
KTUNOTES

MODULE-V

RUN TIME ENVIRONMENT

At the time of execution, the allocation and deallocation of data objects is managed by run-time support package. The design of run-time support package is influenced by semantics of the procedures. Each execution of a procedure is referred to as an activation of the procedure. If the procedure is recursive several of its activations may be alive at the same time.

SOURCE LANGUAGE ISSUES

Consider a program consists of procedures

Procedures

A procedure definition is a declaration that associates an identifier with a statement. The identifier is the procedure name and statement is procedure body.

The identifiers appears in procedure definition are called formal parameters of procedure. The actual parameters are passed to a called procedure

```
sum (int x, int y)   → formal
{

}
                          main
                          {
                          a=10;
                          b=20;    → actual
                          c= sum(a,b);
```

They are substituted for formal parameters in the procedure body.

14-3-18 ACTIVATION TREE :

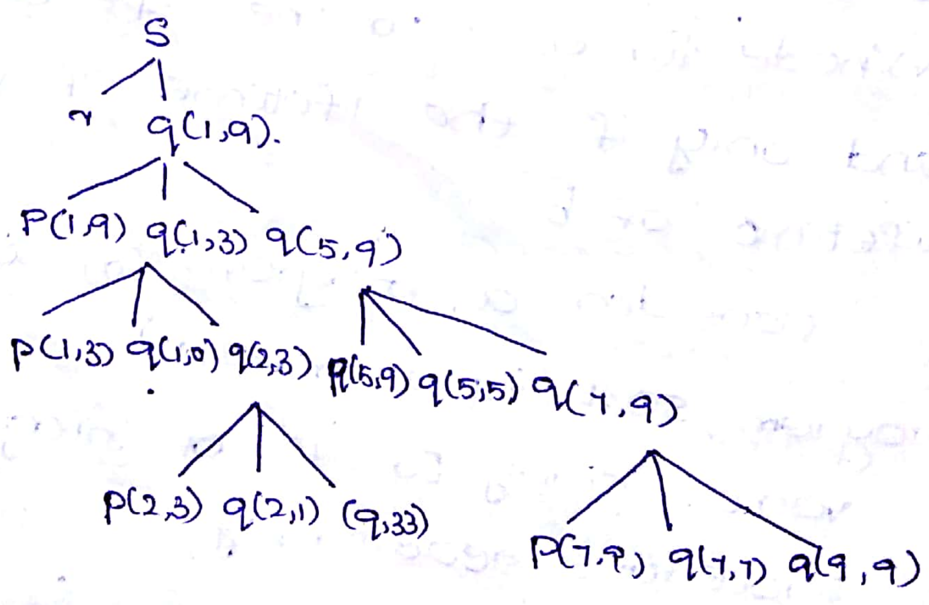
Each execution of a procedure body is referred to as activation of the procedure. The lifetime of an activation of a procedure p is a sequence of steps b/w first and last steps in execution of procedure body. We can use a tree called an activation tree, to depict the way the control enters and leaves activation. In an activation tree:

(i) Each node represents an activation of procedure
(ii) The root represents activation of main program
(iii) The node for a give the parent of node b if and only if control flows from activation a to b and
(iv) node for a is to the left of node for b if and only if the lifetime of a occurs before lifetime of b.

Consider a program for sorting integers

```
Program sort (input, output);
  var a: array [0..10] of integers;
  Procedure readarray;
  var i : integer;
  begin
```

```
for i:= 1 to 9 do read(a[i])
end;
function partition(y,z: integer): integer;
  var i,j,x,v: integer;
  begin.
  end;
Procedure quicksort (m,n: integer);
  var i: integer;
  begin
    (if n>m) then begin
      i := partition (m,n);
      quicksort(i+1,n)
    end
  end;
begin
  a[0] := -9999; a[10] := 9999;
  readarray;
  quicksort(1,9)
end
```



16.3.18

SCOPE OF A DECLARATION.

A Declaration in a language is a syntactic construct that associates information with a name. The portion of a program to which a declaration applies is called scope of the declaration.

