UNIT - 4

~~INTENE~~ **CODE GENERATION**

INTERMEDIATE CODE GENERATION

(1) Intermediate codes are machine independent codes, but they are close to machine ~~codes~~ instructions.

Syntax tree, Postfix Notation, three address codes can be used as intermediate language.

Three-address code -

It is a sequence of statements of the general form $x := y$ op z

Eg - $A = -B * (C+D)$

Three address code is as follows -

$T_1 = -B$

$T_2 = C+D$

$T_3 = T_1 * T_2$

$A = T_3$

Quadruple -

	OPERATOR	OPERAND1	OPERAND2	RESULT
(1)	—	B		T_1
(2)	+	C	D	T_2
(3)	*	T_1	T_2	T_3
(4)	=	T_3		A

Triple -

	OPERATOR	OPERAND 1	OPERAND 2
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple -

	STATEMENT
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	OPERATOR	OPERAND1	OPERAND2
(56)	—	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

Date __/__/__
Page ____

② **Declarations** -

In the declarative statements the data items along with their data types are declared.

Computing the types and relative address of declared names -

Eg -

$S \rightarrow D$	{offset:=0}
$D \rightarrow id : T$	{enter_tab(id.name, T.type, offset); offset:=offset+T.width}
$T \rightarrow$ integer	{T.type:=integer; T.width:=4}
$T \rightarrow$ real	{T.type:=real; T.width:=8}
$T \rightarrow$ array[num] of T_1	{T.type:=array(num.val, T_1.type); T.width:=num.val × T_1.width}
$T \rightarrow *T_1$	{T.type:=pointer(T.type); T.width:=4}

Declarative statement

③ **Assignment Statements** -

It mainly deals with the expressions. The expressions can be of type integer, real, array and record. ~~In this section we will~~

Syntax directed translation schemes for generating three address code -

PRODUCTION RULE	SEMANTIC ACTIONS
$S \rightarrow id := E$	{p:=lookup(id.name); If p ≠ nill then emit(p ':=' E.place) else error}
$E \rightarrow E_1 + E_2$	{E.place := newtemp; emit(E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place:=newtemp; emit(E.place ':=' E_1.place '*' E_2.place)}
$E \rightarrow -E_1$	{E.place:=newtemp; emit(E.place ':=' 'uminus' E_1.place)}
$E \rightarrow (E_1)$	{E.place:= E_1.place}
$E \rightarrow id$	{p:=lookup(id.name); if p ≠ nill then E.place:=p else error}

④ ~~Boolean Expression~~ - **Type Conversion** -

Type conversion take place when two different types are used in a single expression.

Eg - Generating three address code for an expression : $x := a + b * c + d$ - (real: a; integer: b * c + d)

t_1 := b int* c	t_3 := inttoreal t_2	x := t_4
t_2 = t_1 int+ d	t_4 = a real+ t_3	

~~L.place =~~

Array [i]

L.place   L.offset

→ ~~Addressing array elements -~~

~~Row-major order, $A[i,j] = base + [(i - low_i) \times n_2 + (j - low_j)] \times W$~~

~~Column-major order, $A[i,j] = base + [i - low$~~

→ Addressing array elements -

Column-major order, $A[i][j] = B + W\ [(i - l_r) + n_R(j - l_c)]$

Row-major order, $A[i][j] = B + W\ [n_c(i - l_r) + (j - l_c)]$

Translation scheme for addressing array elements -

(1) S → L := E   { if L.offset = null then /* L is simple id */
        emit (L.place ':=' E.place);
    else
        emit (~~L.offset~~ ' L.place ['L.offset'] ':=' E.place) }

(2) E → $E_1 + E_2$ { E.place := newtemp;
        emit (E.place ':=' E_1.place '+' E_2.place) }

