1 * 8
LD   R2, a[R1]      // R2 = contents(a + contents(R1))
ST   b, R2          // b = R2
```

5. Generate m/c code for 3 address statement a[j] = c

```
LD   R1, j          // R1 = j
LD   R2, c          // R2 = c
MUL  R1, R1, 8      // R1 = R1 * c
ST   a[R1], R2      // contents(a + contents(R1)) = R2
```

6. Generate m/c code for 3 address statement

if x < y goto L

```
LD   R1, x          // R1 = x
LD   R2, y          // R2 = y
SUB  R1, R1, R2     // R1 = R1 - R2
BNZ  R1, M          // if R1 < 0 jump to M
```

Program & Instruction Cost

* For simplicity we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.

* A/M involves registers have zero additional cost.
* A/M involving memory locatn or constant have additional cost of 1.
* For example:

a) LD R0, R1 ⇒ cost = 1
b) LD R0, M ⇒ cost = 2.
c) LD R1, *100(R2) ⇒ cost = 3

* Cost of Addressing mode:

	Mode	Form	Address	Added Cost
①	Absolute direct A/M	M	M	1
②	Register direct A/M	R	R	0
③	Indexed A/M	C(R)	C+contents(R)	1
④	Indirect register A/M	*R	Contents(R)	0
⑤	Indirect indexed A/M	*C(R)	contents(C+contents(R))	1
⑥	Immediate A/M	#C	N/A	1

NOTE: Cost of each statement = 1 + cost (Addressing mode)

EXERCISES (8.2)

1. Determine the costs of the following instruct^{n} sequence

LD R0, y	Cost = 1 + cost(AM)
LD R1, z	Cost = 1 + 1 = 2
ADD R0, R0, R1	Cost = 1 + 1 = 2
ST x, R0	Cost = 1 + 0 = 1
	Cost = 1 + 1 = 2
	Total Cost = 7

2.
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

	Cost = Cost (AM) + 1.
LD R0, i	Cost = 1 + 1 = 2
MUL R0, R0, 8	Cost = 1 + 1 = 2
LD R1, a(R0)	Cost = 1 + 1 = 2.
ST b, R1	Cost = 1 + 1 = 2
	Total cost = 8

3.
```
LD  R0, C
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

	Cost = cost(A.M) + 1
LD R0, C	1 + 1 = 2
LD R1, i	1 + 1 = 2
MUL R1, R1, 8	1 + 1 = 2
ST a(R1), R0	1 + 1 = 2
	Total cost = 8

4. LD R0, p
LD R1, 0(R0)
ST x, R1

	cost = cost(A.M) + 1
LD R0, p	1 + 1 = 2
LD R1, 0(R0)	1 + 1 = 2. // ∵ of constar
ST x, R1	1 + 1 = 2
Total cost = 6	

5. LD R0, P
LD R1, x
ST 0(R0), R1

	cost = cost(A.M) + 1
LD R0, P	1 + 1 = 2
LD R1, x	1 + 1 = 2
ST 0(R1), R1	1 + 1 = 2
Total cost = 6	

6. LD R0, x
LD R1, y
SUB R0, R0, R1
BLTZ *R3, R0

	cost = 1 + cost(A.M)
LD R0, x	1 + 1 = 2
LD R1, y	1 + 1 = 2
SUB R0, R0, R1	1 + 0 = 1
BLTZ *R3, R0	1 + 1 = 2 <∵ indirect A
Total cost = 7	