BINDING OF THE NAMES

Even if each name is declared once in a program the same name may denote diff. data objects at run time. In a programming lang. semantics, the environment refers to a function that maps a name to a storage location and term state refers to a function that maps a storage location to the value held there.

$x = x + y;$

An assignment changes state but not the environment. When an environment associates storage location s with a name x, we say x is bound to s i.e., association is itself referred to as a binding of x.

STORAGE ORGANISATION

Sub Division of RunTime Memory

The run time storage might be subdivided to hold:

(i) generated ~~data~~ target code.

(ii) data object

(iii) a counter part of control stack to keep track of procedure activation.

The size of target code fixed at compile time, so compiler can place it in a statically ~~fixed~~ determined area. The data objects are also stored in static area. When a function call occur, execution of an activation is interrupted and information about status of machine such as value of PC and machine registers is stored on stack. A seperate area of runtime memory, called a heap, holds all other information i.e., information about activations.

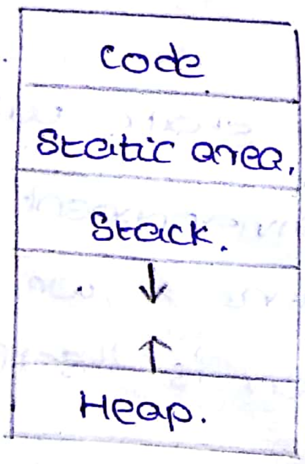
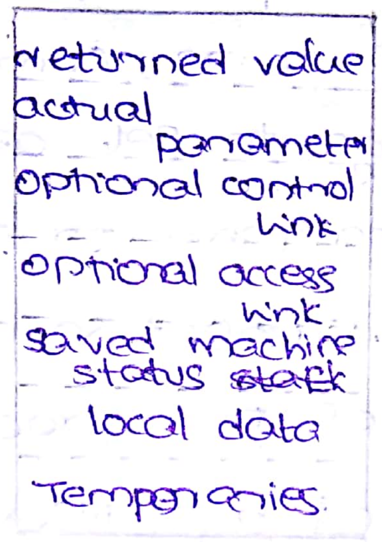


Fig. Subdivision of runtime memory.

ACTIVATION RECORDS

The information needed by single execution of a procedure is managed by a contiguous block of storage called a activation record or frame



19.3.18

STORAGE ALLOCATION STRATEGIES

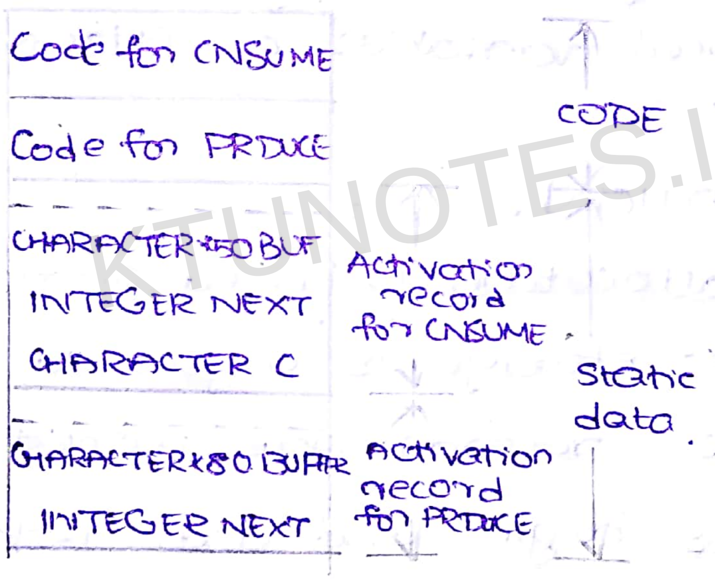


fig 1

The different allocation strategies are:

(i) Static allocation.

(ii) Stack "

(iii) Heap "

STATIC ALLOCATION

In static allocation, names are bound to storage as the program is compiled, so there is no need for run-time support package.

From type of name, compiler determines the amount of storage to set aside for that name.

