CVR COLLEGE OF ENGINEERING

In Pursuit of Excellence

(An Autonomous Institution, NAAC 'A' Grade)

Subject Name : AUTOMATA AND COMPILER DESIGN

Subject Code : 67302

Topic : Unit IV - Intermediate Code Generation and Runtime Storage

Class : III YEAR - I Sem

Date : 12-11-2022

Faculty Name : Dr. R RAJA

Designation : Associate Professor

Department : CSIT

Unit IV - Intermediate Code Generation and Runtime Storage

Intermediate code- abstract syntax tree, translation of simple statements and control flow statements, storage organizations, storage allocation strategies, access to non-local names, parameter passing techniques, language facilities for dynamic storage allocation, symbol table and implementation.

Intermediate Code Generation

- Translating source program into an "intermediate language."
  - Simple
  - CPU Independent,
  - …yet, close in spirit to machine language.
- Or, depending on the application other intermediate languages may be used, but in general, we opt for simple, well structured intermediate forms.
- (and this completes the "Front-End" of Compilation).

Benefits

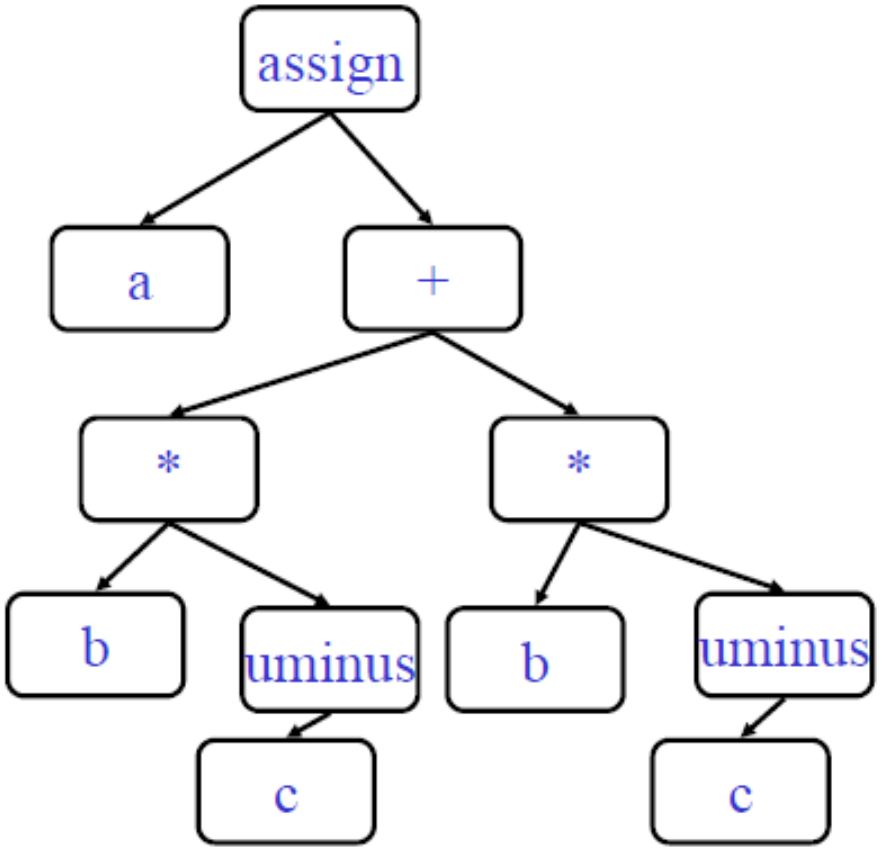
1. **Retargeting is facilitated**
2. **Machine independent Code Optimization can be applied.**

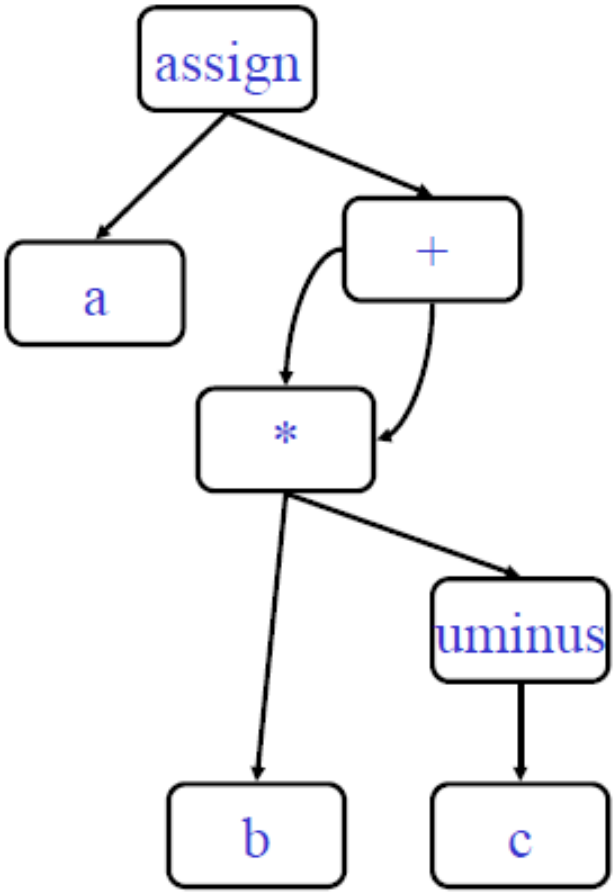
Intermediate Code Generation (II)

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Types of Intermediate Languages

- Graphical Representations.
  - Consider the assignment a:=b*-c+b*-c:





Three Address Code

- Statements of general form x:=y op z
- No built-up arithmetic expressions are allowed.
- As a result, x:=y + z * w
  should be represented as
  t_1:=z * w
  t_2:=y + t_1
  x:=t_2
- Observe that given the syntax-tree or the dag of the graphical representation we can easily derive a three address code for assignments as above.
- In fact three-address code is a linearization of the tree.
- Three-address code is useful: related to machine-language/ simple/ optimizable.

Example of 3-address code

$t_1 := - c$
$t_2 := b * t_1$
$t_3 := - c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

$t_1 := - c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

Types of Three-Address Statements.

Assignment Statement:	x:=y op z
Assignment Statement:	x:=op z
Copy Statement:	x:=z
Unconditional Jump:	goto L
Conditional Jump:	if x relop y goto L
Stack Operations:	Push/pop

More Advanced:

Procedure:

param x_1
param x_2
...
param x_n
call p,n

Index Assignments:

x:=y[i]
x[i]:=y

Address and Pointer Assignments:

x:=&y
x:=*y
*x:=y

Implementations of 3-address statements

- Quadruples

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

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

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements, II

- Triples

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

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

Temporary names are not entered into the symbol table.

Other types of 3-address statements

- e.g. ternary operations like
  x[i]:=y x:=y[i]
- require two or more entries. e.g.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

	op	*arg1*	*arg2*
(0)	[ ] =	y	i
(1)	assign	x	(0)

Declarations:

- When we encounter declarations, we need to lay out storage for the declared variables.
- For every local name in a procedure, we create a ST(Symbol Table) entry containing:
  - The **type** of the name
  - How much **storage** the name requires
  - A **relative offset** from the beginning of the static data area or beginning of the activation record
- To keep track of the current offset into the static data area or the AR, the compiler maintains a global variable, OFFSET.
- OFFSET is initialized to 0 when we begin compiling.
- After each declaration, OFFSET is incremented by the size of the declared variable.

Translation scheme for Declarations:

$P \rightarrow D$ { offset := 0 }

$D \rightarrow D ; D$

$D \rightarrow id : T$ { enter(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 }

$T \rightarrow array [num] of T1$
{ T.type := array(num.val, T1.type);
T.width := num.val * T1.width }

$T \rightarrow \uparrow T1$ { T.type := pointer(T1.type);
T.width := 4 }

Declarations Contd…

- Before the first declarations is considered, offset is set to 0
- As each new name is seen,
  - name is entered in the symbol table with offset (current value of offset).
  - Offset is incremented by the width.
- Procedure enter (name, type, offset) creates a symbol table entry.
- Non terminal T → synthesized attributes used to indicate type and width.
- Non terminals generating epsilon called non terminals, it can be rewrite productions.

