## Unit IV: Part-1

Intermediate Code Generation:

Variants of Syntax Trees & Three-Address Code

Types and Declarations and Translation of Expressions

Type Checking

Control Flow

Back patching

Switch-statements

Intermediate Code for Procedures

## Part-2

Run-time environment: Stora

Stack Allocation of Space

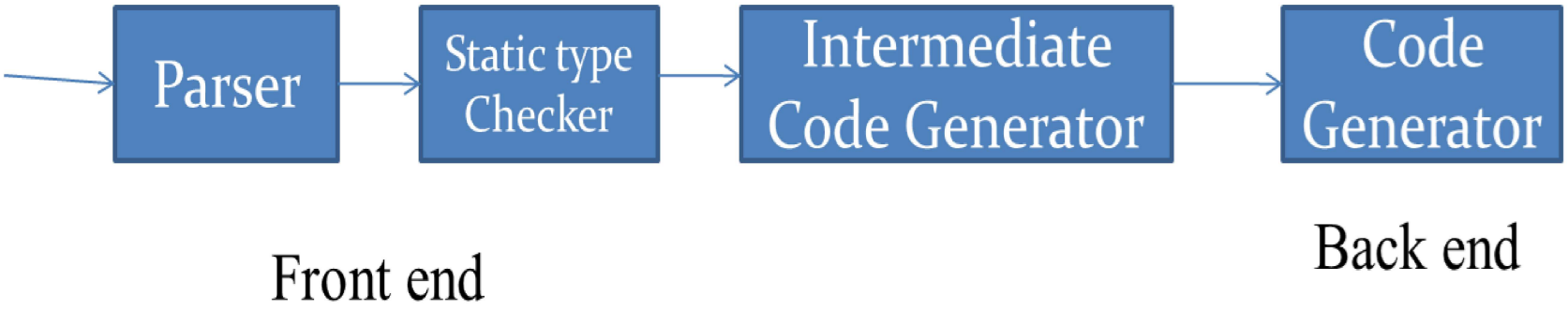
Access to Nonlocal Data on t

Parameter passing

Heap Management and Garba

# INTERMEDIATE CODE GENERATION

The front end translates a source program into an intermediate from which the back end generates target code.




**Benefits of Intermediate Code:**

- Retargeting
- Compiler for different source languages (on same machine) can be created by different front ends.
- A machine independent code optimizer can be applied

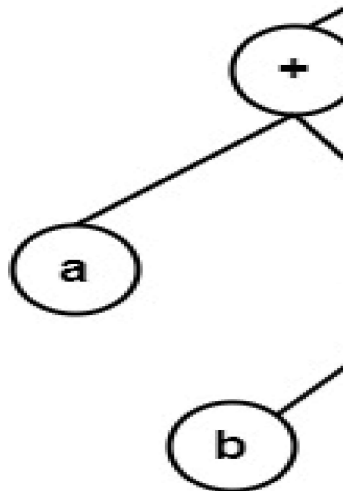
# Forms of Intermediate Code

1. Abstract syntax tree
2. Polish notation
3. Three Address code

## 1. Abstract Syntax Tree:

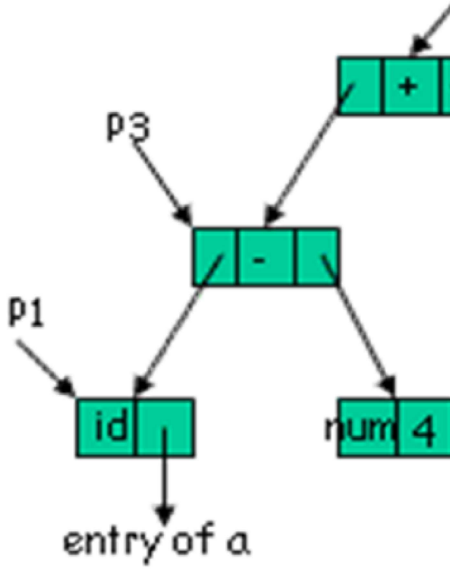
It is a condensed form of parse trees. In an abstract syntax tree, each lea
describes an operand, and each interior node represents an operator.

Example: Abstract Syntax tree for the string : a + b ∗ c − d.



# SDT For Abstract Syntax Tree

| | |
|---|---|
| E -> E1 + T | E.node=new node('+', E1.node,T.node) |
| E -> E1 - T | E.node=new node('-', E1.node,T.node) |
| E -> T | E.node = T.node |
| T -> (E) | T.node = E.node |
| T -> id | T.node = new Leaf(id,id.entry) |
| T -> num | T.node = new Leaf(num,num.val) |

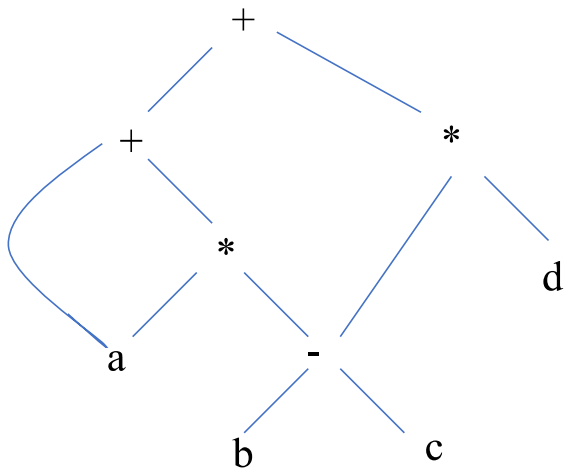


# Variants of syntax trees: DAG

**Directed Acyclic Graph (DAG)**

- It is sometimes beneficial to crate a DAG instead of tree for Expressions
- This way we can easily show the common sub-expressions and then use that knowledge during code generation