Consider a FORTRAN program consist of a main program, subroutine and functions. Main program consists of two procedures, CONSUME and PRODUCE denoted by CNSUME and PRDUCE. The static storage for local identifiers of the program is given in fig 1.

The local variables of CNSUME are BUF, NEXT and C

<u>STACK ALLOCATION</u>.

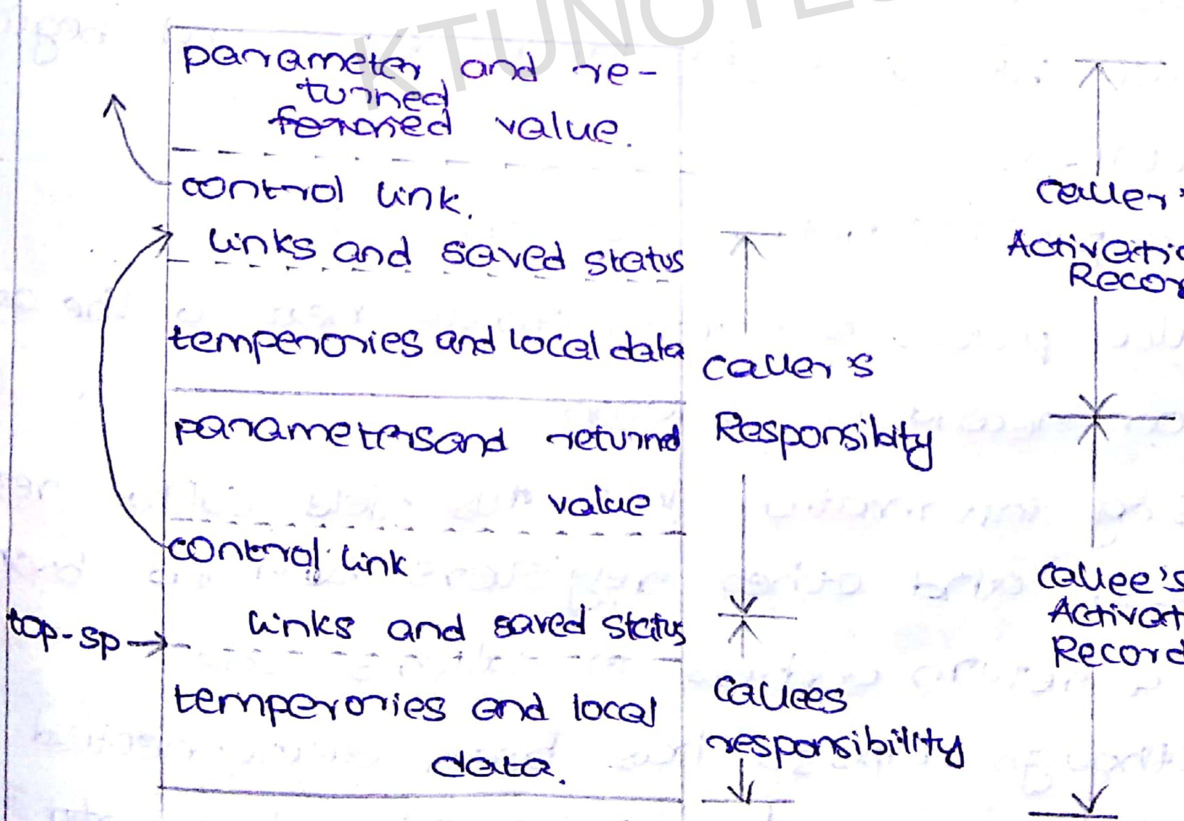
Stack allocation is based on the idea of control stack; storage is organized as a stack and activation records are pushed and popped as activations begin and end, resp.

Suppose that register top marks the top of the stack. At run-time an activation record can be allocated and deallocated by incrementing and decrementing top, resp.

The below given fig. shows the activation of the program sort and procedures readarray and quicksort.

Position in activation tree	Activation record on the stack	Remarks
s	s a: array	Frame for s
s / r	s a: array r i: integer	r activated
s / | r q(1,9)	s a: array q(1,9) i: integer	Frame for r has been popped and q(1,9) pushed.