P →M D
M-> epsilon {offset:=0}

ASSIGNMENT STATEMENTS

- The assignment statements mainly deals with the expressions.
- The Expressions can be of type integer, real, array and record.
- In this translation of assignments into three-address code, we show
  - how names can be looked up in the symbol table
  - how the elements of arrays and records can be accessed.

Translation Scheme to produce three-address code for assignments

1. Syntax Directed Translation Scheme :
   - Translation scheme to produce three 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 := newtemp(); emit(E.place ':=' E1.place '+' E2.place) }**
E → E1 * E2	**{ E.place := newtemp(); emit(E.place ':=' E1.place '*' E2.place) }**
E → - E1	**{ E.place := newtemp(); emit(E.place ':=' 'uminus' E1.place) }**
E → (E1)	**{ E.place := E1.place }**
E → id	**{ p := lookup(id.name); if p != nil then E.place := p else error }**

ASSIGNMENT STATEMENTS

- lookup returns the entry for id.name in the symbol table if it exists there. otherwise it returns nil to indicate that no entry was found.
- procedure emit to emit three-address statements to an output file, rather than building up code attributes for non terminals.
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E.

Generate 3-address code for assignment statement

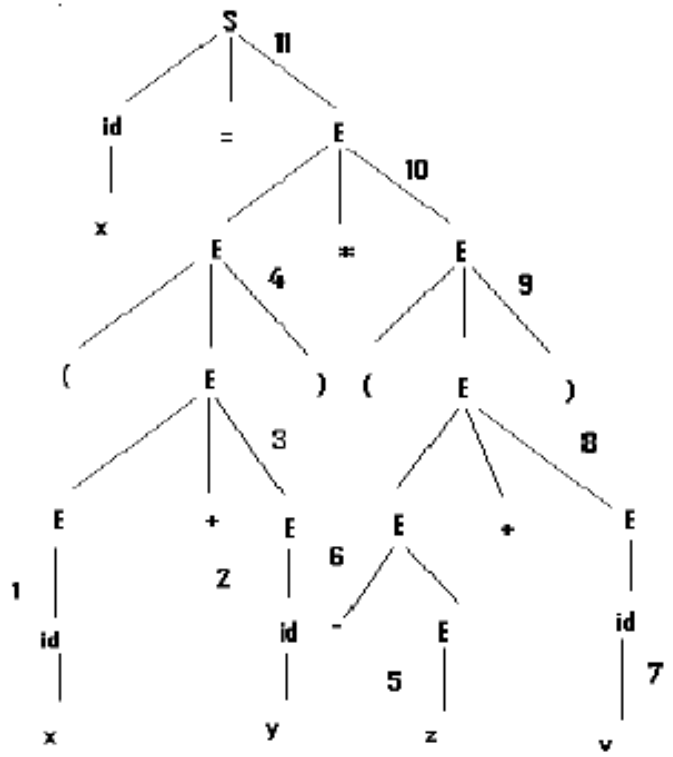
- **x:=(a+b)*(c+d)**

Production Rule	Semantic Action for Attribute Evaluation	Output
E → id	E.place := a	
E → id	E.place := b	
E → E1 + E2	E.place := t1	t1 := a + b
E → id	E.place := c	
E → id	E.place := d	
E → E1 + E2	E.place := t2	t2 := c + d
E → E1 * E2	E.place := t3	t3 := t1 * t2
S → id := E		x := t3

Generate 3-address code for assignment statement

- **x := (x + y) * (-z + v)**

1. Parse tree will be as follows:



- Three-address code will be as follows:

  t1 := x + y
  t2 := -z
  t3 := t2+ v
  t4 := t1 * t3
  x := t4

Example:

- **Generate the 3-Address code for the Assignment Statement:**
  x := a + b*c + d
  Assuming b ,c and d are integers, x and a are real variables

Solution:

```
t1 := b int * c
t2 := t1 int + d
t3 :=  inttoreal  t2
t4 := a real + t3
x := t4
```

Addressing Array Elements

- **1. One Dimensional array**

X[i] = A or A = X[i] where X is the array name and i is the offset from X.

Address of X[i] in terms of base address, low (ie. Lower bound) and high (ie.upper bound) :

= base + (i – low) * w ---- (i)

(w= size of each address block in bytes)

Expression (i) can be partially evaluated at compile time if it is rewritten as :

= i * w + (base – low * w)

Example :

- **Generate the 3-Address Code for: a := b[i] * 2 .**
- **Assume low = 1 and Take w to be 4.**
- Solution:
  - We Know that b[i] = i * w + (base – low * w)
    = i * 4 + (base – 1 * 4)
    = 4 * i + (base – 4)
  - Three-address Code:

    1. t1 := 4 * i
    2. t2 := addrB - 4
    3. t3 := t2[t1]
    4. t4 := 2 * t3
    5. a := t4

Two Dimensional Array :A[i1 , i2]

- Relative Address of A[i1,i2] calculated as :

$$= \text{base} + ((i1 - \text{low1}) * n2 + (i2 - \text{low2})) * w \quad \text{-- (ii)}$$

where

low1, low2 → lower bounds on values of i1 and i2.

n2 → number of values that i2 can take (ie. n2 = high2 – low2 +1)

- high2 → upper bound on the value of i2.

- Assuming that i1 and i2 are not known at compile time, we can rewrite the expression (ii) as :

$$= ((i1 * n2) + i2) * w + (\text{base} - ((\text{low1} * n2) + \text{low2}) * w)$$

Example :

- Let A be a 10x 20 array with low1 = low2 =1.

- Take w to be 4. Generate the 3-address statement :
  - Solution:

i. Given: low1 = low2 =1 and high1 = 10; high2 =20
n2 = high2 – low2 +1 = 20-1+1 = 20

ii. X = A[i1,i2]
= base + ((i1 – low1) * n2 + (i2 – low2))* w
= base + ((i1 – 1) * 20 + (i2 – 1)) * 4
= base – (20 + 1) * 4 + (i1 * 20 + i2) * 4
= base – 84 + (i1 * 20 + i2) * 4

iii. Three-address Code:
1. t1 := 20 * t1
2. t2 := t1 + i2
3. t3 := base - 84

BOOLEAN EXPRESSIONS

- **Three Address Code Generations for Boolean Expressions**
- **Topics:**
- **Boolean expressions**
- **Methods of translating Boolean expressions**
- **Semantic Actions for producing 3-Address Code for Boolean Expressions**
- **Translation Scheme for Boolean Expressions**
- **Flow Control Representation (Short-Circuit Code)**
- **SDD for Boolean Expressions**

1. Boolean expressions:

- Composed of the Boolean operators (and, or, and not) applied to Boolean variables or relational expressions.

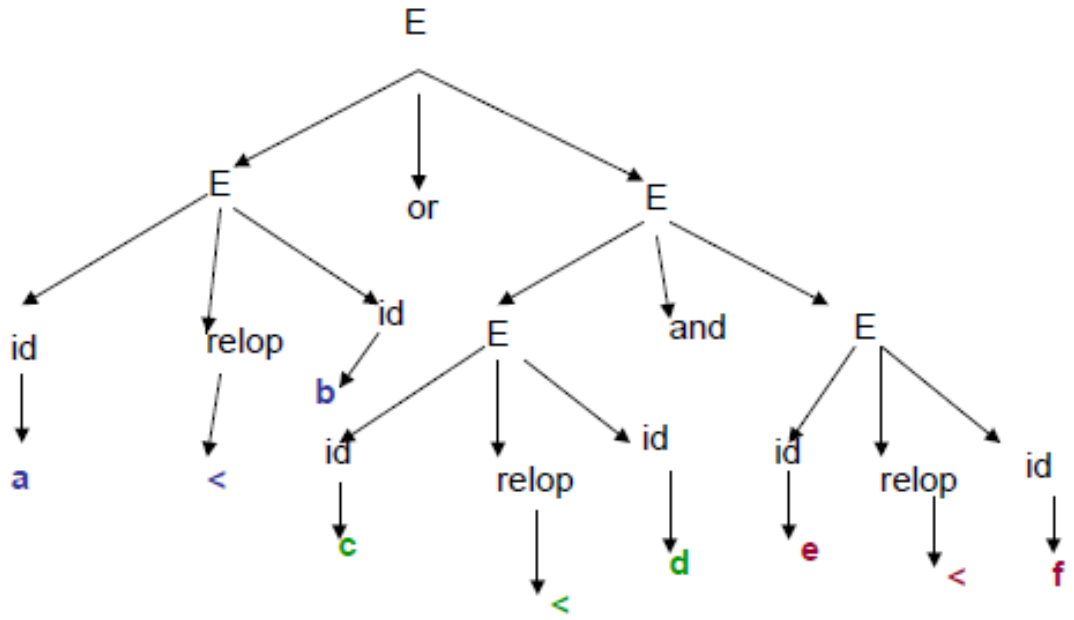
- Primary purpose:
  1. Compute logical values.
  2. used as conditional expressions in control-flow-statements like if-then, If-then-else, while-do statements.