Example: a+a*(b-c)+(b-c)*d



# Forms of Intermediate Code

## 3. Three Address code

→ there is at most one operator at the right side of an instruction

Example: a+a*(b-c)+(b-c)*d

```
t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

# Forms of Intermediate Code

## 2. Polish notation

- It is a linearization of abstract syntax tree.
- Most natural way of representing an expression evaluation
- It is also called as prefix expression (operator operand1 operand2)
- In this representation operator can be easily associated with the corresponding operands.

**Example:**

(a+b)*(c-d) => *(+ab)(-cd)

*+ab-cd

Reverse Polish notation (called as postfix expression) Example:

(a+b)*(c-d) => (ab+)(cd-)*

ab+cd-*

# Three Address code: Advantages

Three-address code is a sequence of statements of the general form

x : = y op z

where x, y and z are names, constants, or compiler-generated temporaries; op sta operator,

Example : x+ y*z t1 : = y * z

t2 : = x + t1

**Example: a+a*(b-c)+(b-c)*d**

t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

**Advantages of three-address code:**

→The unraveling of complicated arithmetic expressions and of nested flow-of-contro makes three-address code desirable for target code generation and optimization.

→The use of names for the intermediate values computed by a program allows three- a to be easily rearranged – unlike postfix notation.

# Three Address code: Forms

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and ifFalse x goto L
- if  x relop y goto L
- Procedure calls using:

  param x

  call p,n

  y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

# Three Address code: Example

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

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

Symbolic labels

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

Position numbers

# Three Address code: Implementation

**Quadruples**

Has four fields: op, arg1, arg2 and result

**Triples**

Temporaries are not used and instead references to instructions a

**Indirect triples**

In addition to triples we use a list of pointers to triples

# Three Address code: Implementation

Example: b * minus c + b * minus c

## Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

## Quadruples

| op | arg1 | arg2 | result |
|---|---|---|---|
| minus | c | | t1 |
| * | b | t1 | t2 |
| minus | c | | t3 |
| * | b | t3 | t4 |
| + | t2 | t4 | t5 |
| = | t5 | | a |

## Triples

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

## Indirect Triples

| | op |
|---|---|
| 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) |

# Types and Declarations

- Data type of a identifier and type checking of the expression is to be done before ICG
- The applications of types can be grouped under type checking and type translation.
- Type checking uses logical rules to reason about the behaviour of a program at run t

**From the type of a name/identifier,**

- A compiler can determine the storage that will be needed for that identifier at r

**Type information is also needed**

- to calculate the address denoted by an array reference,
- to insert explicit type conversions, and
- to choose the right version of an arithmetic operator, among other things.

Programming Languages supports two types of data types:

**Basic types:** integer, character, real, Boolean.
**Derived types**: Array, structure(record), set and pointer derived types are built by usin

# Declarations

Types and Declarations are specified by using a simplified grammar that dec
one name at a time.

**D -> T id ; D | ϵ**
**T -> B C | record '{' D '}'**
**B -> int | float**
**C -> ϵ | [ num ] C**

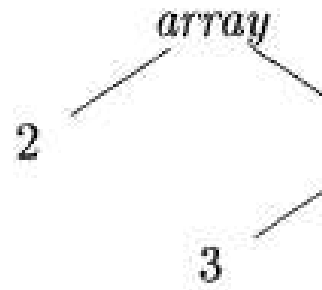
- Nonterminal D generates a sequence of declarations.
- Nonterminal T generates basic, array, or record types.
- Nonterminal B generates one of the basic types int and float.
- Nonterminal C generates strings of zero or more integers, each integer surr
  by brackets. An array type consists of a basic type specified by B, followed b
  components specified by nonterminal C.
- A record type (the second production for T) is a sequence of declarations
  fields of the record, all surrounded by curly braces.

# Type Expressions

- Types have structure, represented by using type expressions.
- A type expression is either a basic type or it is formed by applying an opera a type constructor to a type expression.
- Type constructors are derived types (array, structure, pointer, function)

**Example**: int[2][3]

array(2,array(3,integer))




- A basic type is a type expression
- A type name is a type expression
- A type expression can be formed by applying the array type c number and a type expression.
- A record is a data structure with named field
- A type expression can be formed by using the type constructor → for function types
- If s and t are type expressions, then their Cartesian product s*t is a type expression
- Type expressions may contain variables whose values are type expressions

# Type Equivalence

**General Form:**

"if two type expressions are equal then return a certain type else error."

Ambiguities arise when names are given to type expressions and the names are subsequent type expressions.