CALLING SEQUENCES



Procedure calls are implemented by generating calling sequences in target code. A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine, so the calling procedure can continue execution.

The call sequence is:

(i) the caller evaluates actuals

(ii) caller stores a return address and old value of top-sp into callee's activation record. The caller then increments top-sp.

(iii) The callee saves registers values and other status information

(iv) callee initializes its local data and begins execution.

The possible return sequence is

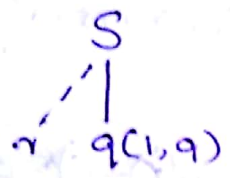
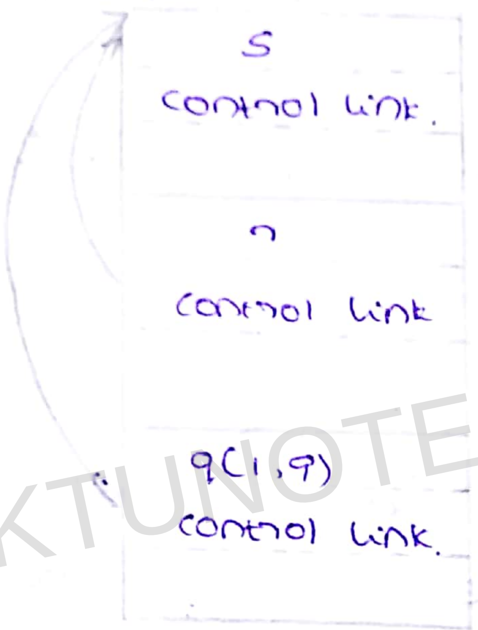
(i) callee places a return value next to the activation record of column.

(ii) Using information in status field callee restores top-sp and other registers and branches to a return address in caller's code.

(iii) Although top-sp has been decremented the caller can copy the returned value into its own activation record and use it to evaluate an expression.

HEAP ALLOCATION

Heap allocation parcels pieces of contiguous storage as needed for activation records or other objects.

Position in activation tree	Activation records in the heap	Remark.
		retained activation record for r.

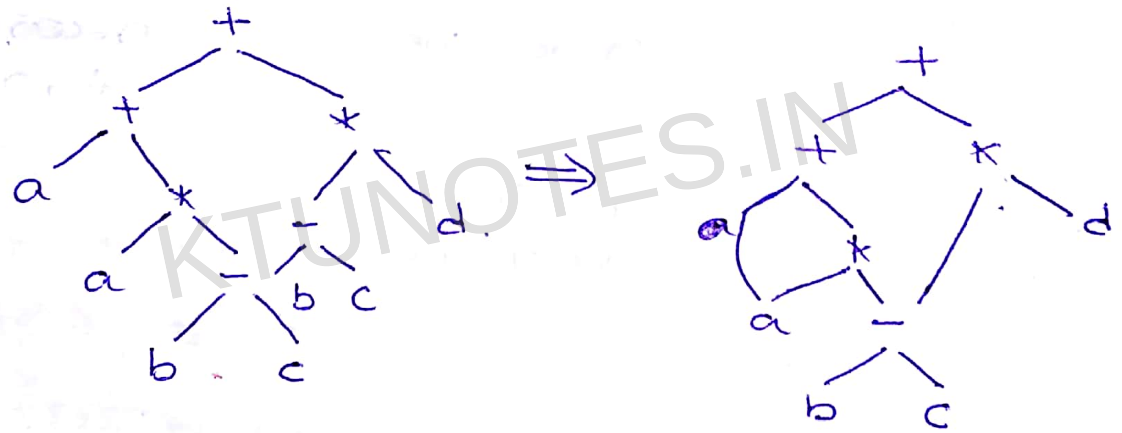
20318

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

DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Nodes in a syntax tree represents constructs in source program; children of a node represent the meaningful components of a construct. A DAG for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression. In DAG leaves corresponds to atomic ~~correspo~~ operands, interior node corresponds to operators. In DAG, some of the nodes will have more than one parent (that particular node will represent common subexpression).