- Relational expressions : **A relop B**
  where A & B are arithmetic expressions.
  relop ⇒ < , > , <= , >= , == , !=

- **E → E or E | E and E | not E | id1 relop id2**
  | true | false | (E)
  relop → < | > | <= | >= | == | !=

Example : a < b or c < d and e <f

- E → E or E | E and E | not E | id1 relop id2 | true | false | (E)



2. Methods of translating Boolean expressions:

- Numerical Evaluation: Encode true and false numerically and to evaluate a Boolean expression analogous to an arithmetic expression.

  ie. 1 denotes true and 0 denotes false

- Flow of Control Evaluation : Implementing Boolean expression is by flow of control.

  ie. Given the expression E1 or E2 if we determine that E1 is true, then we can conclude that the entire expression is true without having to evaluate E2

Numerical Representation:

- **Example 1: a or b and not c**

  3-Address Code Sequence:

  t1 = not c
  t2 = b and t1
  t3 = a or t2

- **Example 2: A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0.**

  3-Address Code Sequence:

```
if a < b goto L1;

        goto L2;
L1:  t  = 1;
        goto L3;

L2:  t  =0;

L3:
```

3. Semantic Actions for producing 3-Address Code for Boolean Expressions:

E → E1 or E2	{ E.place := newtemp(); emit (E.place ':=' E1.place 'or' E2.place);}
E → E1 and E2	{ E.place := newtemp(); emit (E.place ':=' E1.place 'and' E2.place);}
E → not E1	{ E.place := newtemp(); emit (E.place ':=' 'not' E1.place); }
E → (E1)	{ E.place := E1.place; }

4. Translation Scheme for Boolean Expressions:

E → id1 relop id2

```
{  E.place := newtemp;
emit ('if' id1.place relop.op id2.place 'goto'
        nextstat+3);
emit ( E.place ':=' '0' );
emit ( 'goto' nextstat+2 );
emit ( E.place ':=' '1' );   }
```

nextstat: gives the index of the next three address statement in the output sequence

E → true

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

E → false

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

Translation of a<b or c<d and e<f

100: if a<b goto 103	E.place=t1
101: t1 := 0	
102: goto 104	
103: t1 := 1	
104: if c<d goto 107	E.place=t2
105: t2 := 0	
106: goto 108	
107: t2 := 1	
108: if e<f goto 111	E.place=t3
109: t3 := 0	
110: goto 112	
111: t3 := 1	
112: t4 := t2 and t3	E.place=t4
113: t5 := t1 or t4	E.place=t5

5. Flow Control Representation: (Short-Circuit Code)

- We can translate a Boolean expression into three address code without generating code for any of the Boolean operators and without having the code necessarily evaluate the entire expression. This is called Short-Circuit or Jumping code.

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

Figure 6.36: Syntax-directed definition for flow-of-control statements.

Translation of Boolean Expr Using short-circuit Code

- **Example 1:**
  a<b or c<d and e<f

101: if a<b goto 107
102: goto 103
103: if c<d goto 105
104: goto 108
105: if e<f goto 107
106: goto 108
107: true
108: false

Example 1:
a<b or c<d and e<f
if a<b goto Ltrue
goto L1
L1: if c<d goto L2
goto Lfalse
L2: if e<f goto Ltrue
goto Lfalse
Ltrue: ------
Lfalse: -----

Translation of Boolean Expr Using short-circuit Code

- Example 2:
  p<q and r<s or t<u

```
101: if p<q goto 103
102: goto 105
103: if r<s goto 107
104: goto 105
105: if t<u goto 107
106: goto 108
107: true
108: false
```

Example 2:
p<q and r<s or t<u

```
    if p<q goto L1
                goto L2
L1: if r<s goto Ltrue
                goto L2
L2: if t<u goto Ltrue
                goto Lfalse
Ltrue:  ----
Lfalse: -----
```

6. SDD for Boolean Expressions:

E → E1or E2	E1.true := E.true; E1.false := newlabel E2.true := E.true; E2.false := E.false E.Code := E1.Code || gen(E1.false ':') || E2.Code
E → E1 and E2	E1.true := newlabel E1.false := E.false E2.true := E.true; E2.false := E.false E.Code := E1.Code || gen(E1.true ':') || E2.Code
E → not E1	E1.true := E.false E1.false := E.true E.Code := E1.Code
E → (E1)	E1.true := E.true E1.false := E.false E.Code := E1.Code
E → id1 relop id2	E.Code:=gen ('if' id1.place relop.op id2.place 'goto' E.true) || gen ('goto' E.false)
E → true	E.Code := gen ('goto' E.true)
E → false	E.Code := gen ('goto' E.false)

3-Address Code Generation Control Statements

- **Topics:**
  - **Introduction**
  - **Syntax directed definition for flow of control statements**

- **1. Introduction**
  - Simulate the "branching" of the flow of control.
  - Skipping to different parts of the code.
  - Suppose we have the following grammar:-

    - S → if E then S1
    - S → if E then S1 else S2
    - S → while E do S1

3-Address Code Generation Control Statements

- In this translation scheme, a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.
- With Boolean expression E, associate with two labels:
  - E.true – label to which control flows if E is true
  - E.false – label to which control flows if E is false