The key issue is whether a name in a type expression stands for itself or wh abbreviation for another type expression.

**When are two type expressions equivalent?**

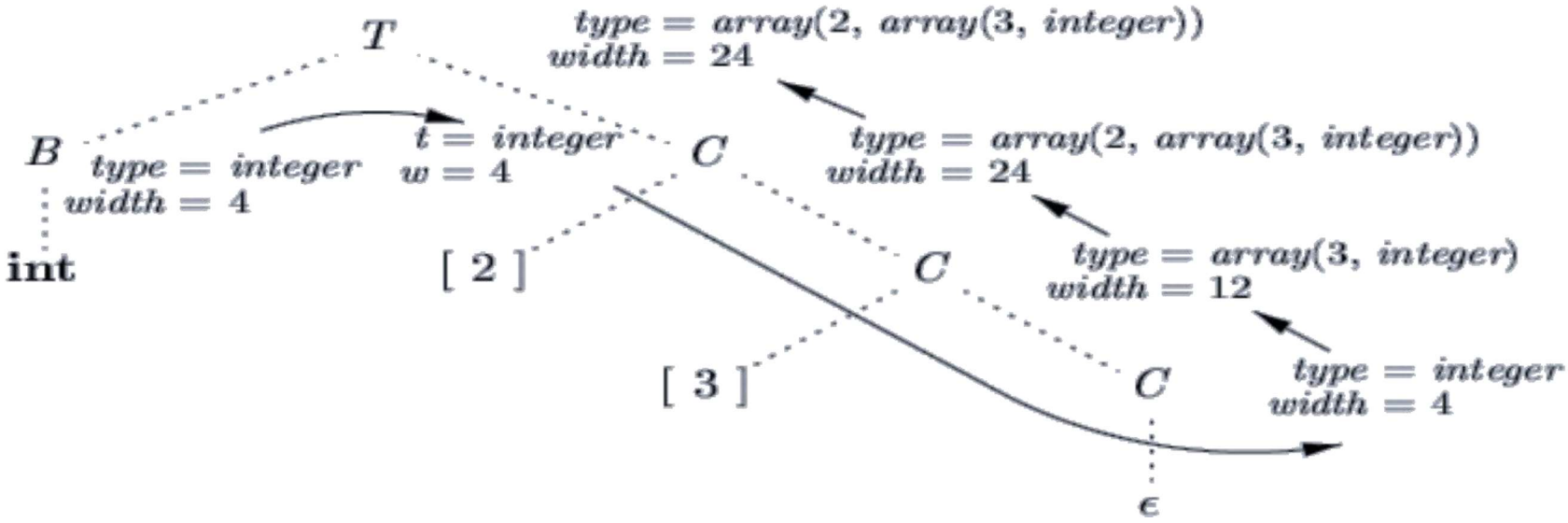
- If they are of the same basic type.
- If they are formed by applying the same constructor to structurally equivalentty
- One is a type name that denotes the other.

# Storage Layout for Local Names

- From the type of a name, we can determine the amount of storage that is needed
- At compile time, these amounts are used to assign each name a relative address.
- The type and relative address are saved in the symbol-table entry for the name.
- Data of varying length, such as strings, or data whose size cannot be determi time, such as dynamic arrays, is handled by reserving a known fixed amount of pointer to the data.
- The width of a type is the number of storage units needed for objects of that t
- Abasic type requires an integral number of bytes.
- For easy access, storage for aggregates such as arrays and classes are allo contiguous block of bytes

# Storage Layout for Local Names

| | | | |
|---|---|---|---|
| $T$ | $\rightarrow$ | $B$ | $\{\ t = B.type;\ w = B.width;\ \}$ |
| | | $C$ | $\{\ T.type = C.type;\ T.width = C.width;\ \}$ |
| $B$ | $\rightarrow$ | **int** | $\{\ B.type = integer;\ B.width = 4;\ \}$ |
| $B$ | $\rightarrow$ | **float** | $\{\ B.type = float;\ B.width = 8;\ \}$ |
| $C$ | $\rightarrow$ | $\epsilon$ | $\{\ C.type = t;\ C.width = w;\ \}$ |
| $C$ | $\rightarrow$ | [ **num** ] $C_1$ | $\{\ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$ |



# Translation of Expressions

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

Example Operational expressions:

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

# Translation of ARRAYS

Array elements can be accessed quickly if they are stored in a block of consecutiv
(Contiguous). (row major order).

If the width of each array element is w, then the $i_{th}$ element of array A is in location

**base + i * w**

In two-dimensional array, the relative address of A[i1][i2] can be calculated by the formu

**base + $i_1$ * $w_1$ + $i_2$ * $w_2$**

- Compile-time pre-calculation can also be applied to address calculations for multidimensional arrays.
- However, if the array's size is dynamic, we cannot use compile-time pre-calculation.

# Translation of ARRAYS

A grammar for array references:

L -> L [ E ] | id [ E ]

Nonterminal L has three synthesized attributes:

- **L.addr:** a temporary, used while computing the offset of array reference by summing the
- **L.array:** L.array is a pointer to the symbol-table entry for the array name.
- **L.type:** L.type is the type of the subarray generated by L.

| | | |
|---|---|---|
| $S \rightarrow$ | $\mathbf{id} = E\ ;$ | $\{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ |
| $\mid$ | $L = E\ ;$ | $\{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.$ |
| $E \rightarrow$ | $E_1 + E_2$ | $\{\ E.addr = \mathbf{new}\ Temp\ ();$<br>$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.add$ |
| $\mid$ | $\mathbf{id}$ | $\{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}$ |
| $\mid$ | $L$ | $\{\ E.addr = \mathbf{new}\ Temp\ ();$<br>$gen(E.addr\ '='\ L.array.base\ '['\ L.a$ |
| $L \rightarrow$ | $\mathbf{id}\ [\ E\ ]$ | $\{\ L.array = top.get(\mathbf{id}.lexeme);$<br>$L.type = L.array.type.elem;$<br>$L.addr = \mathbf{new}\ Temp\ ();$<br>$gen(L.addr\ '='\ E.addr\ '*'\ L.type.wi$ |
| $\mid$ | $L_1\ [\ E\ ]$ | $\{\ L.array = L_1.array;$<br>$L.type = L_1.type.elem;$<br>$t = \mathbf{new}\ Temp\ ();$<br>$L.addr = \mathbf{new}\ Temp\ ();$<br>$gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \}$<br>$gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$ |

# Translation of CONTROL FLOW

**Compute logical values.**

- A boolean expression can represent true or false as values.
- Such boolean expressions can be evaluated in analogy to arithmetic expressio three-address instructions with logical operators.

**Alter the flow of control**

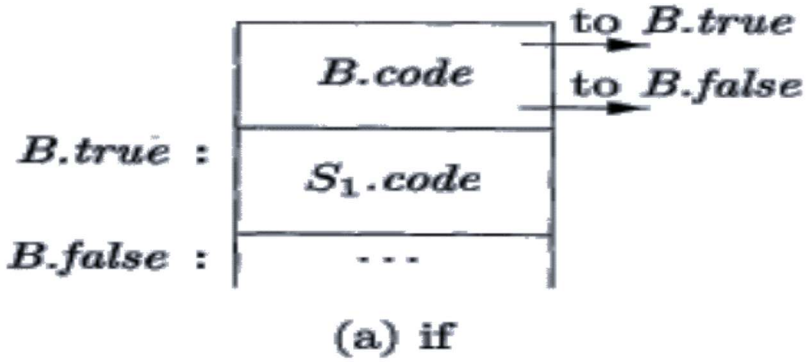
- Boolean expressions are used as conditional expressions in statements that a flow of control.
- The value of such boolean expressions is implicit in a position reached in a p For example, in if (E) S, the expression E must be true if statement Sis reached

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $B \rightarrow B_1 \;\texttt{||}\; B_2$ | $B_1.true = B.true$<br>$B_1.false = newlabel()$<br>$B_2.true = B.true$<br>$B_2.false = B.false$<br>$B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$ |
| $B \rightarrow B_1 \;\texttt{\&\&}\; B_2$ | $B_1.true = newlabel()$<br>$B_1.false = B.false$<br>$B_2.true = B.true$<br>$B_2.false = B.false$<br>$B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$ |
| $B \rightarrow \texttt{!}\, B_1$ | $B_1.true = B.false$<br>$B_1.false = B.true$<br>$B.code = B_1.code$ |
| $B \rightarrow E_1 \;\textbf{rel}\; E_2$ | $B.code = E_1.code \;\|\|\; E_2.code$<br>$\|\| \; gen(\texttt{'if'}\; E_1.addr\; \textbf{rel}.op\; E_2.addr\; \texttt{'goto'}\; B.true)$<br>$\|\| \; gen(\texttt{'goto'}\; B.false)$ |
| $B \rightarrow \textbf{true}$ | $B.code = gen(\texttt{'goto'}\; B.true)$ |