(3) E → (E_1)   { E.place := E_1.place }

(4) E → L   { if L.offset=null then /* L is simple id */
        E.place := L.place
    else begin
        E.place := newtemp;
        emit (E.place ':=' L.place ['L.offset'])
    end }

(5) L → Elist]   { L.place := newtemp; L.offset := newtemp;
        emit (L.place ':=' c (Elist.array));
        emit (L.offset ':=' Elist.place '*' width (Elist.array)) }

(6) L → id   { L.place := id.place; L.offset := null }

(7) Elist → $Elist_1$, E   { t := newtemp; m := $EList_1$.ndim + 1;
        emit (t ':=' $EList_1$.place '*' limit ($Elist_1$.array, m));
        emit (t ':=' t '+' E.place); Elist.array = $Elist_1$.array;
        Elist.place := t; Elist.ndim := m }

(8) Elist → id [E   { Elist.array := id.place; Elist.place := E.place;
        Elist.ndim = 1 }

Date ___/___/___
Page ______

(4) **Boolean Expressions** –

It can be generated by the following grammar –

$E \to E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$

relop → comparison operators (<, ≤, =, ≠, > or ≥)

→ **Methods of Translating Boolean Expressions** –

(1) The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.

Translation scheme using a numerical representation for booleans –

$E \to E_1 \text{ or } E_2$ {E.place := newtemp; emit(E.place ':=' E_1.place 'or' E_2.place)}

$E \to E_1 \text{ and } E_2$ {E.place := newtemp; emit (E.place ':=' E_1.place 'and' E_2.place)}

$E \to \text{not } E_1$ {E.place := newtemp; emit (E.place ':=' 'not' E_1.place)}

$E \to (E_1)$ {E.place := E_1.place}

$E \to id_1 \text{ relop } id_2$ {E.place := newtemp; emit ('if' id_1.place relop.op id_2.place 'goto' nextstat + 3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1');

E → true {E.place := newtemp; emit (E.place ':=' '1')}

E → false {E.place := newtemp; emit (E.place ':=' '0')}

Eg – a < b ~~and c~~

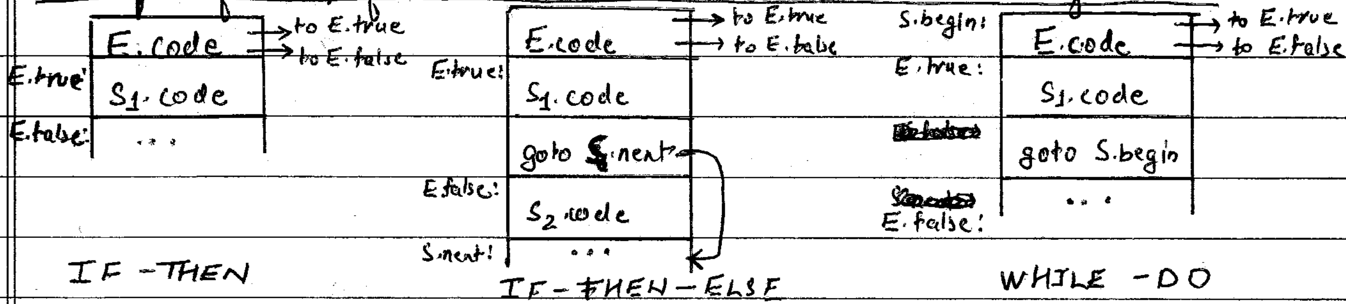
100: if a < b goto 103 (nextstat + 3)
101: t := 0
102: goto 104 (nextstat + 2)
103: t := 1
104:

→ **Short circuit** (jumping) code – We can also translate a boolean expression into three-address code without generating code for any of the boolean operations and without having the code necessarily evaluate the entire expression.

(2) The second method of implementing boolean expressions is by flow of control, that is, representing the value of a boolean expression by a position reached in a program.