3-Address Code Generation Control Statements

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

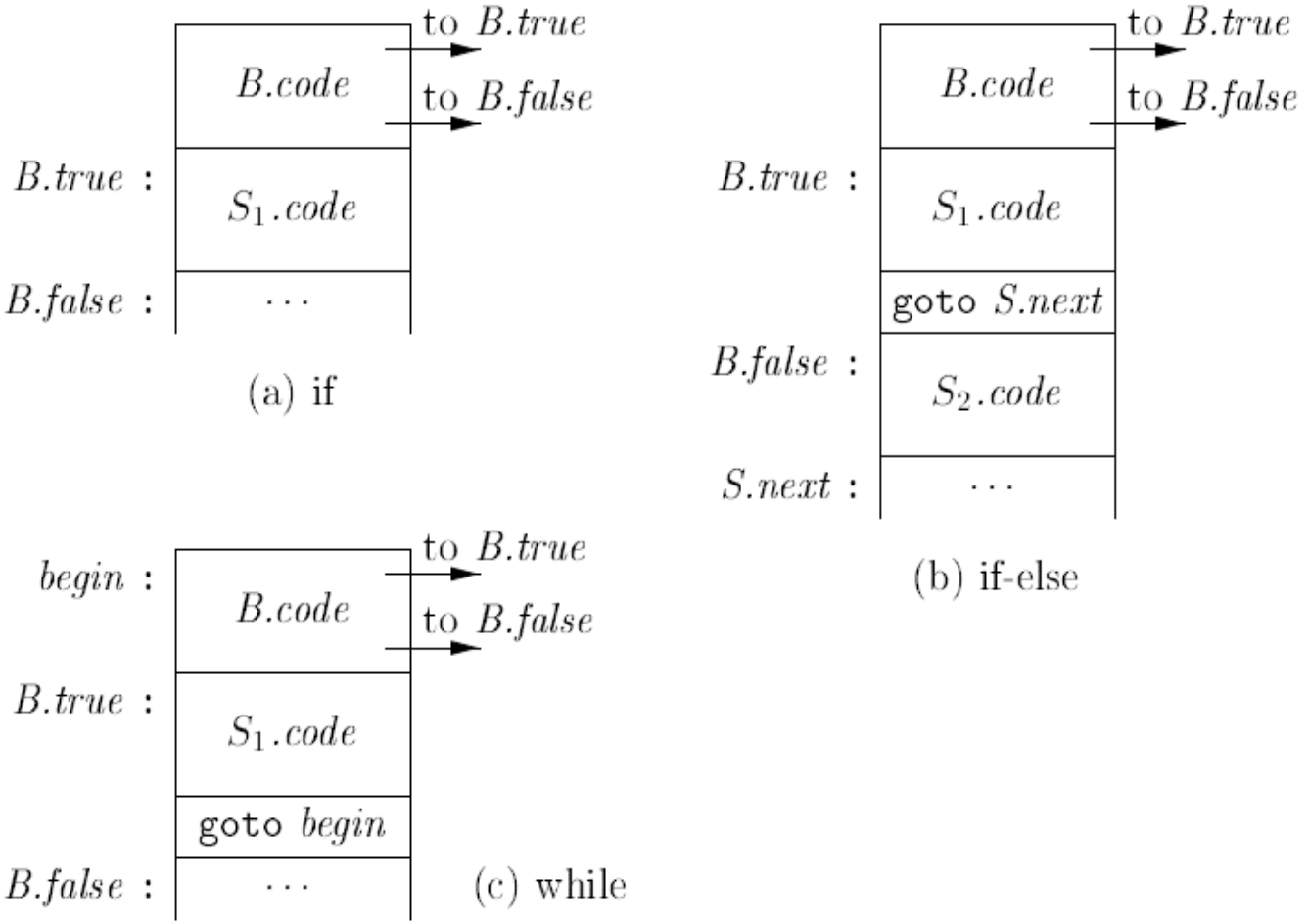


Figure 6.35: Code for if-, if-else-, and while-statements

Example 1:

- **Consider the Statement:**

```
while (i<10)
  {
      x=0;
      i = i+1;
  }
```

- Three Address code :

```
L1:   if i<10 goto L2
          goto Lnext
      L2: x = 0
          i = i+1
          goto L1
      Lnext:
```

Example 2:

- **Consider the Statement**

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y – z
```

- Three Address code :

```
L1:   if a < b goto L2
      goto Lnext

L2:   if c < d goto L3
      goto L4

L3:   t1 = y +z
      x = t1
      goto L1

L4:   t2 = y – z
      x = t2
      goto L1

Lnext:
```

Example 3:

Consider the statements:

```
a=3;
b=4;
i=0;
while (i<n)
{
a=b+1;
a=a*a;
i++;
}
c=a;
```

Three Address code :

```
a=3;
b=4;
i=0
L1: if (i<n)
goto L2
gotoLnext

L2: t1=b+1;
a=t1;
t2=a*a;
a=t2;
i++;
goto L1

Lnext: c=a;
```

2. SDD for 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:
```

3. Examples:

- Consider the Statement

```
switch ( ch )
{       case 1: c=a + b
                break;
        case 2: c=a - b
                break;
}
```

Three Address code :

```
        if ch = 1 L1
        if ch = 2 L2
L1:     t1 = a + b
        c = t1
        goto Next
L2:     t1 = a - b
        c = t1
        goto Next
Next
```

Example:

- **Translate executable Statements of the following into 3-Address Code :**

```
if (c+f)>(d-e) then
 do
   x=x+a[i]*b[i];
 while (i <20);
   else
      x = 0;
```

Three Address code :

1. t1 = c+f
2. t2 = d-e
3. If (t1>t2) goto 5
4. goto 17
5. t3 = 4 * i
6. t4 = addrA - 4
7. t5 = t4[t3]
8. t6 = 4 * i
9. t7 = addrA - 4
10. t8 = t7[t6]
11. t9 = t5 * t8
12. t10 = X
13. t11 = t10 + t9
14. X = t11
15. if (i<20) goto 5
16. goto 17
17. X = 0

Example:

- **Translate executable Statements of the following C-program into 3-Address Code**
- main()

```
{
  int i;
  int a[10];
  i = 1;
  while (i <=10)
    {
      a[i] = 0;
      i = i+1;
      }
  }
```

Three Address code :

1. i = 1
2. if (i<=10) goto 4
3. goto 10
4. t1 = 4 * i
5. t2 = addrA - 4
6. t2[t1] = 0
7. t3 = i+1
8. i = t3
9. goto 2
10. ----

Run-Time Environments

- This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters, and the interfaces to the operating system, input/output devices, and other programs.
- The two themes in this Run-Time Environment are the allocation of storage locations and access to variables and data.
- Memory management in some detail, including stack allocation, heap management, and garbage collection.

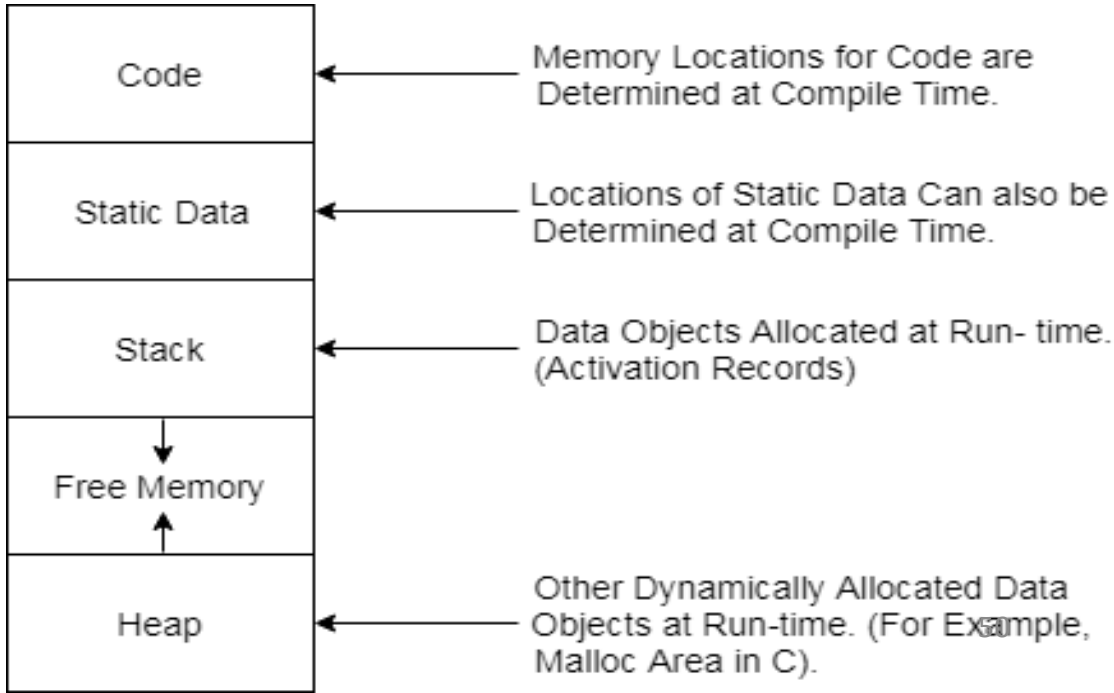
Run-Time Environments

- How do we allocate the space for the generated target code and the data object of our source programs?
- The places of the data objects that can be determined at compile time will be *allocated statically.*
- But the places for the some of data objects will be *allocated at run-time*.
- The allocation of de-allocation of the data objects is managed by the *run-time support package*.
  - run-time support package is loaded together with the generate target code.
  - the structure of the run-time support package depends on the semantics of the programming language (especially the semantics of procedures in that language).
- Each execution of a procedure is called as *activation of that procedure*.

Storage Organization

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

The run-time representation of an object program in the logical address space consists of data and program areas



- Run-time storage comes in blocks, where a byte is the smallest unit of memory. Four bytes form a machine word. Multibyte objects are bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

STORAGE ALLOCATION STRATEGIES

- Static Allocation: It is for all the data objects at compile time.
- Stack Allocation: Stack is used to manage the run time storage. For example recursive calls make use of this area.
- Heap Allocation: Heap is used to manage the dynamic memory allocation.

Static Allocation

- The size of data objects is known at compile time.
- The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
- The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.
- In this the compiler can determine the amount of storage required by each data object and therefore it becomes easy for a compiler to find the address of these data in the activation record.
- At compile time compiler can fill the address at which the target code can find the data it operates on.
- Used by simple or early programming languages. FORTRAN uses the static allocation.

Advantages of static allocation

- No stack manipulation or indirect access to names, i.e., faster in accessing variables.
- Values are retained from one procedure call to the next if block structure is not allowed.