| $B \rightarrow \textbf{false}$ | $B.code = gen(\texttt{'goto'}\; B.false)$ |

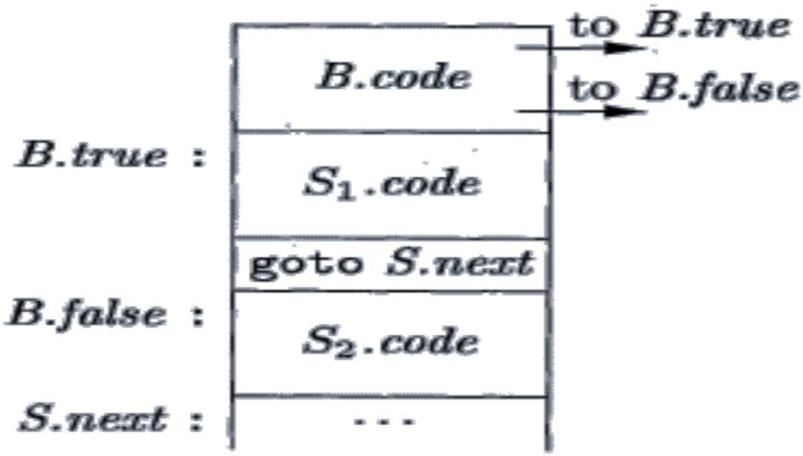
# FLOW OF CONTROL STATEMENTS

S -> if (B) S1
| if (B) S1 else S2
| while (B) S1

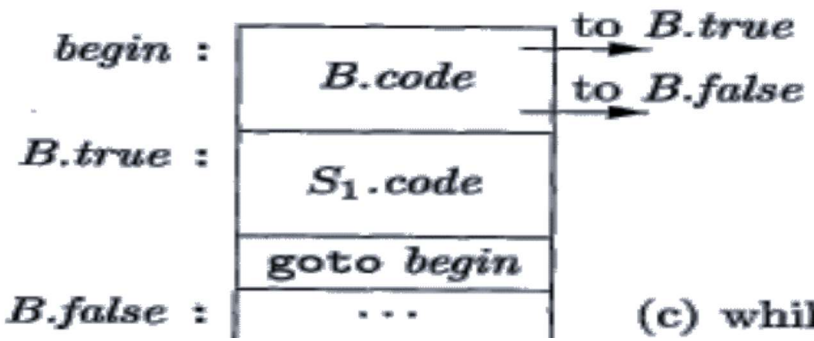
B.true := newlabel()
B.false := S.next
S1.next:= S.next
S.code:= B.code || label(B.true) || S1.cod



(a) if



(b) if-else



(c) while

# Type Checking

$E \rightarrow$ **id** { E.type=lookup(id.entry) }

$E \rightarrow$ **charliteral** { E.type=char }

$E \rightarrow$ **intliteral** { E.type=int }

$E \rightarrow$ **realliteral** { E.type=real }

$E \rightarrow E_1 + E_2$ { if ($E_1$.type=int and $E_2$.type=int) then E.type=int
else if ($E_1$.type=int and $E_2$.type=real) then E.type=rea
else if ($E_1$.type=real and $E_2$.type=int) then E.type=rea
else if ($E_1$.type=real and $E_2$.type=real) then E.type=re
else E.type=type-error }

$E \rightarrow E_1 [E_2]$ { if ($E_2$.type=int and $E_1$.type=array(s,t)) then E.type=t
else E.type=type-error }

$E \rightarrow E_1 \uparrow$ { if ($E_1$.type=pointer(t)) then E.type=t
else E.type=type-error }

Type checking can take one of the two forms:

- **synthesis**
- **Inference**

# Type Checking: synthesis

**Type synthesis** builds the type of an expression from the types of its subexpressions. requires names to be declared before they are used.

The type of E1 + E2 is defined in terms of the types of E1 and E2.

A typical rule for type synthesis has the form

**if f has type s -> t and x has type s,**

**then expression f(x) has type t**

Here, f and x denote expressions, and s -> t denotes a function from s to t.

This rule is for functions with one argument and to functions with several arguments.

The rule can be adapted for E1 + E2 by viewing it as a function application

**add(E1, E2). => E.type = add(E1, E2).**

# Type Checking: Inference

determines the type of a language construct from the way it is used. In programming some statements do not have values. So, void type is used.

A typical rule for type inference has the form

**if f(x) is an expression,**

**then for some α and β, f has type α -> β and x has type α**

Statements:

The rules for checking statements are similar to those for expressions.

For example, we treat the conditional statement “if (E) S;" as if it were the application of a to E and S.

**S -> if (E) S1 : if E.type = boolean true, S.type = S1.type**

Then function if expects to be applied to a boolean and a void; the result of the application i

# TYPE CONVERSION

→Converting one type to another type of data

Example, the integer 2 is converted to a float

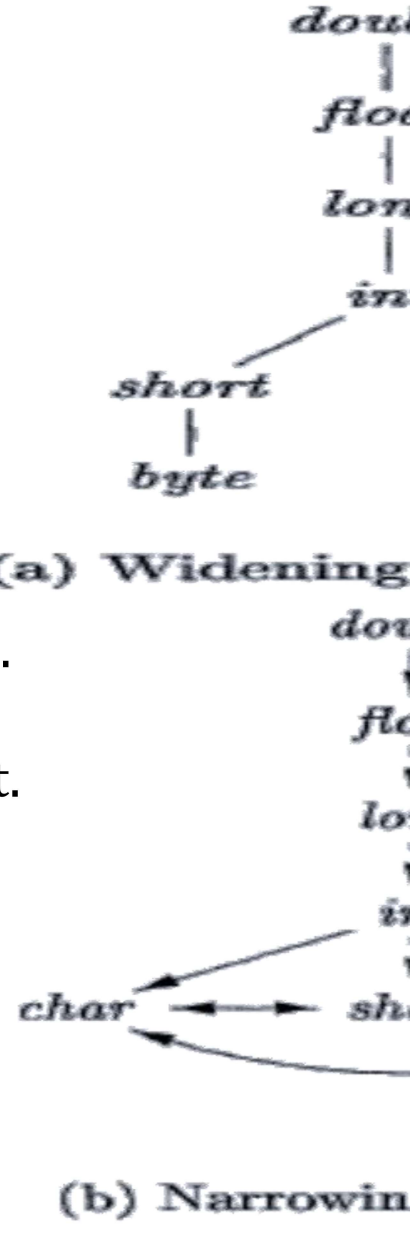
t1 = (float) 2  //explicit

t2= t1 * 3.14

**Two types of conversions:**

**widening conversions**, which are intended to preserve information
any type lower in the hierarchy can be widened to a higher type.

**narrowing conversions**, which can lose information.
a type 's' can be narrowed to a type t if there is a path from s to t.



# TYPE CONVERSION: Types

**Implicit:** Conversion from one type to another is said to be implici
done automatically. Implicit type conversions, also called coercio
limited in many languages to widening conversions.

**Explicit:** Conversion is said to be explicit if the programmer mus
something to cause the conversion. Explicit conversions are also
casts.

Type checking rules for conversion from integer to float.

# Backpatching

- Previous codes for Boolean expressions insert symbolic labels for
- It therefore needs a separate pass to set them to appropriate add
- We can use a technique named backpatching to avoid this
- We assume we save instructions into an array and labels will be ir the array

For nonterminal B we use two attributes B.truelist and B.falselist t with following functions:

makelist(i): create a new list containing only I, an index into the array of instru

Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a poin concatenated list

Backpatch(p,i): inserts i as the target label for each of the instruction o pointed to by p

# Backpatching for Boolean Expressions

$B \rightarrow B_1 \;||\; M\; B_2 \mid B_1 \;\&\&\; M\; B_2 \mid !\, B_1 \mid (\, B_1\,) \mid E_1 \;\mathbf{rel}\; E_2 \mid \mathbf{true} \mid \mathbf{false}$

$M \rightarrow \epsilon$

1) $B \rightarrow B_1 \;||\; M\; B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1 \;\&\&\; M\; B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$;

3) $B \rightarrow\; !\, B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow (\; B_1\;)$ { $B.truelist = B_1.truelist$;
$B.falselist \;=\; B_1.falselist$; }

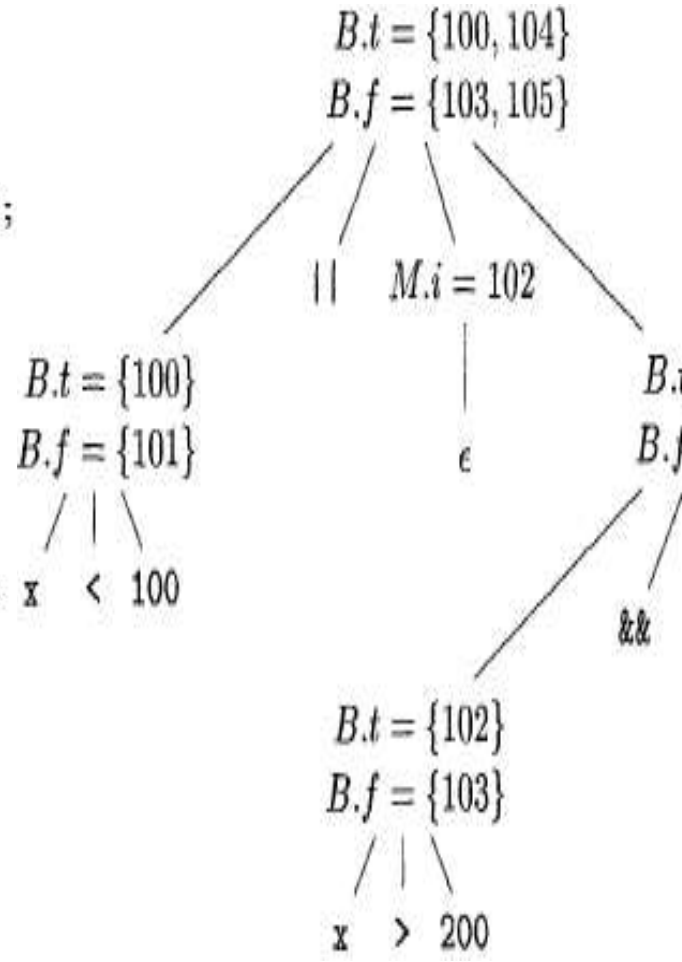
5) $B \rightarrow E_1\; \mathbf{rel}\; E_2$ { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit(\texttt{'if'}\; E_1.addr\; \mathbf{rel}.op\; E_2.addr\; \texttt{'goto \_'})$;
$emit(\texttt{'goto \_'})$; }

6) $B \rightarrow \mathbf{true}$ { $B.truelist = makelist(nextinstr)$;
$emit(\texttt{'goto \_'})$; }

7) $B \rightarrow \mathbf{false}$ { $B.falselist = makelist(nextinstr)$;
$emit(\texttt{'goto \_'})$; }

8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

Annotated parse tree for x < 100 |



# Translation of a switch-statement

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