Eg: consider expression a+a*(b-c)+(b-c)*d



The SDD for constructing the above given DAG is given below

$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new Leaf(id, id.entry)
$T \rightarrow num$	T.node = new Leaf(num, num.value)

The sequence of steps to construct DAG for expression a+a*(b-c)+(b-c)*d.

step 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})

21.3.18 13 : P_{13} = node ('+', P_7, P_{12})

THREE ADDRESS CODE

In three address code, there is atmost one operator on the right side of an instruction i.e., no build up arithmetic expressions are permitted.

Eg:

The source language expression x+y*z can be translated into the sequence of three address instructions as follows:

$t_1 = y*z$.

$t_2 = x+t_1$

where t_1 and t_2 are compiler generated temperory names.

Write the three address code for expression $a+a*(b-c)+(b-c)*d$.

$t_1 = b-c$

$t_2 = t_1*d$.

~~$t_3 = b-a$~~.

$t_3 = a*t_1$

$t_4 = t_3+t_2$

$t_5 = a+t_4$

Addresses and instructions

Three address code is build from two concepts addresses and instructions. Address can be one of the following:

(i) Name.

(ii) Constant.

(iii) compiler generated temperory

Different forms of three address instructions are:

1) Assignment instructions of the form $x = y \text{ op } z$, where op is a binary arithmetic or logical operation, and x, y and z are expression

2) Assignment of form $x = \text{op } y$ where op is unary operator.

3) Copy instruction of form $x = y$ where x is assigned the value of y.

4) An unconditional jump goto L where L is the label to the next instruction to be executed

5) Conditional jumps of the form if x goto L and if False x goto L

6) Conditional jumps such as if x relop y goto L

7) Procedure calls are and returns are implemented using the following instructions

(i) param x for parameters

(ii) call p, x and y = call p, x

for the procedure and function calls resp. and

(iii) return y where y represents return value which is optional.

~~The call statem~~

Eg: ~~param~~ x_1 The call statement for procedure $P(x_1, x_2, x_3, \ldots x_n)$ is given below

param x_1

param x_2

:

param x_n

call p,n.

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

8) Indexed copy instructions of the form x=y[i] and x[i]=y

9) Address and pointer assignments of form x=&y x=*y and *x=y.

consider statement

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

⇓

L: t_1 = i + 1

i = t_1

t_2 = i*8.

t_3 = a[t_2]

if t_3 < v goto L.

23.3.18

QUADRUPLES

A quadruple has 4 fields, op, arg1, arg2, result. In the three address instruction x=y+z is represented by placing + in the op field, z in arg2 field, y in arg1 field and x in result field.

Write 3 address code and equivalent quadruple for statement a= b* -c + b* -c

t_1 = minus c

$t_2 = b * t_1$

t_3 = minus c

$t_4 = b * t_3$

~~$t_5 = b * t_3$~~

$t_5 = t_2 + t_4$

$a = t_5$

	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	+	b	t_4	t_5
5	=	a t_5		a

TRIPLE

has only 3 field which we call op, arg1 and arg2.

a=b*-c+b*-c

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

INDIRECT TRIPLES

Indirect triples consist of listing of pointers of triples rather than triples themselves. For eg. let us use an array instruction to list pointers to triples in desired order.

Advantage:

With direct triples, an optimizer can move an instruction by reordering instruction list without affecting the triples themselves.

The triple representation of $a = b^* - c + b * - c$ is given below

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

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

Q6.3. ASSIGNMENT STATEMENTS

As a part of translation of assignments into 3 address code, we show how names can of arrays and records can be accessed.

NAMES IN SYMBOL TABLE

Lexeme for name represented by id is given by attribute id.name. The operation lookup (id.name) checks if there is an entry for this occurance of name in symbol table.

If so a pointer to entry is returned otherwise lookup returns nil to indicate no entry was found.

The procedure emit is used to emit 3 address statements to output file.