  For example: static variables in C.

Disadvantages of static allocation

- Recursive procedures are not supported by this type of allocation.
- Waste lots of space when procedures are inactive
- No dynamic allocation. The static allocation cannot manage the allocation of memory at run time.

Stack Allocation

- Storage is organized as stack .This stack is also called control stack.
- As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.
- The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.
- The data structures can be created dynamically for stack allocation.

Limitation of stack allocation

- The memory addressing can be done using pointers an index registers. Hence this type of allocation is slower than static allocation.

Procedure Activations

- An execution of a procedure starts at the beginning of the procedure body;
- When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
- Each execution of a procedure is called as its activation.
- Lifetime of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
- If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested.
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.

Activation Tree

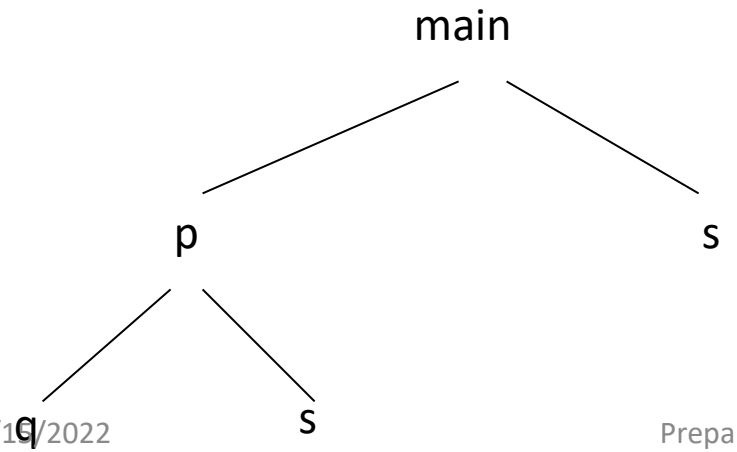
- We can use a tree (called activation tree) to show the way control enters and leaves activations.

In an activation tree:

- Each node represents an activation of a procedure.
- The root represents the activation of the main program.
- The node a is a parent of the node b iff the control flows from a to b.
- The node a is left to to the node b iff the lifetime of a occurs before the lifetime of b.