```
        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-
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

# INTERMEDIATE CODE FOR PROCEDURES

In three-address code, a function call is specified with the below steps

- Evaluation of parameters in preparation for a call (pass by value)
- Function call itself.

Suppose that ‘a’ is an array of integers, and that ‘f’ is a function from integers to integers.

Then, the assignment

n = f(a[i]);

might translate into the following three-address code:

1) t1 = i * 4 2) t2 = a [ t1 ]
3) param t2
4) t3 = call f, 1
5) n = t3

# INTERMEDIATE CODE FOR PROCEDURES

**Example :**
three Address code for Function Definition

```
int f (int x, int y)
{
return x + y + 1;
}
```

Function call : f(2 + 3, 4)

```
1)      t1 = x
2)      t2 = y
3)      param t1
4)      param t2
5)      t3 = call f, 2        // 2 indicates no. of arguments of function
```

# RUN-TIME ENVIRONMENT

- ✓ the compiler creates and manages a run-time environment in which it assumes its target programs are being executed

**This environment deals with:**

- Layout and storage allocations for objects in source program
- Mechanism of accessing variables by target program
- Linkage between procedures
- Parameter passing mechanism
- Interface to OS, I/O and other programs
- Storage allocation and access to variables and data
- Memory management (stack and heap)
- Garbage collection

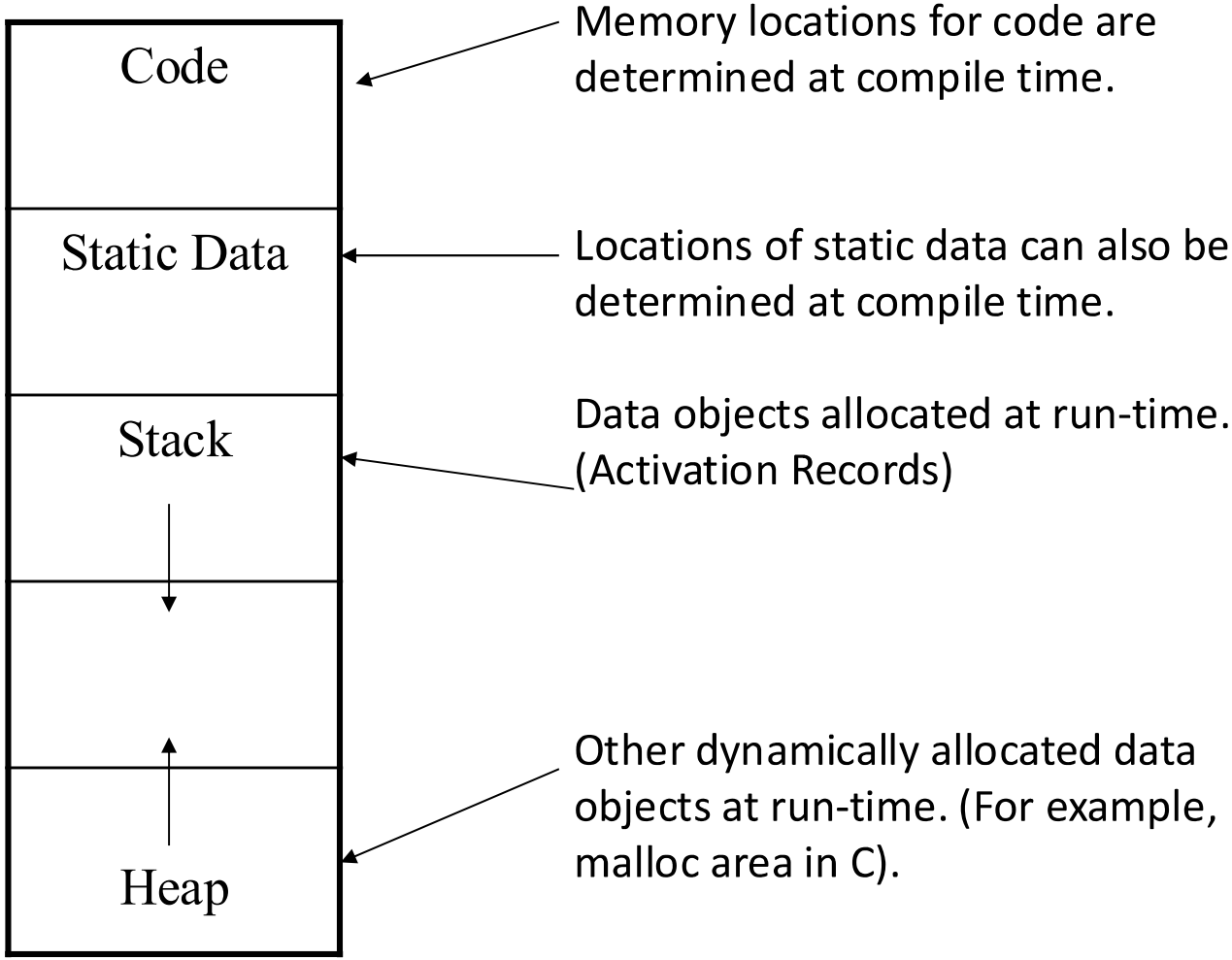
# Memory Organization

Various language features that affect the organization of memory:

1) Does source language support recursion?
   - →Several instances of procedures are in active
   - →Memory allocation is to be done for every instance
2) How are parameters passed to procedures?
   - →Memory allocation for different parameter passing is d
3) Does the procedures refer nonlocal names?
   - →Often procedures have access to its local names.
   - →But access to nonlocal name is required.
4) Does the language support allocation and deallocation dynamica
   - →Effective utilization of memory is possible.

# Storage Organization

Compiler demands a block of memory from Operating system to run compiled program.



# Static Allocation

- size of data objects is known at compile time, names are bound to sto compile time only
- Memory allocated for objects won't be changes in rum time
- In static allocation, it is easy for the compiler to find the address of the ob in activation record.

At compile time, compiler can fill the addresses at which target code can the it operates on.

- Fortran uses static allocation strategy.

**Limitations:**

- Knowing the size of data object at compile time is mandatory
- Data structures cannot be created dynamically (cannot manage r allocation of memory)
- Recursive procedures are not supported

# Stack Allocation

Stack is used for storage allocation. This stack is called as control stack. Al languages have procedures/functions/methods as units of user-defined actions.

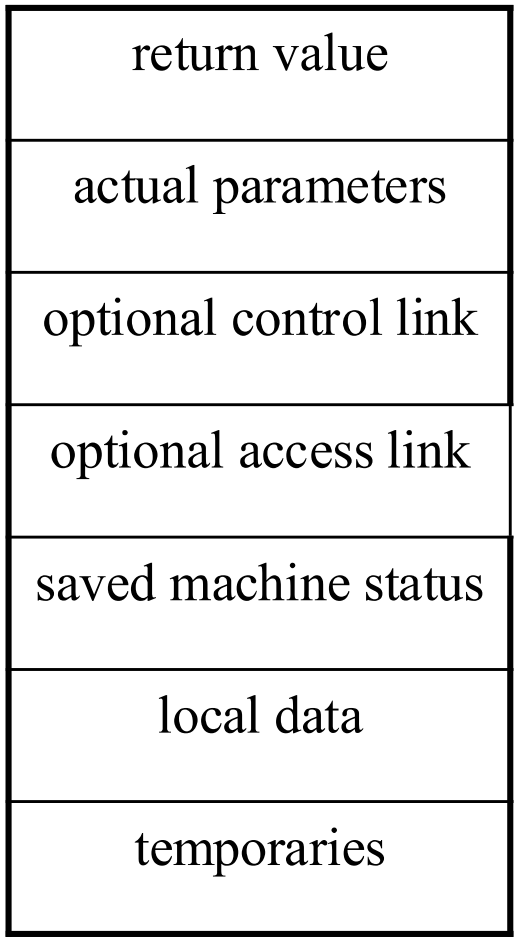
Each time a function is called (activation), space for its local variables is pushed stack, and when the procedure terminates, that space is popped out of the stack.

- So, local variables are stored in activation record and bound to it.
- Data structures can be created dynamically.