Consider translation scheme to produce 3 address code for assignments:

S → id = E { p = lookup (id.name);
if p ≠ nil then
emit (p '=' E.place)
else error }

E → E1 + E2 { E.place = new.temp
emit (E.place '=' E1.place '+' E2.place) }

E → E1 * E2 { E.place = new.temp;
emit (E.place '=' E1.place '*' E2.place) }

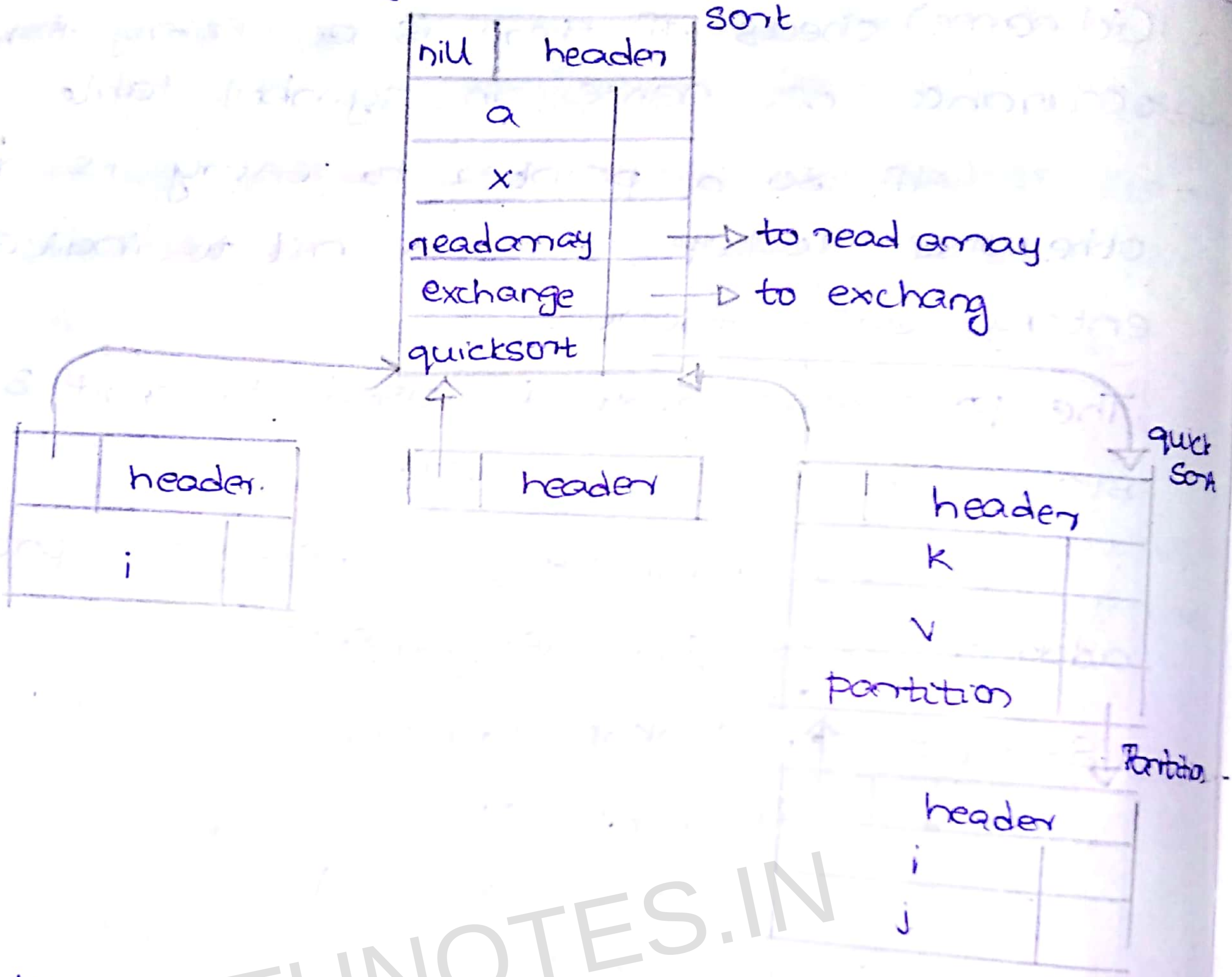
E → -E1 { E.place = new.temp;
emit (E.place '=' 'uminus' E1.place) }

E → (E) { E.place = E1.place }

E → id { p = lookup (id.name);
if p ≠ nil then
E.place = p
else error }

27.3.18

Symbol table entry for nested procedures



Boolean expressions

: are composed of boolean operations (AND OR, ~~AND~~ NOT) applied to that are boolean variables or relational expressions. Consider grammar

E → E or E | E and E | not E | (E) | id | relop id | true | false

Methods for translating boolean expressions:

There are 2 methods for expressing value of boolean expressions

(i) to evaluate true and false numerically and

(ii) to evaluate a boolean expression to analogously to an arithmetic exp.

(ii) implementing boolean exp is by flow of control.

Numerical representation

In boolean exp., 1 denote true, 0 denote false. The expression a or b and not c

The correspondance 3 address represention

$t_1 := not\ c$

$t_2 = b$ and t_1

$t_3 = a$ or t_2

Consider translation scheme given below

E → E_1 or E_2 { E.place := new.temp;
emit(E.place ':=' E_1.place 'or' E_2.place) }

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

E → not E_1 { E.place := newtemp;
emit(E.place ':=' 'not' E_1.place) }

SHORT CIRCUIT CODE

In short circuit (or jumping) code, the boolean operators and, or, not translate into jumps. Consider the statement

if (x<100 or x>200 and x!=y) x=0;

can be translated into 3 address code

if $x<100$ goto L_2

if false $x>200$ goto L_1

if false $x!=y$ goto L_1

L_2: $x=0$.

L_1:

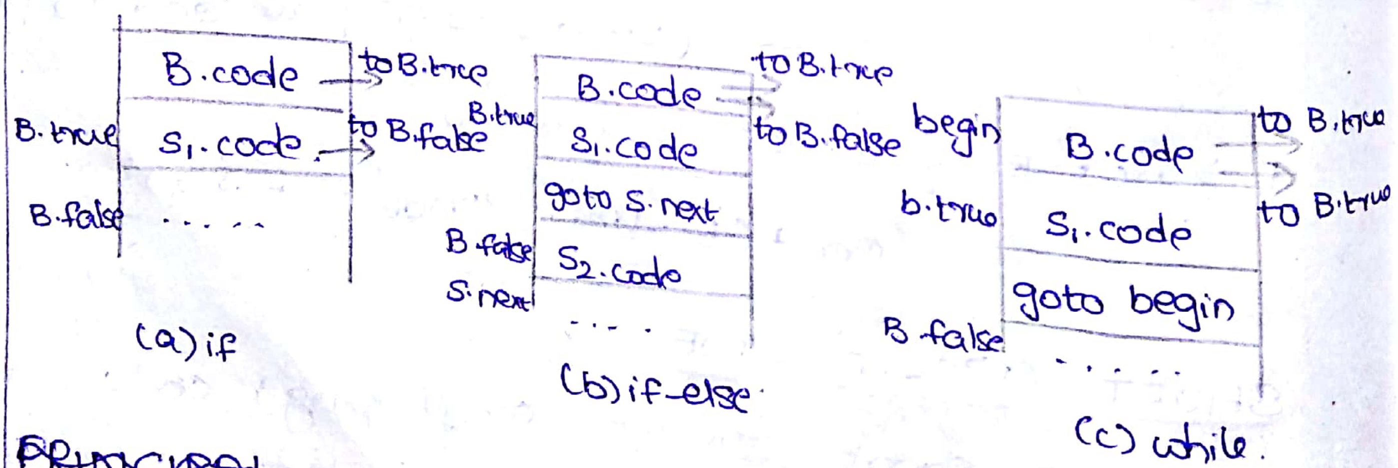
Flow of control.

Consider translation of boolean expression into 3 address code in the context of statements such as those generated by the following grammar

$S \rightarrow$ if (B) S_1

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

$S \rightarrow$ while (B) S_1



PRINCIPAL