Activation Tree

```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```



enter main
enter p
enter q
exit q
enter s
exit s
exit p
enter s
exit s
exit main

A Nested Structure

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

2. The sequence of returns corresponds to a postorder traversal of the activation tree.

3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (live) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

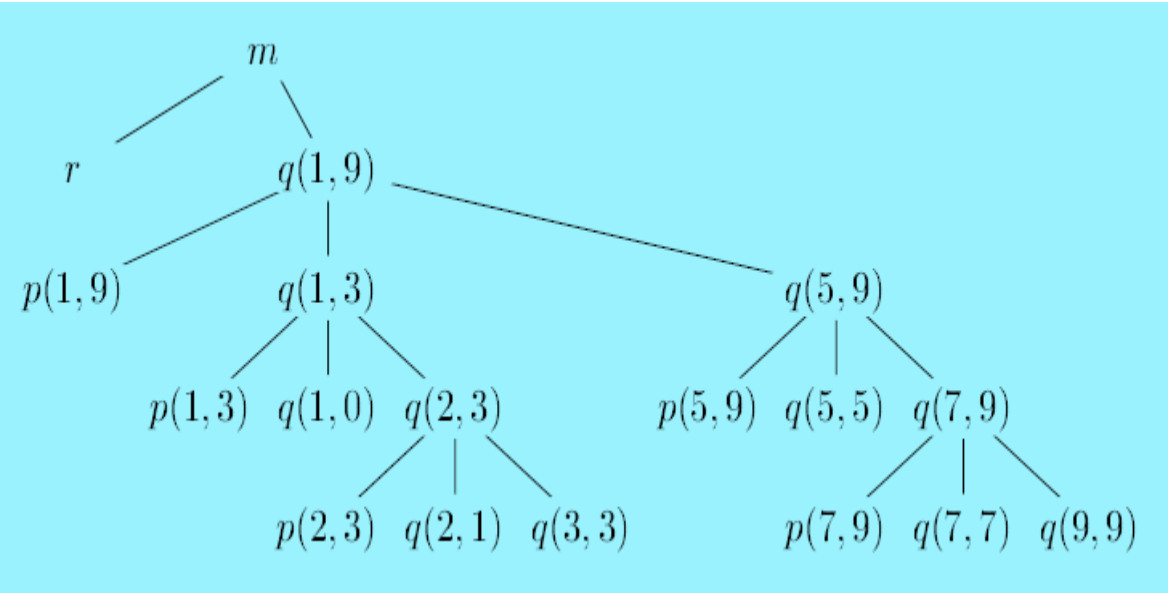
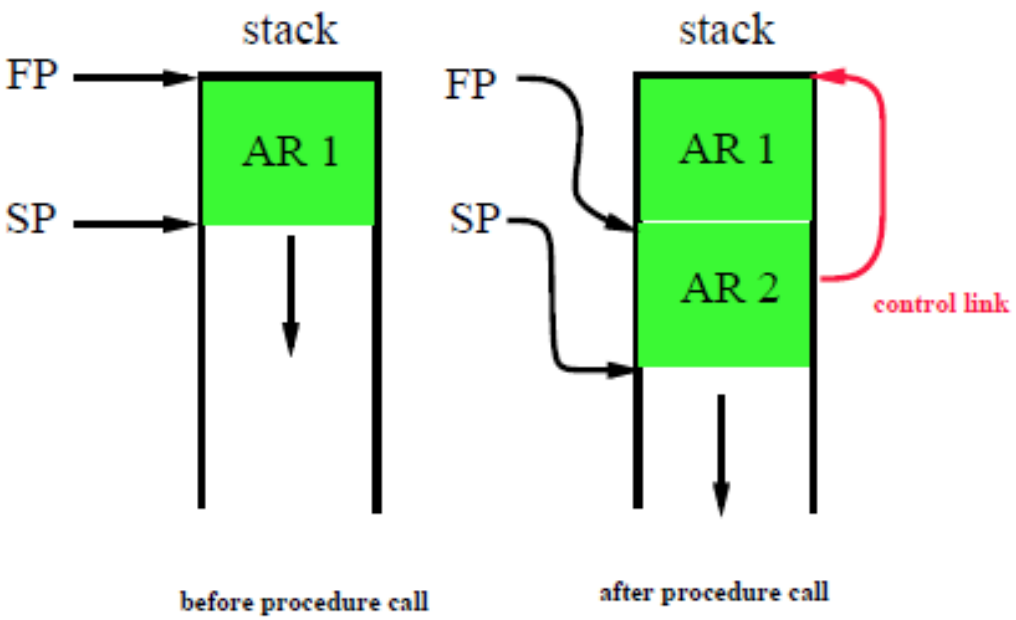


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

If control is currently in the activation q(2; 3) of the tree of Fig. 7.4, then the activation record for q(2; 3) is at the top of the control stack. Just below is the activation record for q(1; 3), the parent of q(2; 3) in the tree. Below that is the activation record q(1; 9), and at the bottom is the activation record for m, the main function and root of the activation tree.

Control Stack

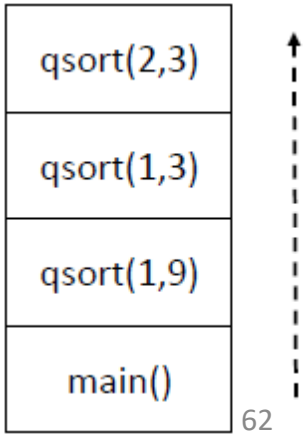
- Procedure calls and returns are usually managed by Control Stack
- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - ✓ starts at the root,
  - ✓ visits a node before its children, and
  - ✓ recursively visits children at each node an a left-to-right order.
- A stack (called control stack) can be used to keep track of live procedure activations.
  - ✓ An activation record is pushed onto the control stack as the activation starts.
  - ✓ That activation record is popped when that activation ends.
- When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.



before procedure call

after procedure call

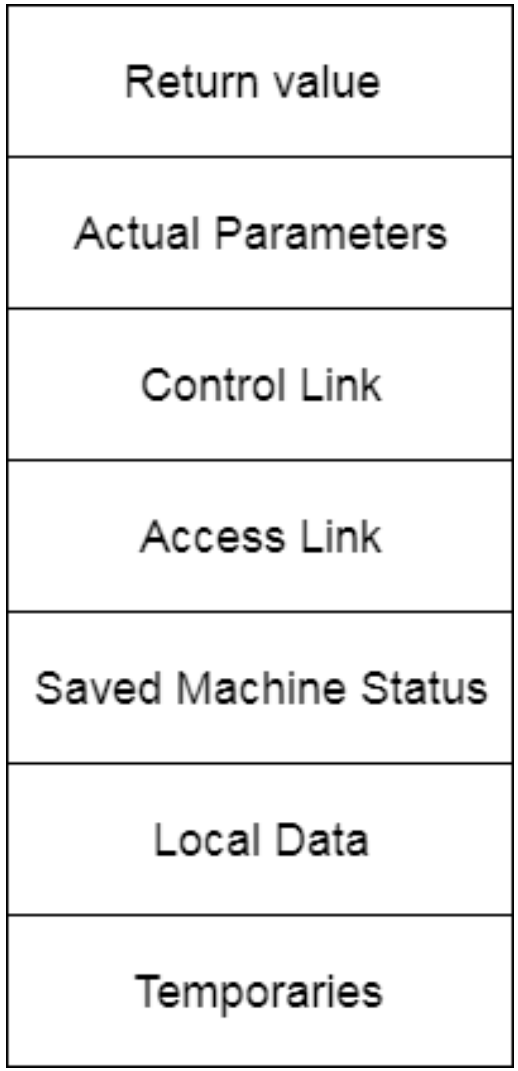
return from procedure call



Activation Record

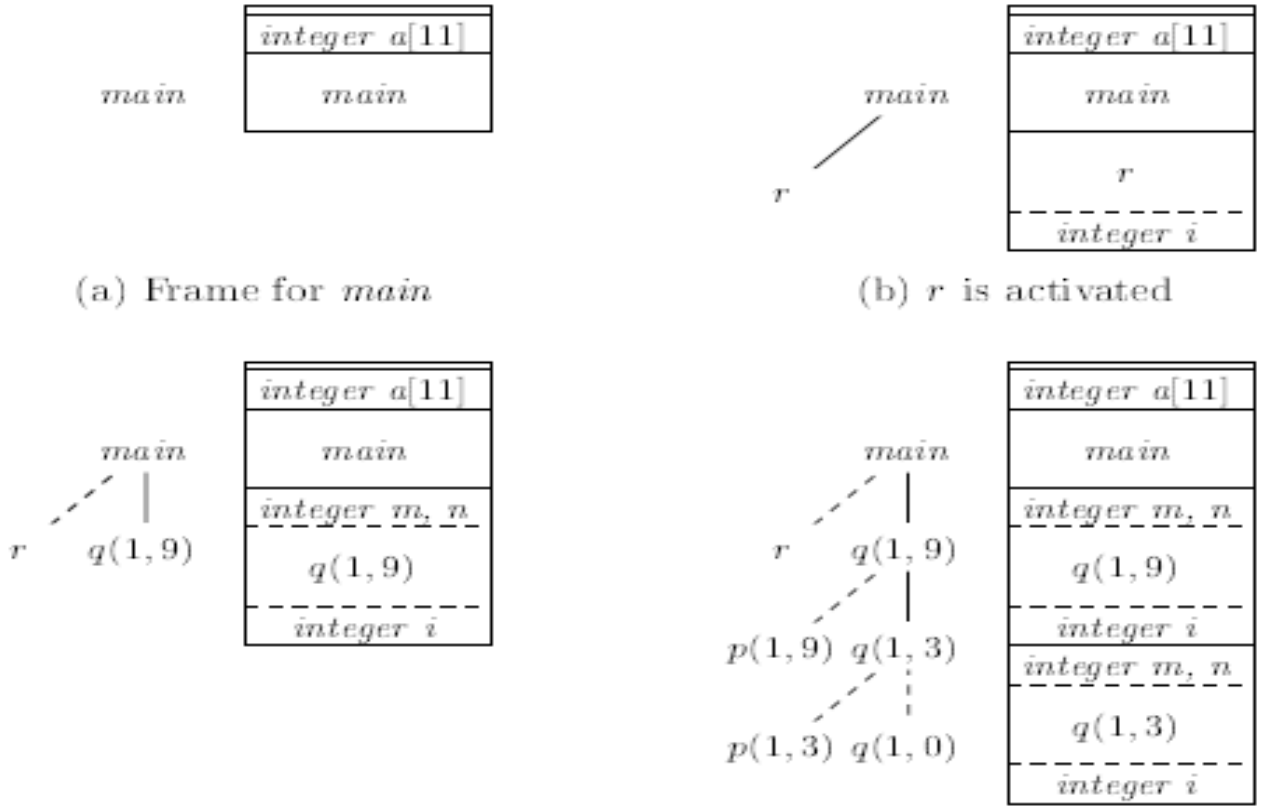
- Control stack is a run time stack - keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
- When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- Activation record is used to manage the information needed by a single execution of a procedure.
- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.
- Size of each field can be determined at compile time (Although actual location of the activation record is determined at run-time).
  - ✓ Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.

Activation Record



1. **Return Value:** It is used by calling procedure to return a value to calling procedure.
2. **Actual Parameter:** It is used by calling procedures to supply parameters to the called procedures.
3. **Control Link:** It points to activation record of the caller.
4. **Access Link:** It is used to refer to non-local data held in other activation records.