$S \rightarrow$ if E then S_1 | if E then S_1 and S_2 | while E do S_1

→ Code for if-then, if-then-else and while-do statements is given as -



→ SDD for flow-of-control statements is given as -

$S \rightarrow$ if E then S_1 E.true := newlabel; E.false := S.next; S_1.next := S.next;
S.code := E.code || gen(E.true ':') || S_1.code;

$S \rightarrow$ if E then S_1 else S_2 E.true := newlabel; E.false := newlabel; S_1.next = S.next;
S_2.next = S.next; S.code := E.code || gen(E.true ':') || S_1.code ||
gen('goto' S.next) || gen(E.false ':') || S_2.code;

$S \rightarrow$ while E do S_1 S.begin := newlabel; E.true := newlabel; E.false := S.next;
S_1.next := S.begin; S.code := gen(S.begin ':') || E.code ||
gen(E.true ':') || S_1.code || gen('goto' S.begin)

→ Control-flow translation of boolean expression -

$E \rightarrow E_1$ or E_2 E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false
E.code := E_1.code || gen(E_1.false ':') || E_2.code

$E \rightarrow E_1$ and E_2 E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false
E.code := E_1.code || gen(E_1.true ':') || E_2.code

$E \rightarrow$ not E_1 E_1.true := E.false; E_1.false := E.true; E.code := E_1.code

$E \rightarrow (E_1)$ E_1.true := E.true; E_1.false := E.false; E.code := E_1.code

$E \rightarrow id_1$ relop id_2 E.code := gen('if' id_1.place relop.op id_2.place 'goto' E.true) ||
gen('goto' E.false)

$E \rightarrow$ true   E.code := gen('goto' E.true)

$E \rightarrow$ false   E.code := gen('goto' E.false)

SDD TO PRODUCE THREE-ADDRESS CODE FOR BOOLEANS

Date __/__/__

Page ____

(5) Case Statements -

→ Switch Statement syntax -

switch expression
begin
case value : statement
case value : statement
...
case value : statement
default : statement
end.

→ Three address code -

case V_1 L_1
case V_2 L_2
...
case V_{n-1} L_{n-1}
case t L_n
label next

$t \to$ name holding the value

$E \to$ Expression, $L_n \to$ default statement

→ Translation of case statement -

```
code to Evaluate E into t
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:
```

```
code to Evaluate E into t
if t ≠ V1 goto L1
code for S1
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:
```

myCOMPANION

*** We generate quadruples into a quadruple array. Label will be indices into this array.

→ *** ~~truelist and falselist of nonterminal E are used to generate jumping code~~ for boolean expressions

(6) __Backpatching__ -

It is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

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

To manipulate lists of labels, we use three functions -

(1) makelist (i) - create a new list containing only i, an index into the array of quadruples. makelist returns a pointer to the list it has made.

(2) merge (p_1, p_2) - concatenates the lists pointed to by p_1 and p_2 and returns a pointer to the concatenated list

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

→ __Boolean Expressions__ - Translation scheme is as follows -

(1) $E \rightarrow E_1$ or $M E_2$ { backpatch (E_1.falselist, M.quad);
E.truelist := merge (E_1.truelist, E_2.truelist);
E.falselist := E_2.falselist }

(2) $E \rightarrow E_1$ and $M E_2$ { backpatch (E_1.truelist, M.quad);
~~E.falselist := merge~~ E.truelist = E_2.truelist;
E.falselist := merge (E_1.falselist, E_2.falselist);

(3) $E \rightarrow$ not E_1 { ~~backpatch~~ E.truelist := E_1.falselist; E.falselist := E_1.truelist; }

(4) $E \rightarrow (E_1)$ { E.truelist := E_1.truelist; E.falselist := E_1.falselist }

(5) $E \rightarrow id_1$ relop id_2 { E.truelist := makelist (nextquad);
E.falselist := makelist (nextquad + 1);
emit ('if' id_1.place relop.op id_2.place 'goto _')
emit ('goto _') }

(6) $E \rightarrow$ true { E.truelist := makelist (nextquad);
emit ('goto _') }

(7) $E \rightarrow$ false { E.falselist := makelist (nextquad);
emit ('goto _') }

(8) $M \rightarrow \epsilon$ { M.quad := nextquad }

Date __/__/__
Page ____

Eg - A<B OR C<D AND P<Q

100 if A<B goto _

101 goto 102 → $E \to E_1$ or $M E_2$ [backpatch (E_1.falselist, M.quad)]

102 if C<D goto 104 → $E \to E_1$ and $M E_2$ [backpatch (E_1.truelist, M.quad)]

103 goto _

104 if P<Q goto _

105 goto _

→ Flow-of-Control Statements - Translation scheme is given as -

(1) $S \to$ if E then M_1 S_1 N else $M_2 S_2$

{backpatch (E.truelist, M_1.quad); backpatch (E.falselist, M_2.quad);
S.nextlist := merge (S_1.nextlist, merge (N.nextlist, S_2.nextlist)) }

(2) $S \to$ if E then M S_1 {backpatch (E.truelist, M.quad);
S.nextlist := merge (E.falselist, S_1.nextlist) }

(3) $S \to$ while M_1 E do M_2 S_1 {backpatch (S_1.nextlist, M_1.quad);
backpatch (E.truelist, M_2.quad);
S.nextlist := E.falselist
emit ('goto' M_1.quad) }

(4) $N \to \epsilon$ {N.nextlist := makelist (nextquad); emit ('goto _') }

(5) $M \to \epsilon$ {M.quad := nextquad}

(6) $S \to$ begin L end {S.nextList := L.nextlist}

(7) $S \to A$ { S.nextlist := nil }

(8) $L \to L_1$; M S {backpatch (L_1.nextlist, M.quad); L.nextlist := S.nextList}

(9) $L \to S$ {L.nextlist := S.nextlist}

⑦ Procedure Calls -

Procedure or function is an important programming construct which is used to obtain the modularity in the user program.

Consider the grammar for a simple procedure call -

$S \to$ call id (Elist)

Elist → Elist, E | E

mycompanion

Translation scheme is given as -

(1) $S \rightarrow$ call id (Elist) { for each item p on queue do emit('param' p); emit ('call' id.place) }

(2) Elist $\rightarrow$ Elist , E { append E.place to the end of queue }

(3) Elist $\rightarrow$ E { initialize queue to contain only E.place }

CODE GENERATION -

① Issues in the design of a code generation -

(1) Input to the Code Generator -

The code generation phase can therefore proceed on the assumption that its input is free of errors (i.e. type checking, type conversion has been done, semantic errors)

(2) ~~Target programs~~ -

Advantage of absolute machine-language program is that it can be placed in a fixed location in memory and immediately executed.

Advantages of relocatable machine-language program is that it allows subprograms to be compiled separately.

Advantage of assembly language program ~~as~~ is that it makes the process of code generation somewhat easier.

(3) Memory Management -

Using the symbol table information about memory requirements, code generator determines the addresses in the target code. Similarly, if the three address code contains the labels then those labels can be converted into equivalent memory addresses

(4) Instruction Selection -

The nature of the instruction set of the target machine determines the difficulty of instruction selection. Important factors are uniformity, completeness, instruction speed and machine idioms.

(5) Register allocation -

Instruction involving register operands are usually shorter and faster than those involving operands in memory. The use of registers is often subdivided into two subproblems -

(i) During register allocation, we select the set of variables that will reside in registers at a point in the program.

Date ___/___/___
Page ______

(iii) During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

Certain machine requires register-pairs for some operands and results.

(6) Choice of Evaluation Order -

The evaluation order is an important factor in generating an efficient target code. We can avoid the problem of choice by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

(7) Approaches to Code Generation -

Designing a code generator so it can be easily implemented, tested, maintained and produce correct code is an important design goal.

