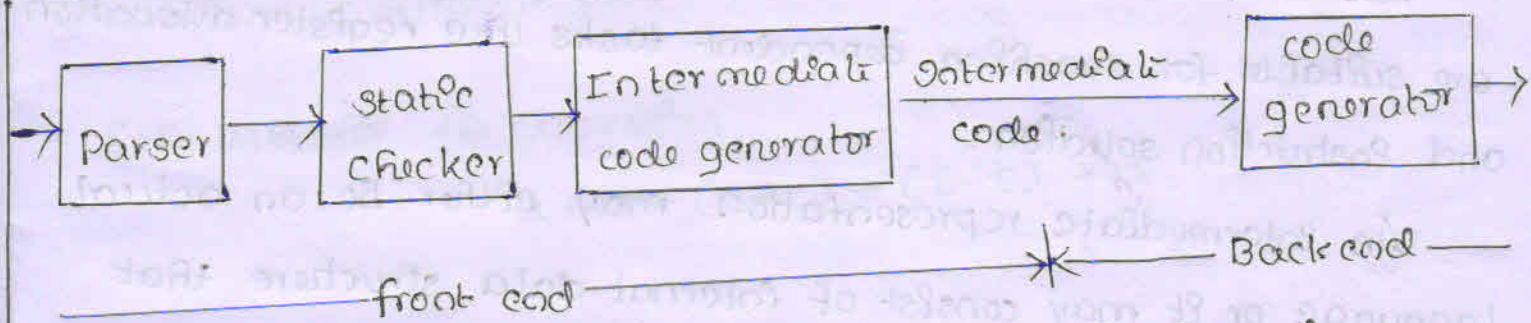
UNIT-6

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.



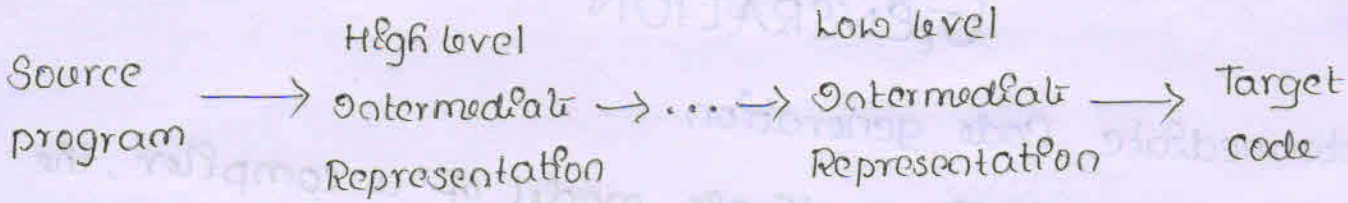
Logical structure of a compiler front end

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~~ assures 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.

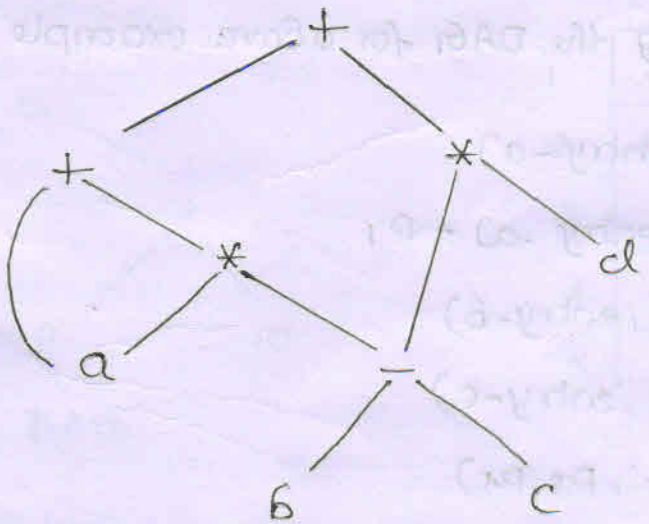
On DAG leaves represents the atomic operands and interior nodes represents the operators. as in the syntax tree.

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
(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)

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.

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

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

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

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

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

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

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

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

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

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

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

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

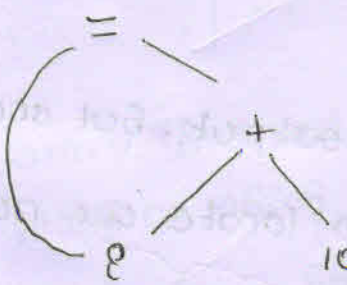
(xiv) P_{13} = Node ('+', P_7, P_{12})

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

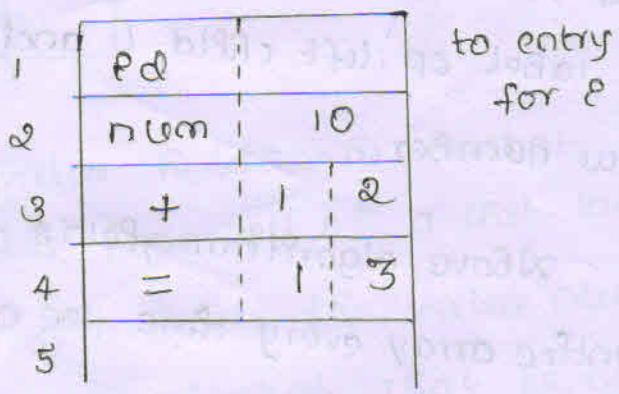
The Value-Number Method for Constructing DAG's

* 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, label ing 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 $i = i + 10$ allocated in an Array



DAG



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.

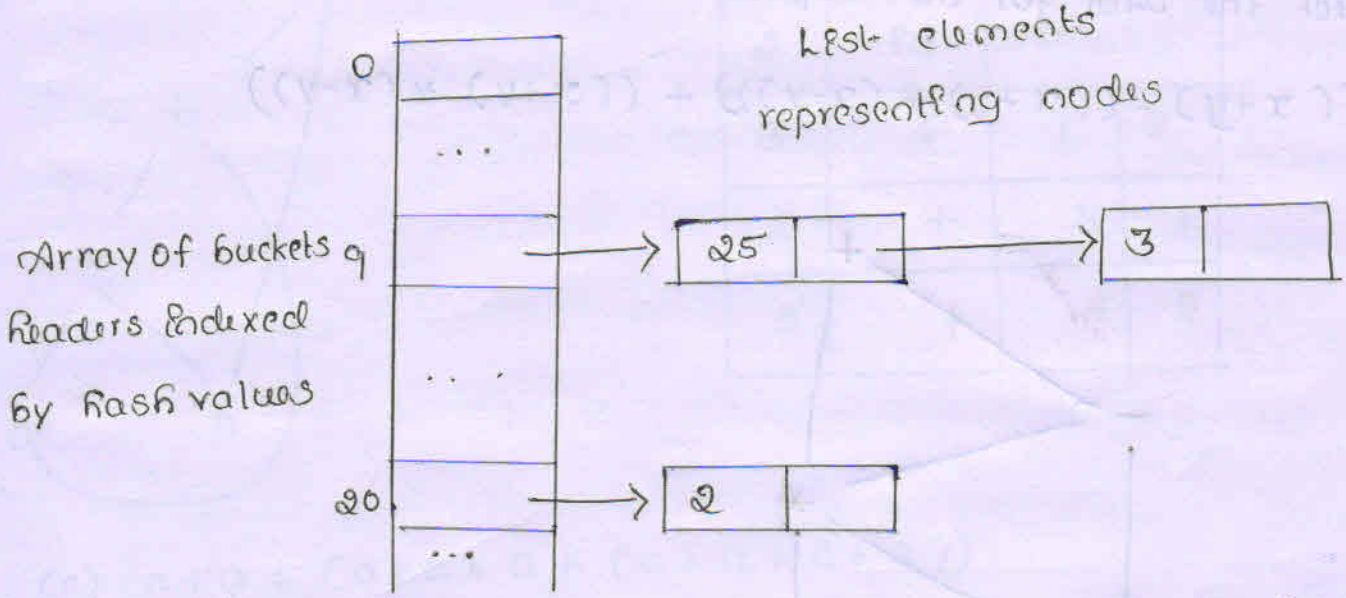
Above algorithm yields the desired output, but searching the entire array every time we are asked to locate one node is expensive.

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 readers, each of which points to the first cell of a list. With 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 reader 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.

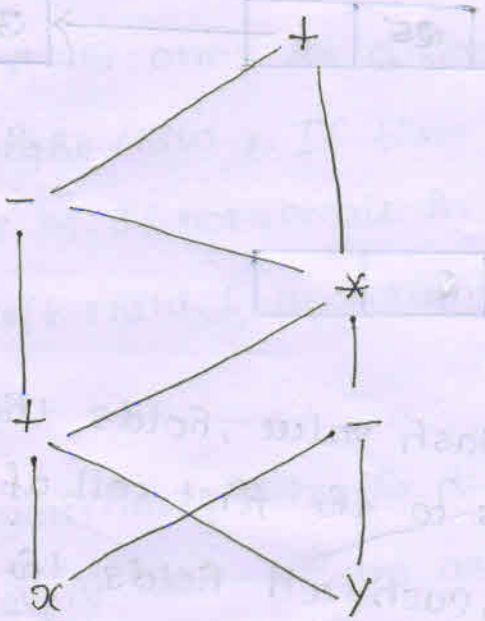
For each value number 'v' found in a cell, we must

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.

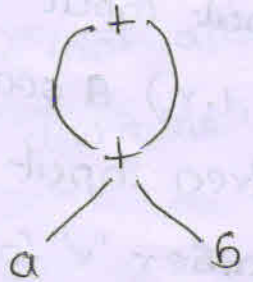
Construct the DAG for the expression.

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



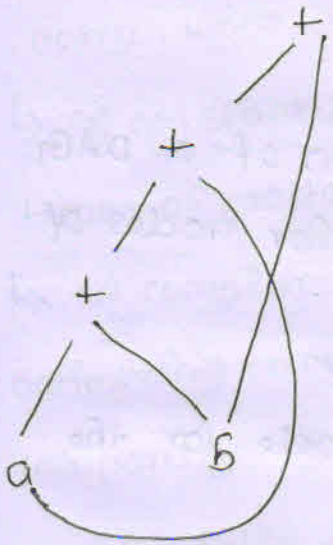
Construct the DAG & identify the value number for the sub expressions of the following expressions, assuming + associa from the left.

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



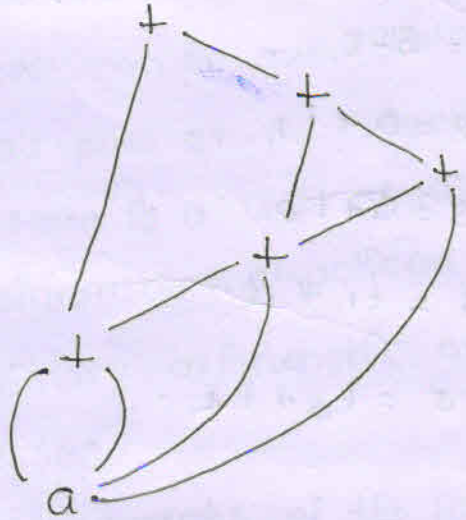
1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

(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 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

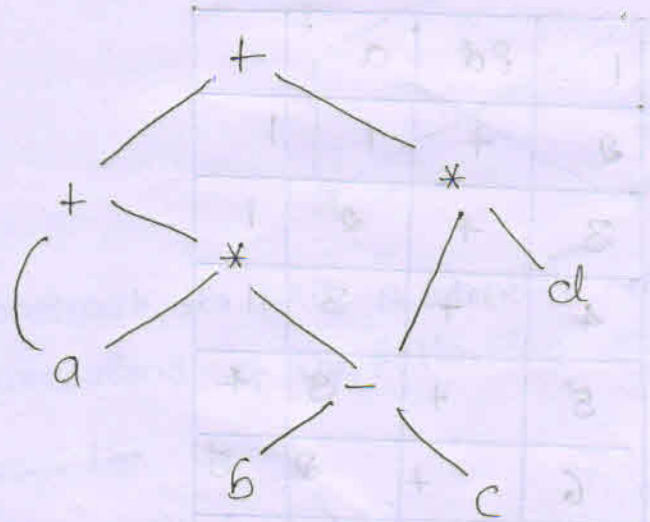
$t_1 = y * z$

$t_2 = x + t_1$

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] = y 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 : t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a[t2]
    if t3 < v goto L
```

(a) Symbolic labels

```
100 : t1 = i + 1
101 : i = t1
102 : t2 = i * 8
103 : t3 = a[t2]
104 : if t3 < v goto 100
```

(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 arg_2. 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 arg_2 nor result.

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

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

$t_1 = minus\ c$

$t_2 = b * t_1$

$t_3 = minus\ c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

(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

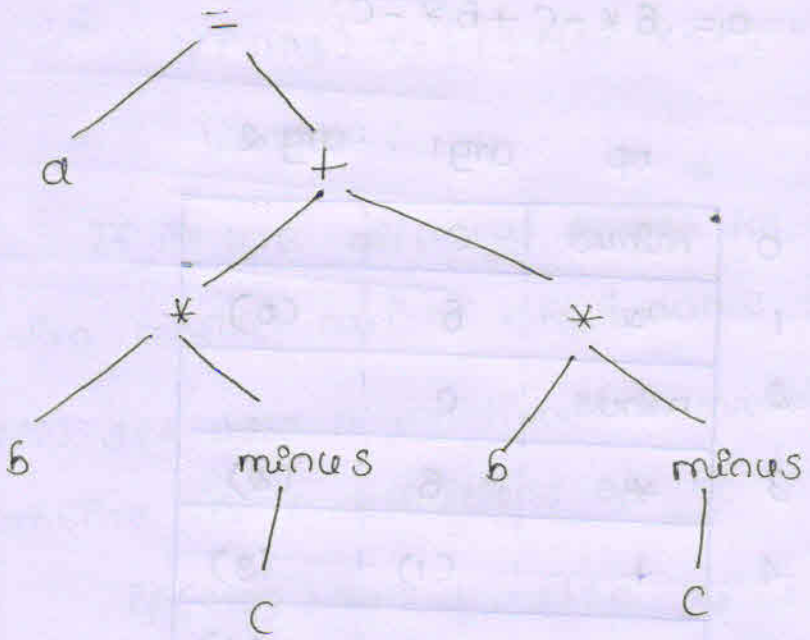
(b) Quadruples.

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	arg 1	arg 2
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: $p = a + b$
$q = p - c$
$p = q * d$
$p = e - p$
$q = p + q$

3-address code

$p_1 = a + b$
$q_1 = p_1 - c$
$p_2 = q_1 * d$
$p_3 = e - p_2$
$q_2 = p_3 + q_1$.

static single assignment form

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 $y = x * a$.

(SSA) 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

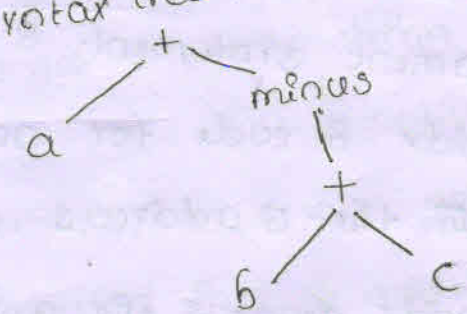
(a) A syntax tree

(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	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 at most 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 & attributes 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

that will hold the value of E.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E;$	S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
| id	E.addr = top.get(id.lexeme) E.code = ''

Consider $E \rightarrow id$, when an expression is a single identifier say x then x itself holds the value of the expression. The semantic rules E.addr point to the symbol table entry. If top denote the current symbol table then function top.get retrieves the entry when it is applied to the string representation id.lexeme E.code is set to empty string.

For $E \rightarrow E_1 + E_2$, compute value of E from E_1 & E_2. Values are computed into newly generated temporary names. If E_1 is computed into E_1.addr & E_2 into E_2.addr then $E_1 + E_2$

translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. E.addr is set to t.

The notation gen ('x' '=' 'y' '+' 'z') is used to represent the 3 address instruction $x = y + z$. Variables like x, y & z are evaluated & when passed to gen & quoted strings like '=' are taken literally.

E.code is build up by concatenating $E_1.code$, $E_2.code$ & an instruction that adds the values of E_1 & E_2. The instruction puts the result of the addition into a new temporary names for E, denoted by E.addr.

The production $S \rightarrow id = E$; generates instructions that assign the value of expression E to the identifier id. The semantic rule for this production uses function top.get to determine the address of the identifier represented by id.

S.code consists of the instructions to compute the value of E into an address given by E.addr, followed by an assignment to the address top.get (id.lexeme') for this instance of id.

Incremental Translation.

In incremental translation, instead of building up E.code we can arrange to generate only the new 3-address instruction In the incremental approach, gen not only constructs a 3 address instruction, it appends the instruction to the sequence of instruction generated so far.

With the incremental approach, the code attribute is not used, since there is a single sequence of instructions that is created by successive calls to gen.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$	{ gen (top.get (id.lexeme) '=' E.addr); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp ( ) ; gen (E.addr '=' E_1.addr '+' E_2.addr); }
$\mid -E_1$	{ E.addr = new Temp ( ); gen (E.addr '=' 'minus' E_1.addr); }
$\mid (E_1)$	{ E.addr = E_1.addr ; }
$\mid id$	{ E.addr = top.get (id.lexeme) ; }

The semantic rule for $E \rightarrow E_1 + E_2$ simply calls gen to generate an add instruction the instructions to compute E_1 into E_1.addr & E_2 into E_2.addr have already been generated.

The new semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$E \rightarrow E_1 + E_2$ { E.addr = new Node ('+', E_1.addr, E_2.addr); }

Here, attribute addr represents the address of a node rather than a variable or constant.

Addressing array elements.

Array elements can be accessed quickly if they are stored in a block of consecutive locations. If the width of each array element is w, then the i^{th} element of array A begins in location

$$base + i * w \quad — (1)$$

Base is the relative address of A[0].

The formula (1) generalizes to 2 or more dimensions. In 2 dimension, A[l_1][l_2], let w_1 be the width of a row & let w_2 be the width of an element in a row. The relative address of A[i][j] can then be calculated by

$$base + l_1 * w_1 + l_2 * w_2 \quad — (2)$$

In k dimensions, the formula is,

$$base + l_1 * w_1 + l_2 * w_2 + \ldots + l_k * w_k \quad — (3)$$

The relative address can be calculated in terms of no. of elements of along dimension j of the array and the width $w = w_k$ of a single element of the array.

In 2 dimensions, the location of A[l_1][l_2] is given by

$$base + (l_1 * n_2 + l_2) * w \quad — (4)$$

In k dimensions,

$$base + ((\ldots((l_1 * n_2 + l_2) * n_3 + l_3)\ldots) * n_k + l_k) * w$$

The array elements need not be numbered starting at 0. In a one-dimensional array, the array elements are numbered low, low+1 ... high & base is the relative address. (1) can be replaced as

$$base + (i - low) * w \quad — (5)$$

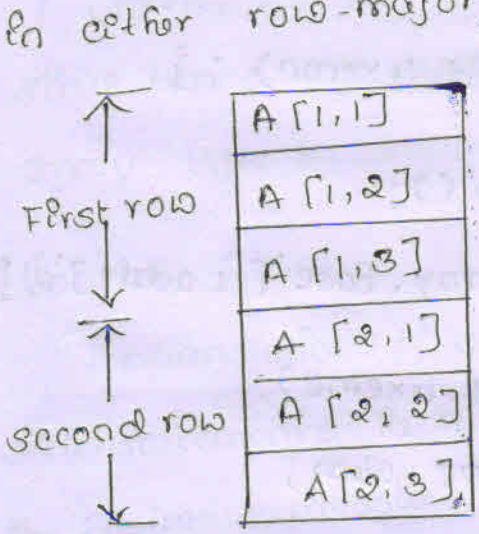
Both (1) & (5) can be rewritten as

$i * w + c$ where,

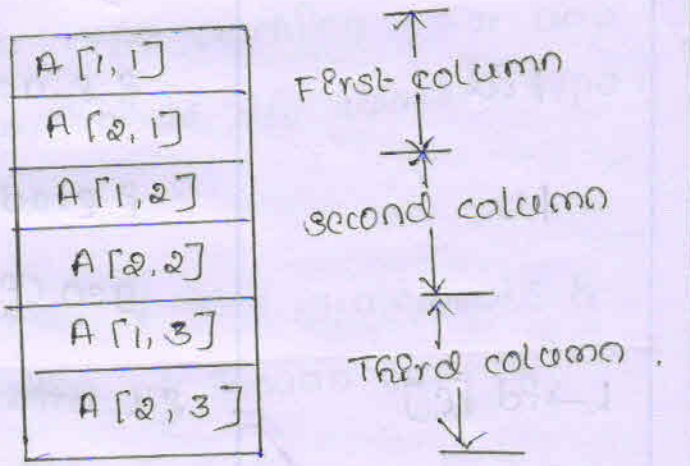
$c = base - low * w$ calculated at compile time

Compile time precalculation can not be used when the array's size is dynamic. If we don't know the values of low & high at compile time, then we can't compute constants such as c

The above address calculations are based on row-major layout for arrays. A 2 dimensional array is normally stored in either row-major or column major forms.



(a) Row major



(b) Column major

We can generalize row or column major form to many dimensions. The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest, like the numbers on an odometer. Column-major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest.

Translation of Array References.

Let nonterminal L generate an array name followed by a sequence of index expressions

$$L \rightarrow L[E] \mid id[E]$$

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E ;$	{ gen (top.get(id.lexeme) '=' E.addr); }
$\mid L = E ;$	{ gen (L.array.base '[' L.addr ']' '=' E.addr
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp ( ) ;
	gen (E.addr '=' E_1.addr '+' E_2.addr); }
$\mid id$	{ E.addr = top.get(id.lexeme) ; }
$\mid L$	{ E.addr = new Temp ( );
	gen (E.addr '=' L.array.base '[' L.addr ']');
$L \rightarrow id [E]$	{ L.array = top.get (id.lexeme);
	L.type = L.array.type.elem ;
	L.addr = new Temp( );
	gen (L.addr '=' E.addr '*' L.type.width); }
$\mid L_1 [E]$	{ L.array = L_1.array ;
	L.type = L_1.type.elem ;
	t = new Temp( );
	L.addr = new Temp( );
	gen (t '=' E.addr '*' L.type.width);
	gen (L.addr '=' L_1.addr '+' t); }

~~Note~~: Nonterminal L has 3 synthesized attributes

(i) L.addr denotes a temporary that is used while computing the offset for the array references by summing the terms ($i_3 * w_3$)

(ii) L.array is a pointer to the symbol table entry name. The base address of the array, say, L.array.base is used to determine the actual l-value of an array reference after all the index expressions are analyzed.

(iii) L.type is the type of the subarray generated by L. For any type t, we assume that its width is given by t.width. We use types as attributes, rather than widths, since types are needed anyway for type checking. For any array type t, suppose that t.elem gives the element type.

Control Flow

The translation of statements such as if-else-statements & while-statements is tied to the translation of boolean expression. The boolean expression are often used to

(i) Alter the flow of control

(ii) Compute logical values

Boolean Expressions.

Boolean expressions are composed of the boolean operators applied to elements that are boolean variables or relational expression.

Relational expression are of the form E_1 rel E_2 where E_1 & E_2 are arithmetic expression & rel.op is used to indicate which of the 6 comparison operators $<, <=, =, !=, >$ or $>=$ is represented by rel.

The semantic definition of the programming language determines whether all part of a boolean expression must be evaluated.

Ex: In expression $B_1 || B_2$, if B_1 is true then the expression is true.

In expression $B_1 \&\& B_2$ if B_1 is false then the entire expression is false.

Short-Circuit code

In short circuit (or jumping) code, the boolean operators && , || & ! translate into jumps. The operators themselves do not appear in the code, instead, the value of a boolean expression is represented by a position in the code sequence

Ex: if ($x < 100$ || $x > 200$ && $x != y$) $x = 0$;

The statement might be translated into the code

```
    if x < 100 goto L2
    if false x > 200 goto L1
    if false x != y goto L1
L2 : x = 0
L1 :
```

The boolean expression is true if control reaches label

Flow-of-control Statements

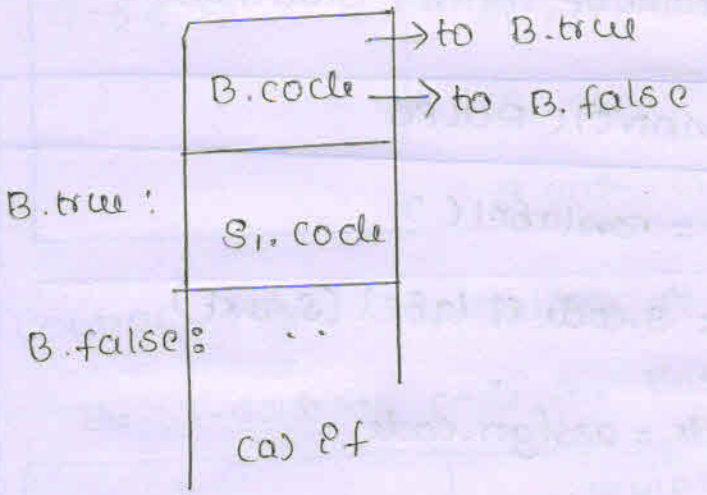
Consider the grammar

$S \rightarrow$ if (B) S_1

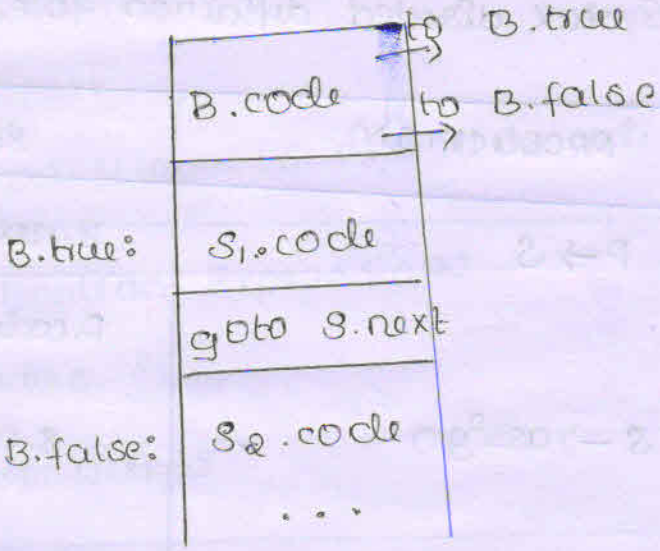
$S \rightarrow$ if (B) S_1 else S_2

$S \rightarrow$ while (B) S_1

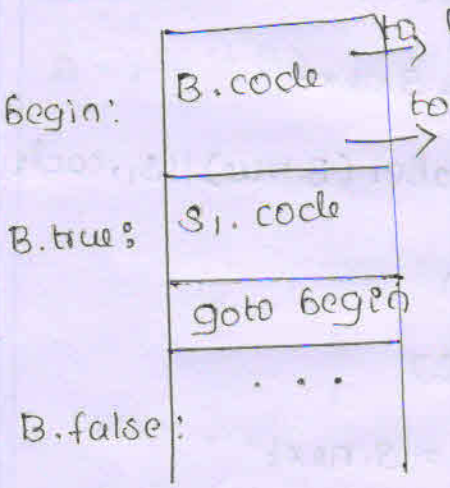
On these productions, nonterminal B represents a boolean expression & non terminal S represents a statement.



(a) If



(b) If-else



(c) while.

The translation of If (B) S_1 consists of B.code followed by S_1.code. Within B.code are jumps based on the value of B. If B is true, control flows to the first instruction of S_1.code & if B is false, control flows to the instruction immediately following S_1.code

With boolean expression B, we associate 2 labels

(i) B.true, the label to which control flows if B is true

(ii) B.false, the label to which control flows if B is false

With statement S, we associate an inherited attribute

S.next denoting a label for the instruction immediately after the code for S

Syntax directed definition for flow of control statement.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	S.next = newlabel() P.code = S.code || label(S.next)
$S \rightarrow assign$	S.code = assign.code
$S \rightarrow if(B)\ S_1$	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$S \rightarrow if\ (B)\ S_1\ else\ S_2$	B.true = newlabel() B.false = newlabel() S_1.next = S_2.next = S.next S.code = B.code || label(B.true) || S_1.code || gen('goto' S.next) || label(B.false) || S_2.code
$S \rightarrow while\ (B)\ S_1$	begin = newlabel() B.true = newlabel() B.false = S.next S_1.next = begin S.code = label(begin) || B.code

	|| label (B.true) || S_1.code
	|| gen ('goto' begin)
$S \rightarrow S_1 S_2$	S_1.next = newlabel()
	S_2.next = S.next
	S.code = S_1.code || label (S_1.next) || S_2.code

Control - Flow Translation of Boolean expressions

The 3-address code for Boolean es,

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \,\|\|\, B_2$	B_1.true = B.true
	B_1.false = newlabel()
	B_2.true = B.true
	B_2.false = B.false
	B.code = B_1.code || label (B_1.false) || B_2.code
$B \rightarrow B_1 \,\&\&\, B_2$	B_1.true = newlabel()
	B_1.false = B.false
	B_2.true = B.true
	B_2.false = B.false
	B.code = B_1.code || label (B_1.true) || B_2.code
$B \rightarrow ! B_1$	B_1.true = B.false
	B_1.false = B.true
	B.code = B_1.code
$B \rightarrow E_1$ rel E_2	B.code = E_1.code || E_2.code

	E_1.addr rel.op E_2.addr 'goto' B.true)
	gen ('goto' B.false)
$B \rightarrow true$	B.code = gen ('goto' B.true)
$B \rightarrow false$	B.code = gen ('goto' B.false)

$B \rightarrow E_1$ rel E_2 is translated directly into a comparison 3-address instruction with jumps to the appropriate place.

Ex: If B of the form a<b

if a<b goto B.true

goto B.false.

The remaining production for B translated as

(i) Suppose B is B_1 || B_2. If B_1 is true, then B itself is true, B_1.true is same as B.true. If B_1 is false then B_2 must be valuated, make B_1.false be the label of the first instruction in the code for B_2.

(ii) The translation of B_1 && B_2 is similar.

(iii) No code is needed for an expression B of the form ! B_1; just interchange the true & false exits of B to get the true & false exits of B_1

(iv) The constants true & false translate into jumps to B.true & B.false respectively.

Ex: consider if (x<100 || x>200 && x != y) x=0;

```
        if x < 100 goto L2
        goto L3
L3 :    if x > 200 goto L4
        goto L1
L4 :    if x != y goto L2
        goto L1
L2 :    x = 0
L1 :
```

control flow translation of a simple-if-statement

Avoiding Redundant Gotos

Of the code is,

```
if ( x > 200 goto L4
goto L1
L4 : . . .
```

Instead, consider the instruction

```
iffalse x > 200 goto L1
L4 : . . .
```

This iffalse instruction takes advantage of natural flow from one instruction to the next in sequence, so control simply "falls through" to label L4 if x > 200, thereby avoiding jump. In if-while statements we can use special label fall. The new rules for S → if (B) S1 set B.true to fall

B.true = fall

B.false = S1.next = S2.next

S.code = B.code || S_1.code .

The semantic rules for $B \rightarrow E_1$ rel E_2 is

test = E_1.addr rel.op E_2.addr

S = if B.true ≠ fall and B.false ≠ fall then

gen ('if' test 'goto' B.true) || gen ('goto' B.false)

else if B.true ≠ fall then gen ('if' test 'goto' B.true)

else if B.false ≠ fall then gen ('ifFalse' test 'goto' B.false)

else ''

B.code = E_1.code || E_2.code || S

Of both B.true & B.false are explicit labels i.e neither equal to fall.

If B.true is an explicit label, then B.false must be fall, so they create an if instruction that lets control fall through if the condition is false.

If B.false is an explicit label then they generate an ifFalse instruction.

The semantic rule for $B \rightarrow B_1 \,||\, B_2$ is,

B_1.true = if B.true ≠ fall then B.true else newlabel()

B_1.false = fall

B_2.false = B.true

B_2.false = B.false

B.code = if B.true ≠ fall then B_1.code || B_2.code

else B_1.code || B_2.code || label (B_1.true)

Here note that the meaning of label fall for B is different from its meaning for B_1. Suppose B.true is fall i.e control ~~flows~~ falls through B, if B evaluates to true. Although B evaluates to true if B_1 does, B_1.true must ensure that control jumps over the code for B_2 to get to the next instruction after B.

On the other hand, if B_1 evaluates to false, the truth value of B is determined by the value of B_2.

Ex: If (x<100 || x>200 && x!=y) x=0;

translates into.

```
    if x<100 goto L2
    iffalse x>200 goto L1
    iffalse x!=y goto L1
L2: x=0
L1:
```

Boolean values and Jumping code.

A boolean expression may also be evaluated for its value, as in assignment statement such as x=true or

x = a<b;

A clean way of handling both roles of boolean expressions is to first build a syntax tree for expression, using either of the following approaches.

(1) use 2 passes. Construct a complete syntax tree for the input & then walk the tree in depth first order, computing

the translations specified by the semantic rules.

(ii) Use one pass for statements, but 2 pass for expressions. With this approach, we would translate E in while (E) S_1 before S_1 is examined.

Consider

$S \rightarrow id = E ; \mid if (E) S \mid while (E) S \mid SS$

$E \rightarrow E \| E \mid E \&\& E \mid E \text{ rel } E \mid E + E \mid (E) \mid id \mid true \mid fal$

Nonterminal E governs the flow of control in $S \rightarrow while (E) S_1$. The same E denotes a value $S \rightarrow id = E$ and $E \rightarrow E + E$.

Suppose E.n denotes the syntax tree node for an expression E. Let method jump generate jumping code at an expression node & rvalue generate code to compute the value of the node into a temporary.

When E appears in $S \rightarrow while (E) S_1$, method jump is called at node E.n. Jumping code is generated by calling E.n.jump(t, f) where t is a new label for S_1.code & f is the label S.next

When E appears in $S \rightarrow id = E ;$ method rvalue is called at node E.n. If E has the form $E_1 + E_2$ the method call E.n.rvalue() generates code. If E has the form $E_1 \&\& E_2$ we first generate jumping code for E & then assign true or false to a new temporary t at the true & false exits respectively from the jumping code.

Ex: the assignment $x = a < b \;\&\&\; c < d$ can be implemented by the code

```
    if False a<b goto L1
    if False c<d goto L1
    t = true
    goto L2
L1 : t = false
L2 : x = t.
```

Backpatching

A key problem when generating code for boolean expressions & flow of control statement is that matching a jump instruction with the target of the jump.

This can be solved by an approach, called backpatching, in which lists 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 the proper label can be determined. All the jumps on a list have the same target label.

One-pass code Generation using Backpatching

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

Synthesized attributes truelist & falselist of nonterminal B are used to manage labels in jumping code for boolean expression.

B.truelist → list of jump or conditional jump instructio into which we must insert the label to which control goes if B is true.

B.falselist → list of instructions that eventually get the label to which control goes when B is false.

As code is generated for B, jumps to the true & false exit are left incomplete, with the label field unfilled.

S has a synthesized attribute S.nextlist, denoting a list to jumps to the instruction immediately following the code for S.

To manipulate lists of jumps, we use 3 functions

(i) makelist (i) creates a new list containg only i, an index into the array of instructions, makelist returns a pointer to the newly created list.

(ii) merge (p_1, p_2) concatenates the lists pointed to by p_1 & p_2 & returns a pointer to the concatenated list.

(iii) backpatch (p, i) inserts i as the target label for each of the instruction on the list pointed to by p.

Backpatching for Boolean expressions

We can construct a translation scheme suitable for generating code for boolean expression during bottom-up parsing. A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate time, the index of the next instruction to be generated. The grammar is,

$B \rightarrow B_1 \,||\, M\, B_2 \mid B_1 \,\&\&\, M\, B_2 \mid !\, B_1 \mid (B_1) \mid E_1\ rel\ E_2 \mid true \mid fal$

M → ∈

The translation scheme is,

(i) $B \rightarrow B_1 \mid\mid M B_2$	{ backpatch (B_1.falselist, M.instr); B.truelist = merge (B_1.truelist, B_2.truelist); B.falselist = B_2.falselist; }
(ii) $B \rightarrow B_1$ && $M B_2$	{ backpatch (B_1.truelist, M.instr); B.truelist = B_2.truelist; B.falselist = merge (B_1.falselist, B_2.falselist); }
(iii) $B \rightarrow ! B_1$	{ B.truelist = B_1.falselist; B.falselist = B_1.truelist; }
(iv) $B \rightarrow (B_1)$	{ B.truelist = B_1.truelist; B.falselist = B_1.falselist; }
(v) $B \rightarrow E_1$ rel E_2	{ B.truelist = makelist (nextinstr); B.falselist = makelist (nextinstr + 1); gen ('if' E_1.addr rel.op E_2.addr 'goto _'); gen ('goto _'); }
(vi) B → true	{ B.truelist = makelist (nextinstr); gen ('goto _'); }
(vii) B → false	{ B.falselist = makelist (nextinstr); gen ('goto _'); }
(viii) M → ∈	{ M.instr = nextinstr; }

Consider (e) $B \rightarrow B_1 \,||\, M\, B_2$. If B_1 is true, then B is also true, so B_1.truelist becomes part of B.truelist. If B_1 is false, test B_2, so the target for the jumps B_1.falselist must be beginning of the code generated for B_2. This target is obtained by M. It produces synthesized attribute M.instr, the index of the next instruction, just before B_2 code starts being generated.

To obtain that instruction index, we associate with the production $M \rightarrow \epsilon$

{ M.instr = nextinstr ; }

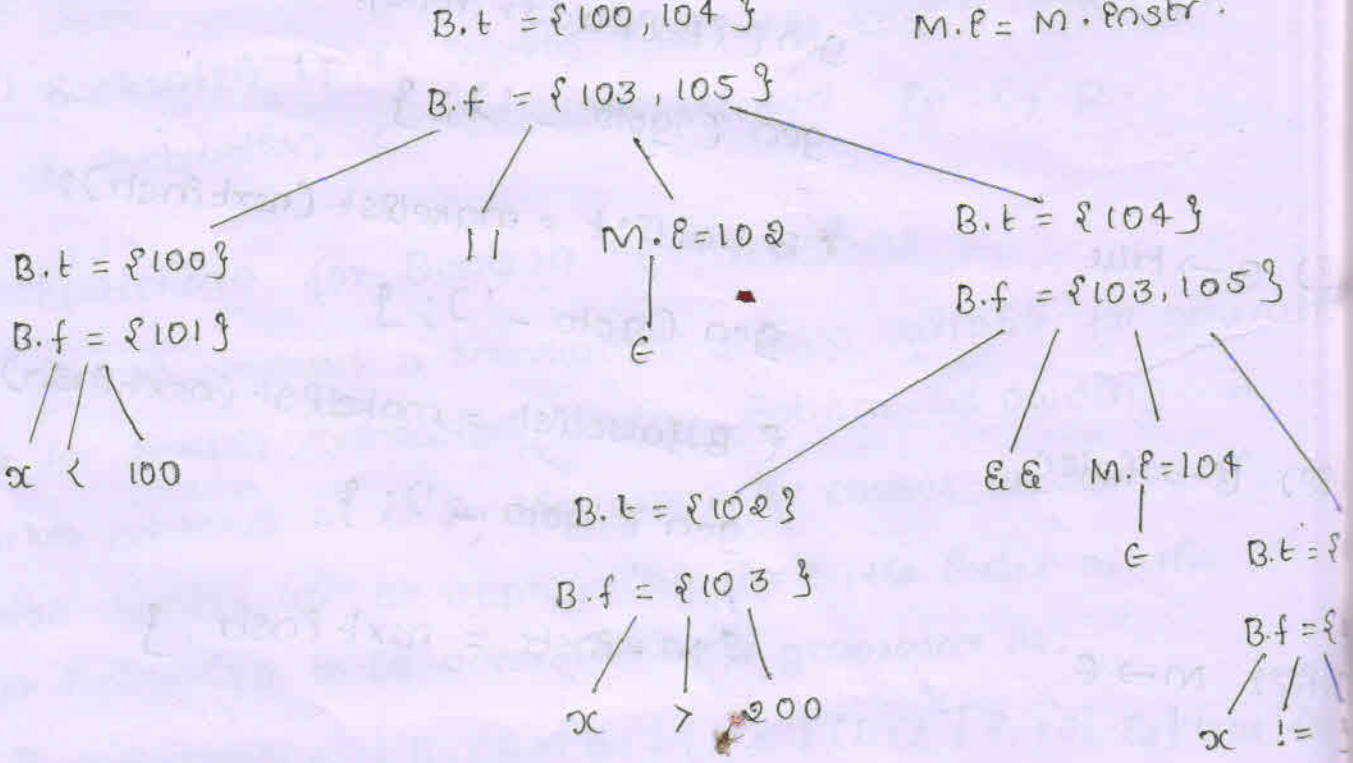
The variable nextinstr holds the index of the next instruction to follow. this value will be backpatched onto B_1.falselist when we have seen the remainder of the production $B \rightarrow B_1 \,||\, M$

Example :

Consider the expression $x < 100 \,||\, x > 200 \;\&\&\; x != y$

The annotated parse tree is,

B.t = B.truelist
B.f = B.falselist
M.i = M.instr.



The reduction of x<100 to B by production (v), the 2 instruction are generated

100: if x<100 goto —

101: goto —

The marker nonterminal M in the production

$B \rightarrow B_1$ || M B 2

records the value of nextinstr, which at this time is 102. The reduction of x>200 to B by production (vi) generate the instructions

102: if x>200 goto —

103: goto —

The sub expression x>200 correspond to B_1 in the production $B \rightarrow B_1$ && MB2.

The marker nonterminal M records the correct value of nextinstr, which is now 104. Reducing x!=y into B by production (v) generates

104: if x!=y goto —

105: goto —

We now reduce by $B \rightarrow B_1$ && MB2. The corresponding semantic action calls backpatch (B_1.truelist, M.instr) to bind the true exit of B_1 to the first instruction of B2. Since B_1.truelist is {102} & M.instr is 104 this call to backpatch fills in 104 in instruction 102. The six instructions generated so far are

100: if x<100 goto —

101 : goto —

102 : if x > 200 goto 104

103 : goto —

104 : if x != y goto —

105 : goto —

The semantic action associated with the final production by $B \rightarrow B_1 \; M \; B_2$ calls backpatch ({101}, 102) which leaves the instruction as

100 : if x < 100 goto —

101 : goto 102

102 : if x > 200 goto 104

103 : goto —

104 : if x != y goto —

105 : goto —

The entire expression is true iff gotos of instructions 100 or 104 are reached, & is false iff gotos of instructions 103 or 105 are reached.

Flow of control statements

We can use backpatching to translate flow of control statements in one pass. Consider

$S \rightarrow$ if (B) S | if (B) S else S | while (B) S | {L} | A ;

$L \rightarrow L \; S \mid S$

where S → statement L → statement list

A → assignment statement B → Boolean expression

The translation scheme is,

(i) S → if (B) M S_1 { backpatch (B.truelist, M.instr);
S.nextlist = merge (B.falselist, S_1.nextlist); }

(ii) S → if (B) M S_1 N else M S_2
{ backpatch (B.truelist, M_1.instr);
backpatch (B.falselist, M_2.instr);
temp = merge (S_1.nextlist, N.nextlist);
S.nextlist = merge (temp, S_2.nextlist); }

(iii) S → while M_1 (B) M S_1
{ backpatch (S_1.nextlist, M_1.instr);
backpatch (B.truelist, M_2.instr);
S.nextlist = B.falselist;
gen ('goto' M.instr); }

(iv) S → { L } { S.nextlist = L.nextlist; }

(v) S → A; { S.nextlist = null; }

(vi) M → ∈ { M.instr = nextinstr; }

(vii) N → ∈ { N.nextlist = makelist (nextinstr);
gen ('goto _'); }

(viii) L → L_1 M S { backpatch (L_1.nextlist, M.instr);
L.nextlist = S.nextlist; }

(ix) L → S { L.nextlist = S.nextlist; }

Consider the production $S \rightarrow$ while M_1 (B) M_2 S_1. The s occurences of marker nonterminal M in the production re the instruction numbers of the beginning of the code for B the beginning of the code for S_1. The production $M \rightarrow \epsilon$ set attribute M.instr to the no. of next instruction. After S_1 is executed, control flows to the beginning. Therefore, when reduce to S, we backpatch S_1.nextlist to make all targe on that list be M_1.instr. An explicit jump to the beginn of the code for B is appended after the code for S_1 because control may also "fall out the bottom". B.truelist is back to go to the beginning of S_1 by making jumps on B.true go to M_2.instr.

S.nextlist & L.nextlist comes when code is generated for If (B) S_1 else S_2. If control "falls out the bottom' of as when S_1 is an assignment, we must include at the end the code for S_1 a jump over the code for S_2. We use a marker nonterminal to generate this jump after S_1. i.e N

We backpatch the jumps when B is true to M_1.instr the latter is the beginning of the code for S_1. Similarly we backpatch jumps when B is false to go to the beginning of code for S_2. The list S.nextlist includes all jumps out of S_1 as well as the jump generated by N

Semantic actions (viii) & (ix) handles sequence of stat

$L \rightarrow L_1 M S$

In the instruction following the code for L_1 in order of exe is the beginning of S. Thus the L_1.nextlist list is backpa to the beginning of the code for S, which is given by M.inst

On $L \rightarrow S$, L.nextlist is same as S.nextlist.

Break-, Continue- & Goto-Statements.

Goto statements can be implemented by maintaining a list of unfilled jumps for each label and then backpatching the target when it is known.

The break statements sends control out of an enclosing loop & continue-statement triggers the next iteration of an enclosing loop.

Consider

(i) for(;;readch()) {

(ii) if(peek == ' ' || peek == '\t') continue;

(iii) else if(peek == '\n') line = line + 1;

(iv) else break;

(v) }

eg. Break statement on line (iv) make control to jump to the next statement after the enclosing for loop.

Continue in line (ii) makes control to jump to readch()

If S is the enclosing loop construct, then a break statement is a jump first instruction after the code for S. we can generate code for the break

(i) keeping track of the enclosing loop statement S

(ii) Generating an unfilled jump for the break statement

(iii) Putting this unfilled jump on S.nextlist

In 2 pass front end S.nextlist can be implemented as a field in the node for S. We can keep track of S by using the symbol table to map a special identifier break to the list for the enclosing loop statement S.

Instead of using the symbol table we can put pointer to S.nextlist in the symbol table. When a break statement is reached, we generate an unfilled jump, look up nextlist through the symbol table & add the jump to the list, where it will be backpatched.

Continue statement can be handled in a manner analogous to the break statement. The main difference b/w the 2 is the target of the generated jump is different.

** Switch - Statements

The switch statement syntax is,

```
switch (E) {
    case V1 : S1
    case V2 : S2
    ...
    case Vn-1 : Sn-1
    default : Sn
}
```

There is a selector expression E, which is to be evaluated followed by n constant values $V_1, V_2 \ldots V_{n-1}$ that the expression might take, including a default "value" which always matches the expression if no other value does.

Translation of Switch-statements.

The intended translation of a switch is code to

(1) evaluate the expression E

(2) find the value Vj in the list of cases that is the same as the value of the expression.

(3) execute the statement Sj associated with the value found.

Step (2) is an n-way branch, if the no. of cases is small then it is reasonable to use a sequence of conditional jump, each tests for an individual value & transfer to the code

A compact way to implement this is to create a table of pairs, each consist of value & label for statement's code. The value of the expression itself, paired with label for the default statement. If no match found, then last entry is sure to match.

If no. of values is more, it is more efficient to construct a hash table, with the labels of various statements as entries. If no entry for value possessed by the switch is found, a jump to the default statement is generated.

There is a common special case that can be implemented even more efficiently than by an n-~~or~~ way branch. If the values all lie in some small range say min to max & no. of diff. values is a reasonable fraction of max-min, then we can construct an array of max-min "buckets", where bucket j-min contains the label of the statement with value j; any bucket that would otherwise remain unfilled contains the default label.

To perform switch, evaluate the expression to obtain j & check that is in the range min to max & transfer indirect to the table entry at offset j-min.

Syntax-Directed Translation of switch-statement.

The translation of a switch statement is,

```
        goto test
L1 :    code for S1
        goto next
L2 :    code for S2
        goto next
        ...
Ln-1 :  code for Sn-1
        goto next
Ln :    code for Sn
        goto next
test :  if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
Next:
```

When we see the keyword switch, we generate a new labels test & next, and a new temporary t. Then, as we parse the expression E, we generate code to evaluate E into t. After processing E, we generate the jump goto test

When we see each case keyword, we create a new label Li & enter it into the symbol table. We place in a queue, used only to store cases, a value-label pair consisting of the value Vi of the case constant & Li. We process each statement case Vi: Si by emitting the label Li attached to the code for Si followed by the jump goto next.

When the end of the switch is found, we are ready to generate the code for the n-way branch. Reading the queue of value-label pairs, we can generate a sequence of 3-address statements of the form.

```
case t V1 L1
case t V2 L2
. . .
case t Vn-1 Ln-1
case t t Ln
next:
```

where t is the temporary holding the value of the selector expression E & Ln is the label for the default statement.

The case tViLi instruction is a synonym for

if t = Vi goto Li.

Intermediate code for Procedures.

In 3-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by the call itself.

Ex: a is an array of integer & that f is a function from integers to integers. Then the assignment

n = f(a[i]);

might translate into the following 3-address code

(i) $t_1 = i * 4$

(ii) $t_2 = a[t_1]$

(iii) param t_2

(iv) t_3 = call f, 1

(v) $n = t_3$

The first 2 lines compute the value of expression a[i] into t_2, Line (iii) makes t_2 an actual parameter for the call of f on line (iv). That line also assigns the return value to temporary t_3. Line (v) assigns the result of f(a[i]) to n.

The below productions allow function definitions & function calls.

D → define T id (F) {S}

F → ∈ | T id, F

S → return E;

E → id(A)

A → ∈ | E, A

* Non terminal D & T general declarations and type

* D consists of keyword define, a return type, the function name, formal parameters in parantheses & function body

consisting of a bracketed statement.

* F generates zero or more formal parameters
* S & E generate statements & expressions.
* S adds a statement that returns the value of an expression
* E adds function call with actual parameter generated by A
* Actual parameter is an expression.

Function types: The type of a function must encode the return type and types of the formal parameters.

Symbol tables: Let s be the top symbol table when the function definition is reached. In the production for D, after seeing define & the function name, we push s & set up a new symbol table

```
Env.push(top); top = new Env(top);
```

Type checking: Within expressions, a function is treated like any other operator. For ex, if f is a function with a parameter of type real, then the integer 2 is coerced to a real in the call f(2).

Function calls: When generating 3-address instructions for a function call id(E, E, ..., E) it is sufficient to generate the 3-address instructions for evaluating or reducing the parameters E to addresses, followed by a param instruction for each parameter.

QUESTIONS

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

A quadruples has 4 fields, op, arg1, arg2 & re the op field contains an incremental code for the operato

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

t_1 = minus c

t_2 = 6 * t_1

t_3 = 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_2	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,

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 DAG

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

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

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

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

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

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

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

~~(ix)~~ (viii) P_8 = Leaf (id, entry-b) = P_3

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

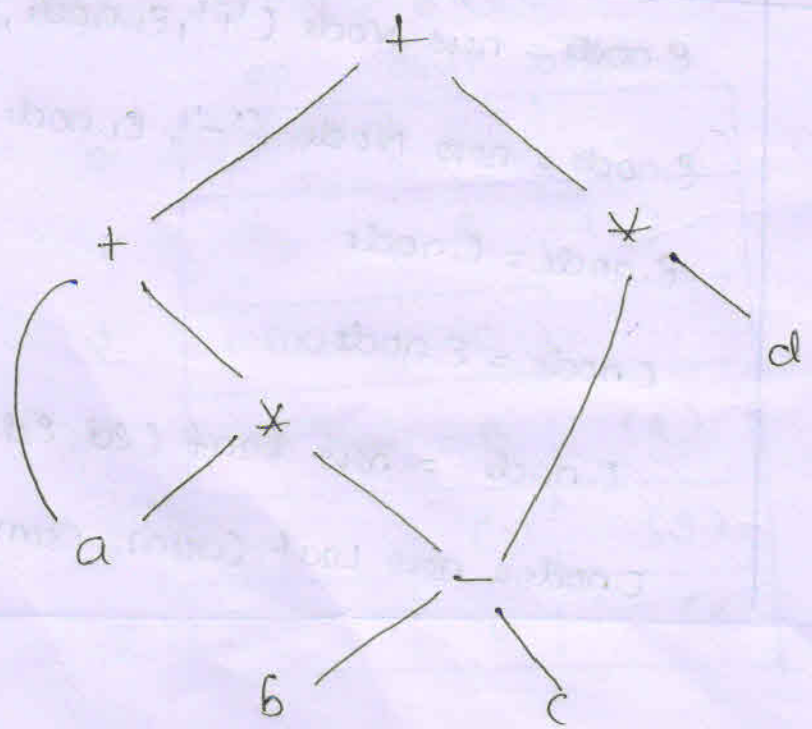
(x) P_{10} = Node ('-', P_3, P_4) = P_5

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

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

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

DAG



3. Write syntax directed deffaltion for flow of control statment

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	S.next = newlabel() P.code = S.code || label (S.next)
$S \rightarrow$ assign	s.code = assign.code
$S \rightarrow$ if (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label (B.true) || S_1.code
$S \rightarrow$ if (B) S_1 else S_2	B.true = newlabel() B.false = newlabel() S_1.next = S_2.next = S.next S.code = B.code || label (B.true) || S_1.code || gen ('goto' S.next) || label (B.false) || S_2.code
$S \rightarrow$ while (B) S_1	begin = newlabel() B.true = newlabel() B.false = S.next S_1.next = begin S.code = label (begin) || B.code || label (B.true) || S_1.code || gen ('goto' begin)
$S \rightarrow S_1 S_2$	S_1.next = newlabel()

	$S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next)$ $\mid\mid S_2.code.$

4. Explain syntax directed translation of switch statement.

Translation of a switch statement

```
        goto test
L1 :    code for S1
        goto next
L2 :    code for S2
        goto next.
        ...
Ln-1 :  code for Sn-1
        goto next
Ln :    code for Sn.
        goto next
test :  if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next :
```

Fig (1)

We can write more straightforward sequence as,

```
        if t != V1 goto L1
        code for S1
        goto next
L1 :    if t != V2 goto L2
```

```
        code for S2
        goto next
L1 :    if t != V2 goto L2
        code for S2
        goto next
L2 :
        . . .
Ln-2 :  if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1 :  code for Sn
next :
```

Above code require the compiler to do extensive analyses to find the most efficient implementation. It is inconvenient in a one pass compiler.

To translate into the form of fig (1), when we see the keyword switch, we generate 2 new labels test & next, & a new temporary t. Then, as we parse the expression E, we generate code to evaluate E into t. After processing E, we generate the jump goto test. When we see each case keyword, we create a new table LI & enter it into the symbol table. We place in a queue used only to store cases, a value-label pair consisting of the value VI of the case constant LI. We process each statement case VI: SI by emitting the label LI attached to the code for SI, followed by the jump goto next.

When the end of the switch is found, we are ready to

generate the code for the n way branch. code is

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
next :
```

where, t → temporary holding the value of the selector expression

Ln → label for the default statement.

case tVeLe is same as if t = Ve goto Le.