**Limitation:**

- Stack allocation would not be feasible if function calls (or activations of function) did not nest in time.
- Memory addressing is slow because of pointers and index registers.

# Stack Allocation of Spaces: Activation Records




The returned value of the called procedure is returr
in this field to the calling procedure. In practice, we
use a machine register for the return value.

The field for actual parameters is used by the calling
procedure to supply parameters to the called proce

The optional control link points to the activation rec
of the caller.

The optional access link is used to refer to nonlocal
held in other activation records.

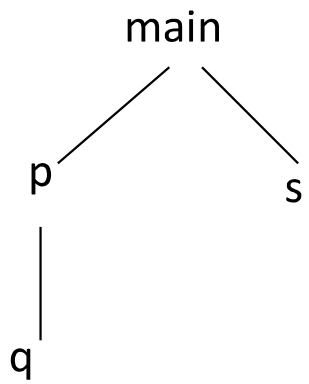
The field for saved machine status holds informatio
the state of the machine before the procedure is ca

The field of local data holds data that local to an exe
of a procedure..

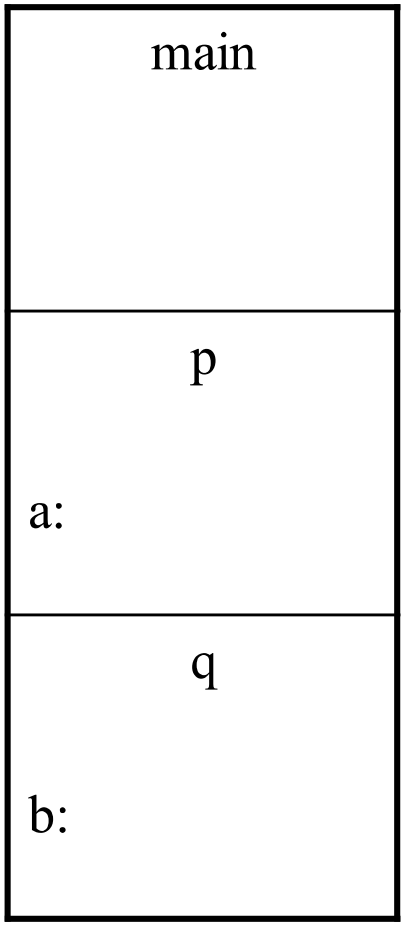
Temporay variables is stored in the field of tempora

# Activation Record : Example

```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```




st



# ACCESS TO NONLOCAL DATA ON THE STACK

Storage allocation is done for two types of data:

1. **Local data**
2. **Non-Local data:** If a procedure refers to variables that are not local to it

There are languages that don't support nesting of procedures, then non-locals are n
global variables (C language).

In such languages, global variables are placed in static storage, storage and location
compile time.

- Static Scope is used to access non-local names

There are languages that support nesting of procedures.
In these languages non-local data are handled by using scope information.

There are two types of scope rules,

- Static scope (used by block structured languages)
- Dynamic scope (used by non-block structured languages)

# Access to Nonlocal Names

Scope rules of a language determine the treatment of referen
nonlocal names.

**Lexical Scope (Static Scope)**

Determines the declaration that applies to a name by exa
the program text alone at compile-time.

Most-closely nested rule is used.

Pascal, C, ..

**Dynamic Scope**

Determines the declaration that applies to a name at run-tir

Lisp, APL, ...

# Lexical/ Static Scope

The scope of a declaration in a block-structured language is given *mostly closed rule*.

Each procedure (block) will have its own activation record.

- procedure
- begin-end blocks
  - (treated same as procedure without creating most part of its activation record)

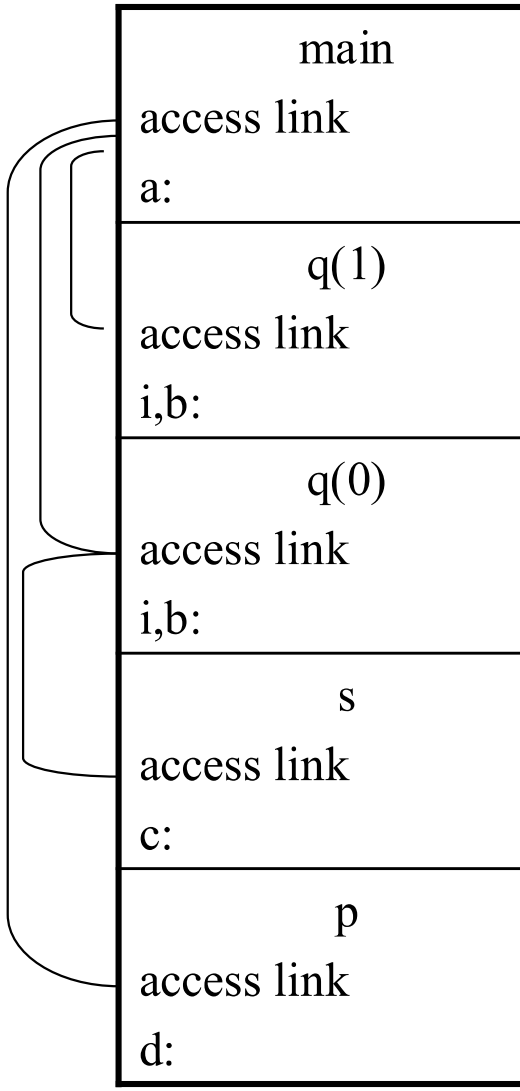
A procedure may access to a nonlocal name using:

- access links in activation records, or
- displays (an efficient way to access to nonlocal names)

# Access Links

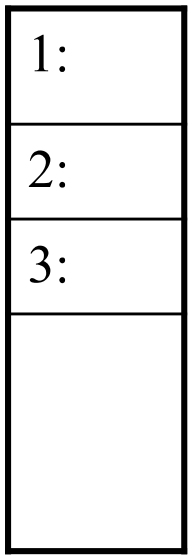
```
program main;
  var a:int;
  procedure p;
    var d:int;
    begin a:=1; end;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      begin p; end;
    begin
      if (i<>0) then q(i-1)
      else s;
    end;
  begin q(1); end;
```

Access Links



# Displays

- An array of pointers to activation records can be used to access activation records.
- This array is called as displays.
- For each level, there will be an array entry.




Current activation record at level 1

Current activation record at level 2

Current activation record at level 3

# Dynamic Scope

It used in non-block structured languages (LISP)

- By considering the current activation, it determines the scope of declarat names at runtime
- In this type, the scope is verified at runtime

**Deep Access:** The idea is to keep a stack of active variables. Use control lin access links and to find a variable, search the stack from top to bottom

- This method of accessing nonlocal variables is called deep access. Since se "deep" in the stack, hence the method is called deep access. In this meth table should be used at runtime

**Shallow Access:** The idea to keep a central storage and allot one slot for e name

- If the names are not created at runtime then the storage layout can be fixe time. Otherwise, when new activation procedure occurs, then that procedure storage entries for its local at entry and exit (i.e., while entering in the pr while leaving the procedure)

# Parameter Passing

If one procedure calls another, communication between them is ma
non-local names and parameters

**L-value** refers to the storage

**R-value:** refers to value contained in the storage

## Parameter Passing Techniques:

1) **call by value:** R-values are passed to formals
2) **call by reference:** L-values are passed to formals
3) **copy restore:**

- hybrid (call by value and call by reference)
- Also called as copy-in copy-out
- R-values are passed to formals, before the call L-values are determined
- When control returns to caller, R-value of formal are copied into l-value of actual

**4) call by name**

- Actual parameters are substituted for the formals
- Macro expansion/inline expansion

# Heap Management

- Heap is used to allocate space for dynamic objects
- May be managed by the user or by the runtime syste

In a perfect world:

| | live | dead |
|---|---|---|
| not deleted | √ | ---- |
| deleted | ---- | √ |

# User Heap Management

User library manages memory; programmer decides when and where to **allocate** and **deallocate**

- void* malloc(longn)
- void free(void*addr)

Library calls OS for more pages when necessary

How does malloc/free work?

- Blocks of unused memory stored on a freelist
- malloc: search free list for usable memory block
- free: put block onto the head of the freelist

# Heap Management

**Drawbacks**

malloc is not free: we might have to do a search to find a big enough block

As program runs, the heap fragments, leaving many small, unusable pieces

Have to have a way to reclaim and merge blocks when freed.

Memory management always has a cost. We want to minimize costs and, these days, maximize reliability

# Heap Management

Memory management would be simpler if

(a) all allocation requests were for chunks of the same size, and
(b) storage was released predictably, say, first-allocated deallocated.

Desirable properties of memory manager are

1. Space Efficiency,
2. Program Efficiency,
3. Low overhead

# Garbage Collection

- Data that cannot be referenced is generally known as garbage. Automatic garbage collection is used in many high-level languages.
- A user program, which we shall refer to as the mutator, modifies the collection of objects in the heap.
- The mutator creates objects by acquiring space from the memory manager.
- Objects become garbage when the mutator program cannot reach them.
- The garbage collector finds these unreachable objects and reclaims their space by handing them to the memory manager

# Design Goals for automatic Garbage

**CollectorsType Safety**

- A language in which the type of any data can be determined is said to be ty
- Statically typed languages/ Statically type safe (ML at compile time)
- Dynamically typed languages/ Dynamically type safe (JAVA at runtime)
- Neither statically nor dynamically type safe, then it is said to be unsafe. (C &

**Performance Metrics**

- Garbage collection is often so expensive

Metrics:

- Overall Execution Time,
- Space Usage,
- Pause Time,
- Program Locality.

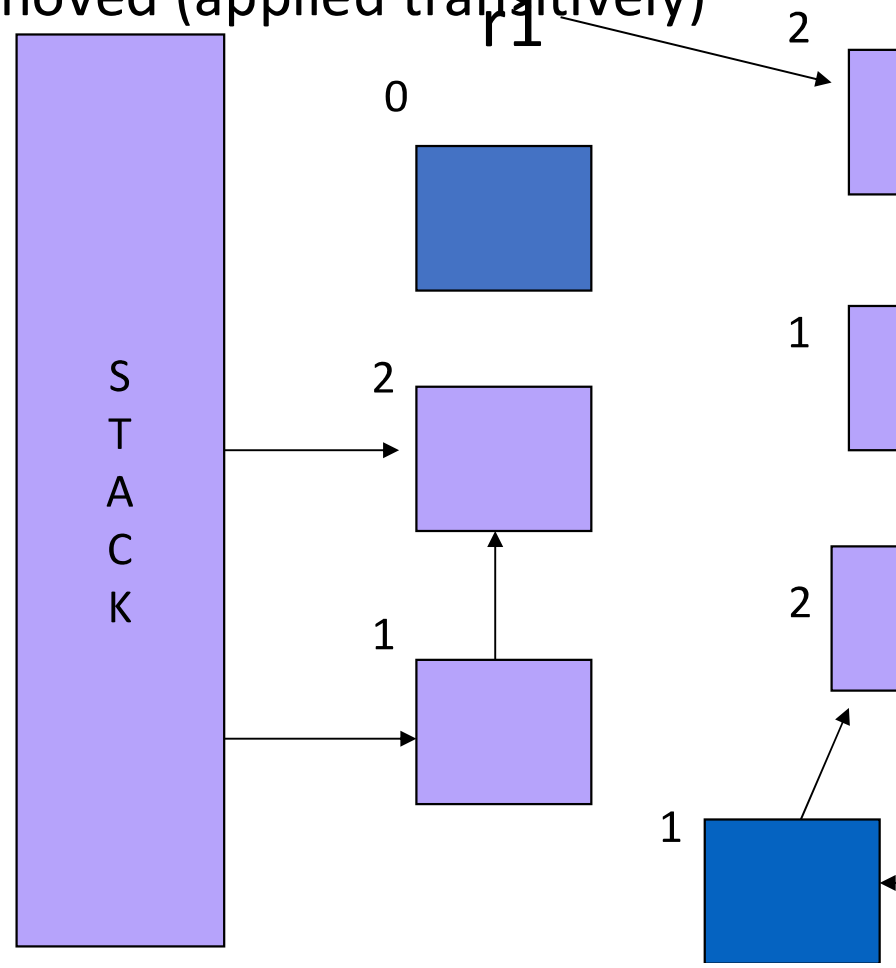
# Garbage Collectors: Reference Counting

Keep a count of pointers to each object

- performance overhead each time the reachable set can change
- storage overhead since each object needs a count field

Zero references → garbage that can be removed (applied transitively)

> zero references → not garbage??



# Garbage Collectors: Tree Based

**Trace-based collectors:**

- Incremental Garbage Collection
- Incremental Reachability Analysis
- Partial-Collection Basics
- Generational Garbage Collection