(2) Basic block and flow graphs -

→ Basic blocks -

It is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

A name in a basic block is said to be live at a given point if its value is used after that point in the program. and if not used after that point in the program it said to be dead.

Partition into basic blocks algorithm -

INPUT → A sequence of three-address statements

OUTPUT → List of basic blocks with each three-address statements in exactly one block

METHOD -

(i) We first determine the set of leaders, the first statement of basic blocks. The rules we use are the following -

1) The first statement is a leader

2) Any statement that is the target of a conditional or unconditional goto is a leader

3) Any statement that immediately follows a goto or conditional goto statement is a leader

(ii) For each leader, its basic block consists of the leader and all statements up to but not includes the next leader or the end of the program.

Date ___/___/___
Page ______

→ **Transformation on basic blocks** -

Two basic blocks are said to be equivalent if they compute the same set of expressions. There are two important classes of local transformations that can be applied to basic blocks are -

(1) Structure-Preserving Transformations - They are as follows -

(i) Common subexpression elimination
(ii) Dead code elimination
(iii) Renaming of temporary files
(iv) Interchange of two independent adjacent statements.

(2) Algebraic transformations

→ **Flow Graphs** -

A graph representation of three-address statements is called a flow graph. It is useful for understanding code-generation algorithms.

Nodes of the flow graph → basic ~~flow~~ blocks,

Block whose leader is the first statement → initial blocks.

There is a directed edge from block B_1 to block B_2 if B_2 immediately follows B_1 in the given sequence or there is any conditional or unconditional jump. We can ~~sy~~ say B_1 is a predecessor of B_2 or B_2 is a successor of B_1

→ **Loops** -

It is a collection of nodes in the flow graph such that -

(i) All such nodes are strongly connected that means always there is a path from any node to any other node within that loop.

(ii) The collection of nodes has unique entry that means there is only one path from a node outside the loop to the node inside of loop.

The loop that contains no other loop is called inner loop

(3) **Register allocation and assignment** -

The most commonly used strategy to register allocation and assignment is to assign specific values to specific register

Advantage - simplified design of for code generation

Date __/__/__
Page ____

Disadvantage - Design of compiler becomes complicated because of restrictive use of registers.

Various strategies used in register allocation and assignment are -

(1) Global Register allocation -

Allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation. Adopted strategies are -

(i) Store the most frequently used variables in fixed registers throughout the loop.
(ii) Assign some fixed number of global registers to hold the most active values in each inner loop.
(iii) Registers not already allocated may be used to hold values local to one block.
(iv) By using register declaration use by C or Bliss programmer.

(2) Usage Count -

It is the count for the use of some variable x in some register used in any basic block. The approximate formula for usage count for the loop L in some basic block B can be given as, $\sum_{\text{block } B \text{ in } L} [\text{use}(x, B) + 2 * \text{live}(x, B)]$

use(x, B) → Number of times x used in block B

live (x, B) → 1, if x is live on exit from B otherwise 0.

(3) Register assignment for Outer loop - L_1 → outer loop, a → variable, L_2 → inner loop

Following criteria should be adopted for register assignment for outer loop-

(i) If 'a' is allocated in loop L2 then it should not be allocated in L1-L2.
(ii) If 'a' is allocated in L1 and it is not allocated in L2 then store 'a' on a entrance to L2 and load 'a' while leaving L2.
(iii) If 'a' is allocated to L2 and not in L1 then load a on entrance of L2 and store 'a' on exits from L2.

(4) Graph coloring for Register assignment -

If all register are occupied then which register should be freed for computation is solved by graph coloring technique which works in two passes-

(1) In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
(2) In the second pass the register inference graph is prepared. In the register inference graph each node is a symbolic registers and an edge connects two nodes

Date ___/___/___
Page ________

where one is live at a point where other is defined.

Then the graph colouring technique is applied for this register inference graph using K-colour in which no two symbolic registers can interfere with each other with assigned physical registers.

(4) **DAG representation of basic blocks** - DAG → Direct Acyclic Graph

A DAG is constructed from three address statement which is used to apply the transformations on basic block.

A DAG is constructed for the following type of labels on nodes -

(1) Leaf nodes are labeled by identifiers or variable names or constants

(2) Interior nodes store operator values.

→ **Algorithm for construction of DAG** -

We assume the three address statements ~~and~~ could of following types -

case (i) $x := y \; op \; z$ case (ii) $x := op \; y$ case (iii) $x := y$

STEP 1 - If y is undefined then create node(y). Similarly if z is undefined create a node (z)

STEP 2 - For the case (i) create a node(op) whose left child is node(y) and node (z) will be the right child. Also check for any common subexpressions. For the case (ii) determine whether is a node labeled op, such node will have a child node (y). In case (iii) node n will be node(y)

STEP 3 - Delete x from list of identifiers from node(x). Append x to the list of attached identifiers for node n found in 2.

→ **Application of DAG** -

(1) Determines the common sub-expressions

(2) Determines which statements of the block could have their computed value outside the block.

(3) Determines which names are used inside the block and computed outside the block.

(4) Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form $x := y$ unless and until it is must.

Date ___/___/___
Page ______

Array pointers and procedure calls -

The rules to be enforced are the following -

(1) Any evaluation of or assignment to an ~~array~~ element of array 'a' must follow the previous assignment to an element of that array if there is one.

(2) Any assignment to an element of array a must follow any previous evaluation of a.

(3) Any use of any identifier must follow the previous procedure call or indirect assignment through a pointer if there is one

(4) Any procedure call or indirect assignment through a pointer must follow all previous evaluation of any identifier.

(5) Peephole Optimization -

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

→ Characteristics of peephole optimization are -

(1) Redundant instruction elimination -

Redundant loads and stores and unreachable code is eliminated

(2) Flow of Control Optimization -

Unnecessary jumps on jumps can be eliminated

(3) Algebric simplification

(4) Reduction in strength -

Certain machine instructions are cheaper than the others. We can replace instructions by equivalent cheaper instruction. Eg - x^2 is cheaper that $x * x$

(5) Use of machine idioms -

We can replace target instructions by equivalent machine instructions in order to improve the efficiency. Eg - Some machines have auto-increment or auto decrement addressing modes that are used to perform add or subtract operations

(6) **Generating code from DAG** -

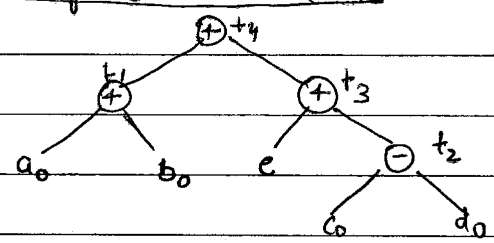
It is much simpler than the linear sequence of three address code which increase the efficiency of the code. Various algorithms are -

(1) **Rearranging order** -

By changing the order in which computations are done we can obtain object code with minimum cost. Eg - $(a+b)+(e+(c-d))$

Code generated by three address code -			By changing the order -		
MOV	a, R0	$t_1 := a+b$	$t_2 := c-d$	MOV	c, R0
ADD	b, R0	$t_2 := c-d$	$t_3 := e+t_2$	SUB	d, R0
MOV	c, R1	$t_3 := e+t_2$	$t_1 := a+b$	MOV	e, R1
SUB	d, R1	$t_4 := t_1+t_3$	$t_4 := t_1+t_3$	ADD	R0, R1
MOV	R0, t_1			MOV	a, R0
MOV	e, R0			ADD	b, R0
ADD	R0, R1			ADD	R1, R0
MOV	t_1, R0			MOV	R0, t_4
ADD	R1, R0				
MOV	R0, t_4				

DAG for $(a+b)+(e+(c-d))$ -



(2) **Heuristic Ordering** -

Node listing algorithm is given as-

```
while unlisted interior node remain do begin
    select an unlisted node n, all of whose parents have been listed;
    list n;
    while the leftmost child m of n has no unlisted parents and is not a leaf do
    /* since n was just listed, m is not yet listed */
    begin
        list m;
        n := m
    end
end
```

(3) **Labelling algorithm** - Generates optimal code in which minimum registers are required.

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$ [bottom up order]

```
procedure gencode(n);
begin
/* case 0 */
if n is a left leaf representing operand name and n is
        the leftmost child of its parent then
    print 'MOV' || name || ',' || top(rstack)
else if n is an interior node with operator op, left child n1,
        and right child n2 then
/* case 1 */
    if label(n2) = 0 then begin
        let name be the operand represented by n2;
        gencode(n1);
        print op || name || ',' || top(rstack)
    end
/* case 2 */
    else if 1 ≤ label(n1) < label(n2) and label(n1) < r then begin
        swap(rstack);
        gencode(n2);
        R := pop(rstack);  /* n2 was evaluated into register R */
        gencode(n1);
        print op || R || ',' || top(rstack);
        push(rstack, R);
        swap(rstack)
    end
/* case 3 */
    else if 1 ≤ label(n2) ≤ label(n1) and label(n2) < r then begin
        gencode(n1);
        R := pop(rstack);  /* n1 was evaluated into register R */
        gencode(n2);
        print op || top(rstack) || ',' || R;
        push(rstack, R)
    end
/* case 4, both labels ≥ r, the total number of registers */
    else begin
        gencode(n2);
        T := pop(tstack);
        print 'MOV' || top(rstack) || ',' || T;
        gencode(n1);
        push(tstack, T);
        print op || T || ',' || top(rstack)
    end
end
```

Let us generate a code from this labelling algorithm.

Consider the labelled tree as follows.

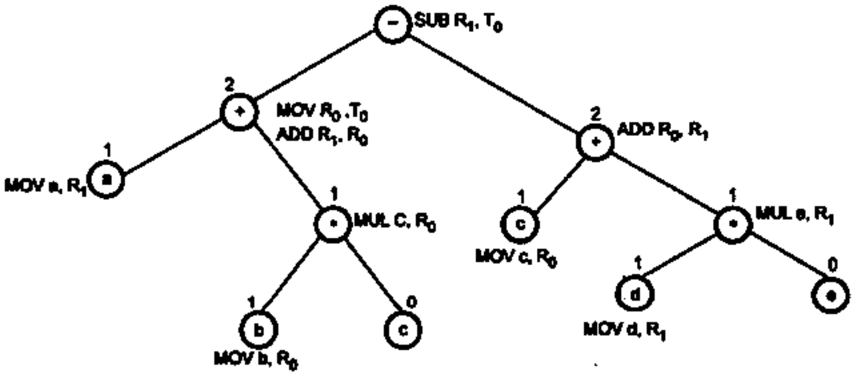


Fig. 7.19 Labelled tree with generated code for $((a+(b*c))-(c+(d*e)))$

```
MOV  b,  R0
MUL  c,  R0
MOV  a,  R1
ADD  R1,  R0
MOV  R0,  T0
MOV  d,  R1
MUL  e,  R1
MOV  c,  R0
ADD  R0,  R1
SUB  R1,  T0
```

Label computation

```
(1)  if n is a leaf then
(2)      if n is the leftmost child of its parent then
(3)          label(n) := 1
(4)      else label(n) := 0
     else begin /* n is an interior node */
(5)      let n1, n2, . . . , nk be the children of n ordered by label,
             so label(n1) ≥ label(n2) ≥ · · · ≥ label(nk);
(6)      label(n) := max (label(ni)+i-1)
                    1≤i≤k
     end
```