5. **Saved Machine Status:** It holds the information about status of machine before the procedure is called.
6. **Local Data:** It holds the data that is local to the execution of the procedure.
7. **Temporaries:** It stores the value that arises in the evaluation of an expression.

Example 7.4: Figure 7.6 shows snapshots of the run-time stack as control flows through the activation tree of Fig. 7.4. Dashed lines in the partial trees go to activations that have ended. Since array a is global, space is allocated for it before execution begins with an activation of procedure *main*, as shown in Fig. 7.6(a).



(a) Frame for *main*

(b) r is activated

(c) r has been popped and $q(1, 9)$ pushed

(d) Control returns to $q(1, 3)$

Figure 7.6: Downward-growing stack of activation records

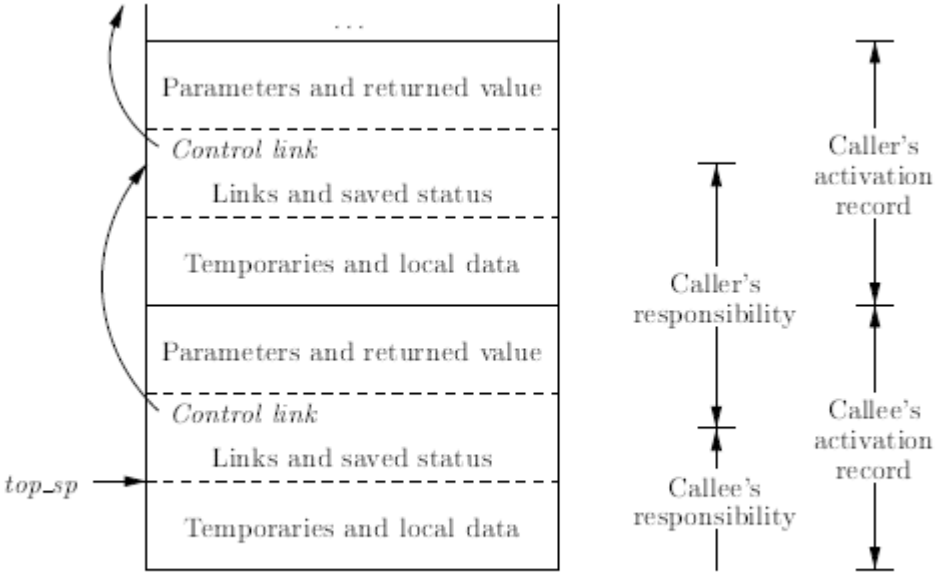


Figure 7.7: Division of tasks between caller and callee

The calling sequence and its division between caller and callee are as follows:

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments top_sp to the position
3. The callee saves the register values and other status information.
4. The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters,
2. Using information in the machine-status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller therefore may use that value.

Variable-length local data

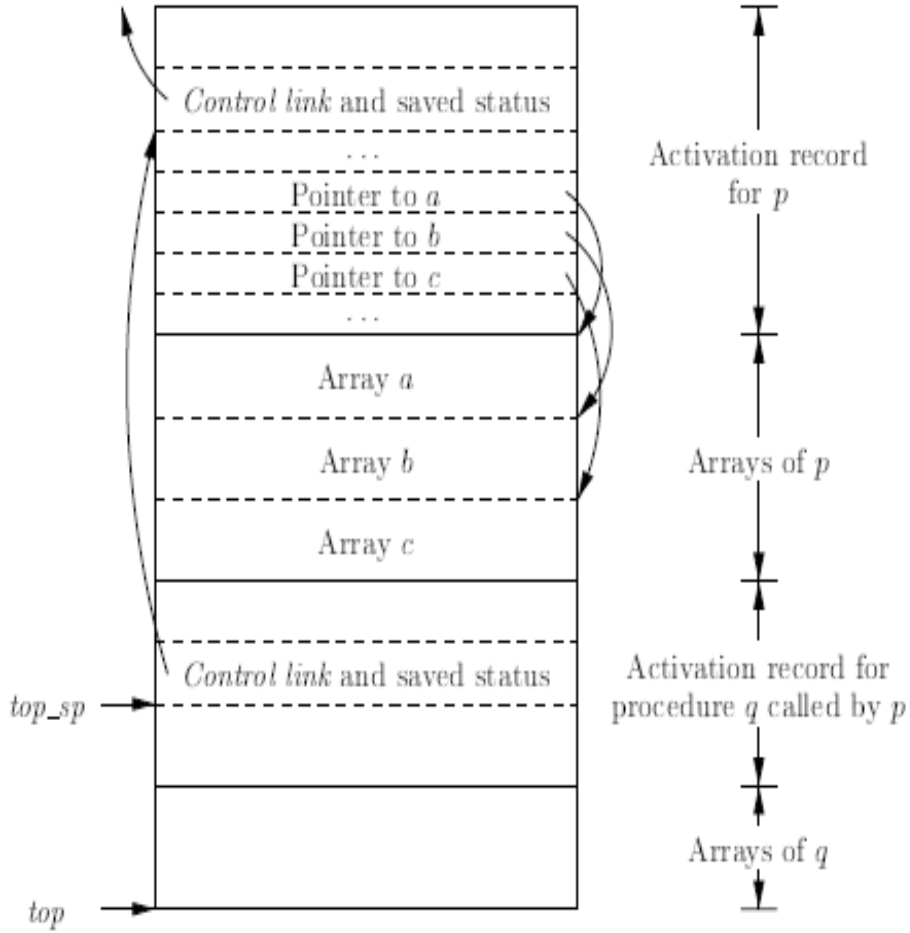


Figure 7.8: Access to dynamically allocated arrays

Allocation of space for objects the sizes of which are not known at compile time.

- Example: Arrays whose size depends on the value of one or more parameters of the called procedure.
- Cannot calculate proper offset if they are allocated on the A.R.

Strategy:

- Allocate these objects at the bottom of A.R.
- Automatically de-allocated when the procedure is returned.
- Keep a pointer to such an object inside the local data area.
- Need to de-reference this pointer whenever it it used.

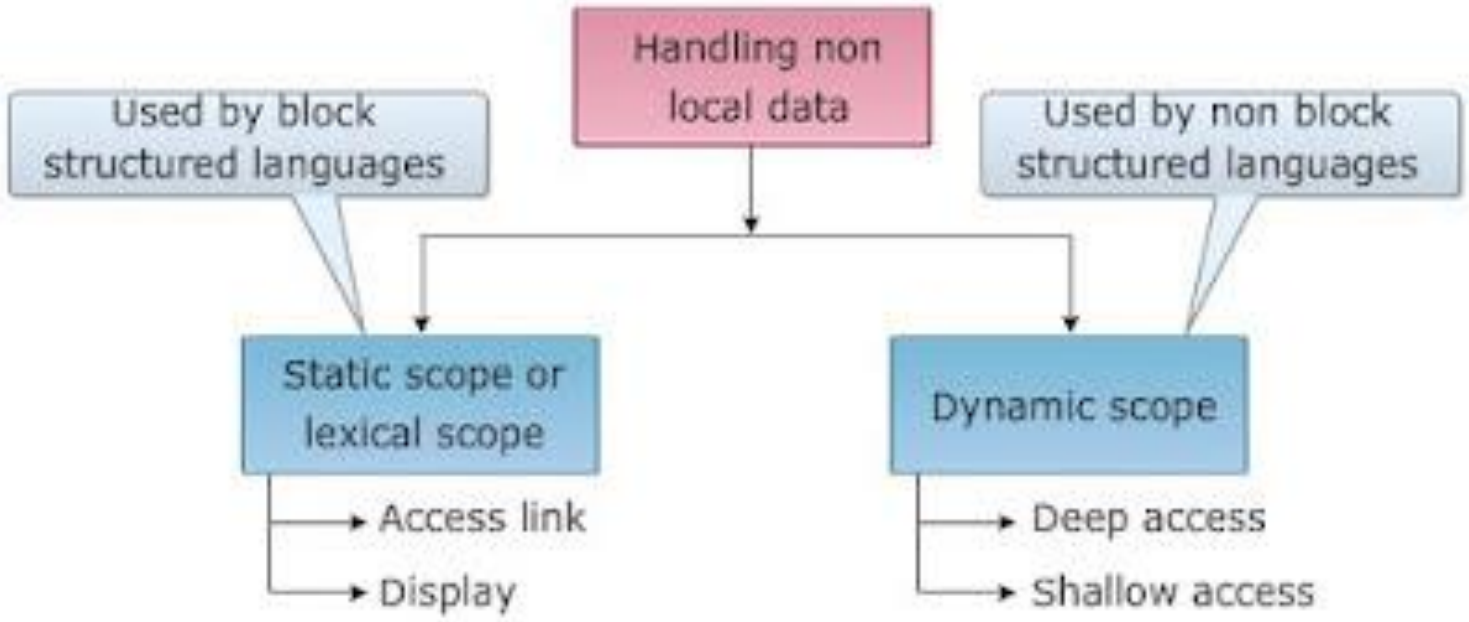
Access to Nonlocal Data on the Stack

In some cases, when a procedure refer to variables that are not local to it, then such variables are called nonlocal variables

There are two types of scope rules, for the non-local names. They are

Static scope

Dynamic scope



Static Scope or Lexical Scope

- Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program.
- Examples: PASCAL, C and ADA are the languages that use the static scope rule.
- These languages are also called block structured languages

Block

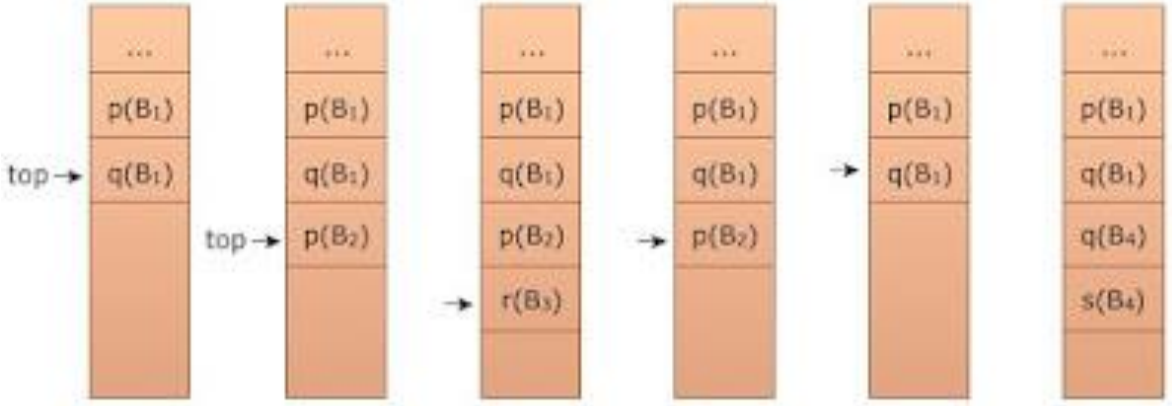
- A block defines a new scope with a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

Example:

```
{
Declaration statements
…….
}
```

Static Scope or Lexical Scope

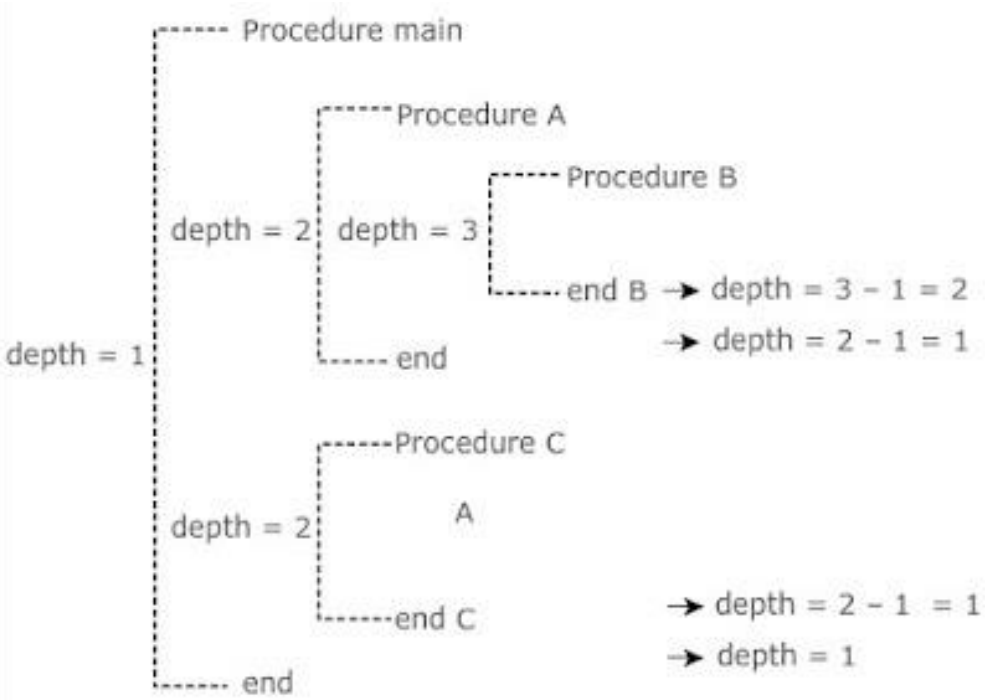
- The beginning and end of the block are specified by the delimiter. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1
- In a block structured language, scope declaration is given by static rule or most closely nested loop
- At a program point, declarations are visible
  - The declarations that are made inside the procedure
  - The names of all enclosing procedures
  - The declarations of names made immediately within such procedures
- The displayed image on the screen shows the storage for the names corresponding to particular block
- Thus, block structure storage allocation can be done by stack



Lexical Scope for Nested Procedure

- If a procedure is declared inside another procedure then that procedure is known as nested procedure
- A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

Procedure main
Procedure P1
Procedure P2
Procedure P3
Procedure P4



Nesting Depth:

Lexical scope can be implemented by using nesting depth of a procedure.
The procedure of calculating nesting depth is as follows:

- The main programs nesting depth is '1'
- When a new procedure begins, add '1' to nesting depth each time
- When you exit from a nested procedure, subtract '1' from depth each time

Algorithm for setting the links

- The control link is set to point to the A.R. of the calling procedure.
- How to properly set the access link at compile time?
  - Procedure P at depth n_P calls procedure X at depth n_X:
  - If $n_P < n_X$, then X is enclosed in P and $n_P = n_X - 1$.
    - ▷ *Same with setting the control (dynamic) link.*
  - If $n_P \geq n_X$, then it is either a recursive call or calling a previously declared procedure.
    - ▷ *Observation: go up the access (static) link once, then the depth is decreased by 1.*
    - ▷ *Hence, the access (static) link of X is the access link of P going up $n_P - n_X + 1$ times.*
  - Content of the access (static) link in the A.R. for procedure P:
    - ▷ *Points to the A.R. of the procedure Q who encloses P lexically.*
    - ▷ *An A.R. of Q must be active at this time.*
    - ▷ *Several A.R.'s of Q (recursive calls) may exist at the same time, it points to the latest activated one.*

Heap Allocation

- If the values of non local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last in First Out nature. For retaining of such local variables heap allocation strategy is used.
- The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object, this allocated memory can be deallocated when activation ends. This deallocated space can be further reused by heap manager.
- The efficient heap management can be done by
  - Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.
  - Allocate the most suitable block of memory from the linked list i.e. use best fit technique for allocation of block.

Parameter passing techniques

There are two types of parameters:

- ❑ Formal parameters
- ❑ Actual parameters

Based on these parameters there are various parameter passing methods.

The most common methods are:

1. Call by value:

- ➢ This is the simplest method of parameter passing.
- ➢ The actual parameters are evaluated and their r-values are passed to called procedure.
- ➢ The operations on formal parameters do not changes the values of actual parameter.
- ➢ **Example:** Languages like C, C++ use actual parameter passing method.

2. Call by reference:

- This method is also called as call by address or call by location.
- The L-value, the address of actual parameter is passed to the called routines activation record.
- The value of actual parameters can be changed.
- The actual parameter should have an L-value.
- Example: Reference parameter in C++,PASCAL'S var parameters.

3. Copy restore:

- This method is a hybrid between call by value and call by reference.
- This is also known as copy-in-copy-out or values result.
- The calling procedure calculates the value of actual parameters and it is then copied to activation record for the called procedure.
- During execution of called procedure, the actual parameters value is not affected.
- If the actual parameters have L-value then at return the value of formal parameter is copied to actual parameter.
- **Example: In Ada this parameter passing method is used.**

4. Call by name:

- This is less popular method of parameter passing.
- Procedure is treated like macro.
- The procedure body is substituted for call in caller with actual parameters substituted for formals.
- The actual parameters can be surrounded by parenthesis to preserve their integrity.
- The locals name of called procedure and names of calling procedure are distinct.
- **Example:** ALGOL uses call by name method.

Language Facilities for dynamic storage allocation

- Explicit and Implicit allocation of memory to variables
  - Most languages support dynamic allocation of memory.
  - Pascal supports new(p) and dispose(p) for pointer types.
  - C provides malloc() and free() in the standard library.
  - C++ provides the new and free operators.
  - These are all examples of EXPLICIT allocation.
  - Other languages like Python and Lisp have IMPLICIT allocation.
- Garbage – Finding variables that are not referred by the program any more
  - In languages with explicit deallocation, the programmer must be careful to free every dynamically allocated variable, or GARBAGE will accumulate.
  - Garbage is dynamically allocated memory that is no longer accessible because no pointers are pointing to it.
  - In some languages with implicit deallocation, GARBAGE COLLECTION is occasionally necessary.
  - Other languages with implicit deallocation carefully track references to allocated memory and automatically free memory when nobody refers to it any longer

Symbol table and implementation:

- Symbol table is an important data structure used in a compiler.
- Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.
- The symbol table used for following purposes:
  - It is used to store the name of all entities in a structured form at one place.
  - It is used to verify if a variable has been declared.
  - It is used to determine the scope of a name.
  - It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.
- A symbol table can either be linear or a hash table.

Symbol table and implementation:

Using the following format, it maintains the entry for each name.

<symbol name, type, attribute>

For example, suppose a variable store the information about the following variable declaration:

static int salary

then, it stores an entry in the following format:

<salary, **int**, **static**>

The clause attribute contains the entries related to the name.

Implementation

- The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.
- A symbol table can be implemented in one of the following techniques:
  - Linear (sorted or unsorted) list
  - Hash table
  - Binary search tree
- Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

int x;

1.Should be processed by the compiler as: **insert (x, int)**

lookup()

- In the symbol table, lookup() operation is used to search a name. It is used to determine:

  - The existence of symbol in the table.
  - The declaration of the symbol before it is used.
  - Check whether the name is used in the scope.
  - Initialization of the symbol.
  - Checking whether the name is declared multiple times.
  - The basic format of lookup() function is as follows:

- **lookup (symbol)**
- This format is varies according to the programming language.

Data structure for symbol table

A compiler contains two type of symbol table: global symbol table and scope symbol table.

Global symbol table can be accessed by all the procedures and scope symbol table.

The scope of a name and symbol table is arranged in the hierarchy structure as shown below: