CSL302: Compiler Design

Intermediate Code Generation

Vishwesh Jatala

Assistant Professor

Department of CSE

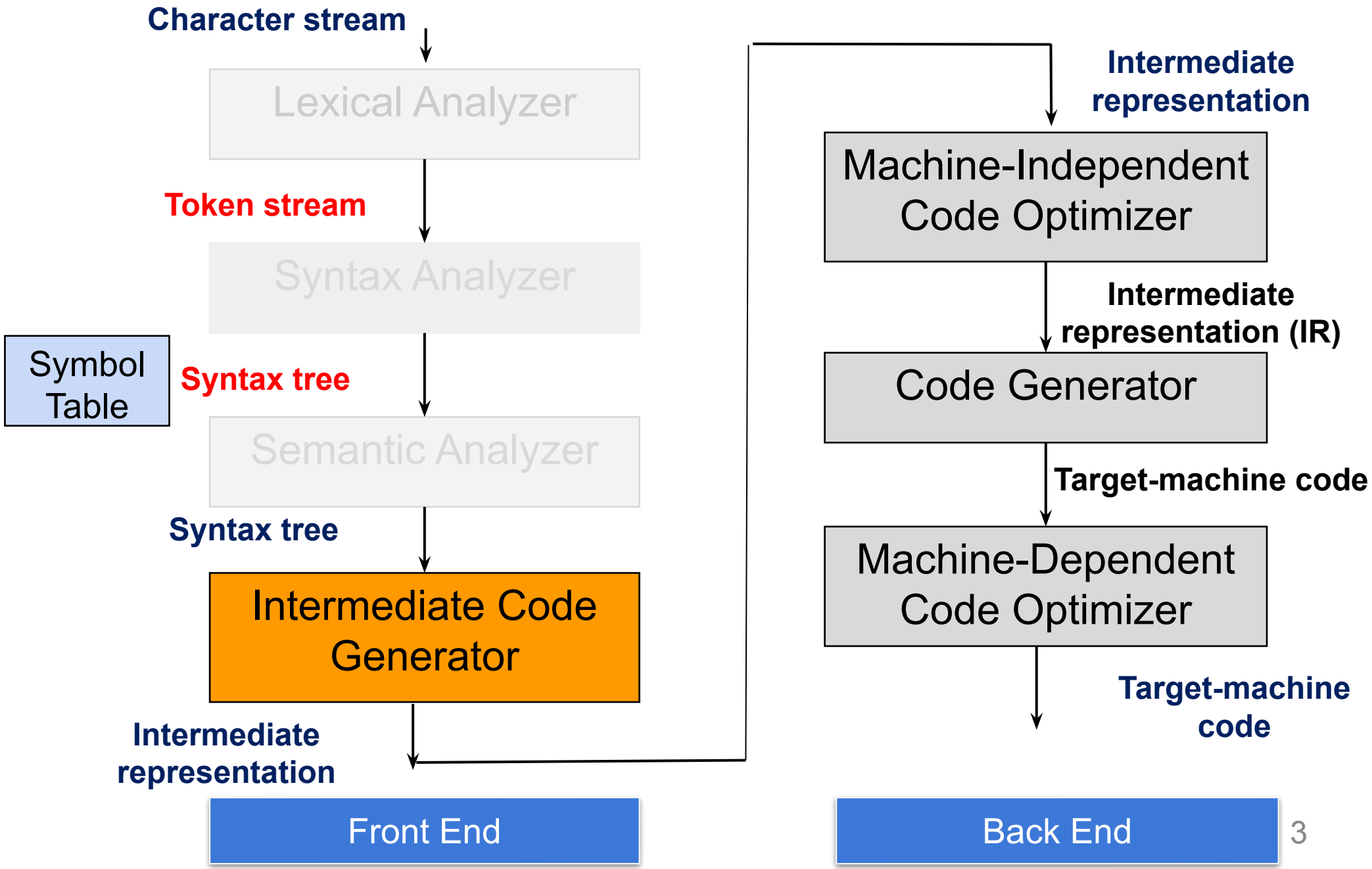
Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

- References for today's slides
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*
  - *IIT Madras (Prof. Rupesh Nasre)*
    http://www.cse.iitm.ac.in/~rupesh/teaching/compiler/aug15
  - *Course textbook*

Compiler Design



Outline

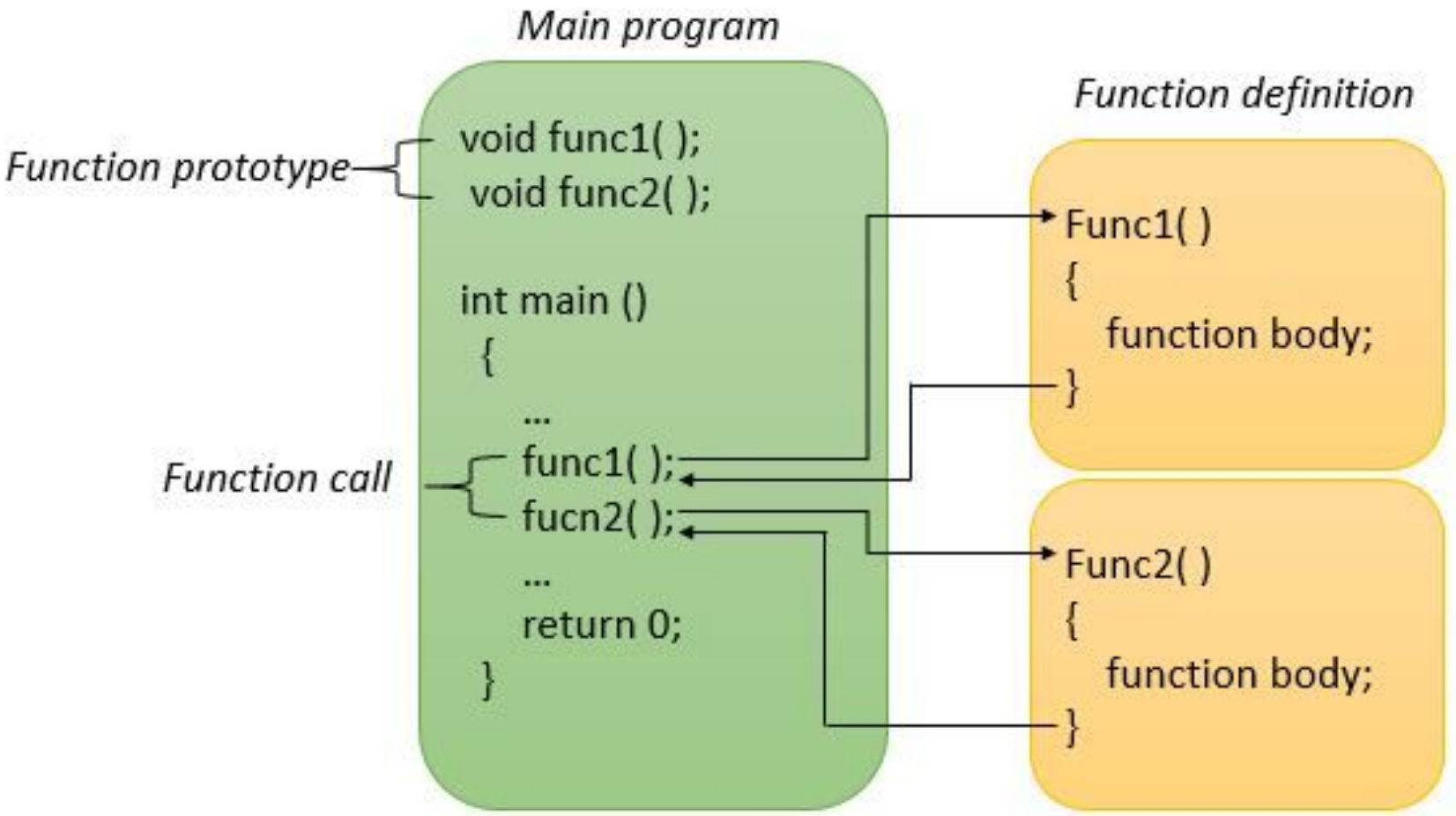
- Intermediate code generation
  - Procedure/Functions
    - Three address code
    - Runtime environment

Procedures

Procedure

- A procedure/function/method/subroutine definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point
- Formal parameters are the one that appear in declaration. Actual parameters are the one that appear in when a procedure is called

Procedure Invocation



Three address code

- Assignment
  - x = y op z
  - x = op y
  - x = y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = &y
  - x = *y
  - *x = y

3AC for Procedure Calls

Procedure Call: p(x1, x2,.....xn)

TAC:

param x1

param x2

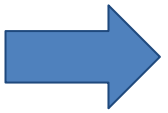
..

param xn

call p, n

Procedure Calls

```
n = f(x+4);
```



```
t1 = x+4
param t1
t2 = call f, 1
n = t2
```

3AC for Procedure Calls

S → call id (Elist)

Elist → Elist , E

Elist → E

Procedure Calls

- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list
- S → call id (Elist)
  for each item p on queue do emit('param' p)
  emit('call' id.place)

  Elist → Elist , E
  append E.place to the end of queue

  Elist → E
  initialize queue to contain E.place

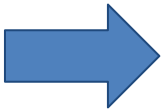
Procedure Calls

```
n = f(x+4, y*4);
```

Generate the three address code?

Procedure Calls

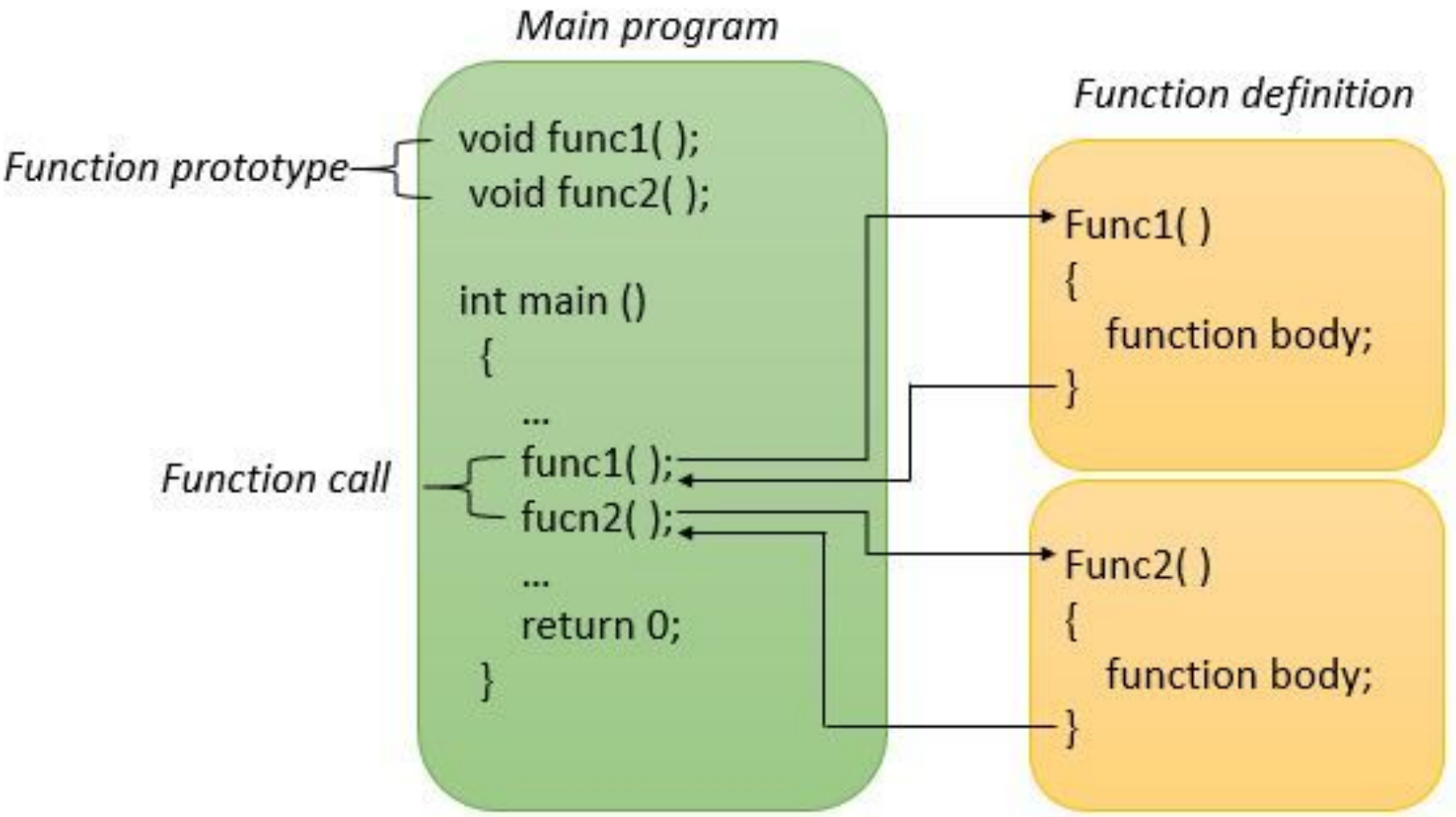
```
n = f(x+4, y*4);
```



```
t1 = x+4
t2 = y*4
param t1
param t2
t3 = call f, 2
n = t3
```

What goes inside the call?

Procedure Invocation



Activation tree

- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node **a** is parent of **b** if control flows from **a** to **b**
  - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

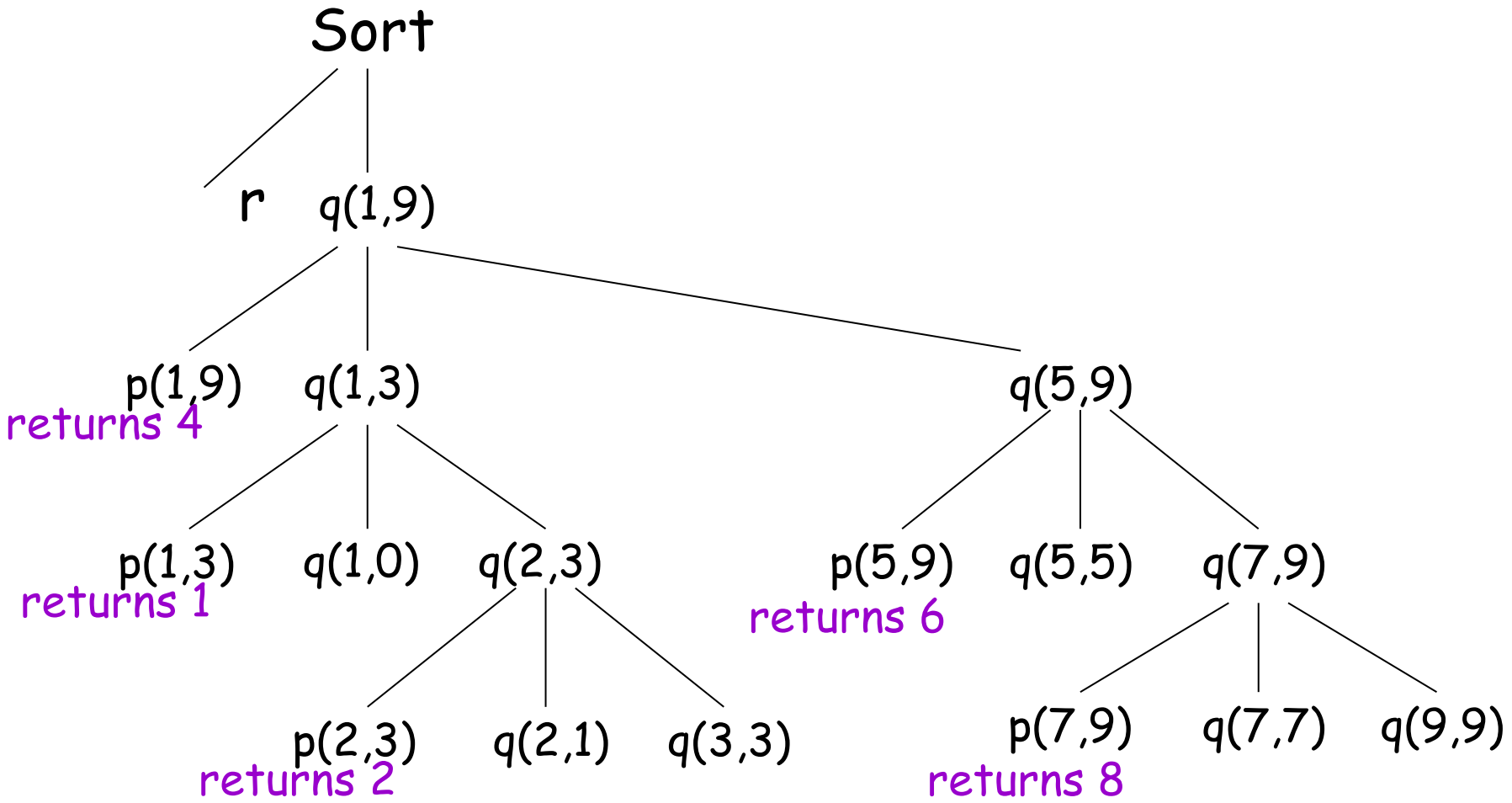
Example

```
program sort;
 var a : array[0..10] of
   integer;

   procedure readarray;
     var i :integer;
             :
   function partition (y, z
             :integer)
   :integer;
     var i, j ,x, v :integer;
         :
```

```
   procedure quicksort (m, n
                        :integer);
     var i :integer;
             :
     i:= partition (m,n);
     quicksort (m,i-1);
     quicksort(i+1, n);
         :
begin{main}
   readarray;
   quicksort(1,9)
end.
```

Activation Tree



Control stack

An activation is a single execution of a procedure (function)

- Flow of control in program corresponds to depth first traversal of activation tree
- How to keep track of activations?
  - Stack
- Push the node when activation begins and pop the node when activation ends
- When the node n is at the top of the stack the stack contains the nodes along the path from n to the root

OR

Control stack

- Maintain a stack
- What do we maintain at the procedure activation in stack?

Control stack

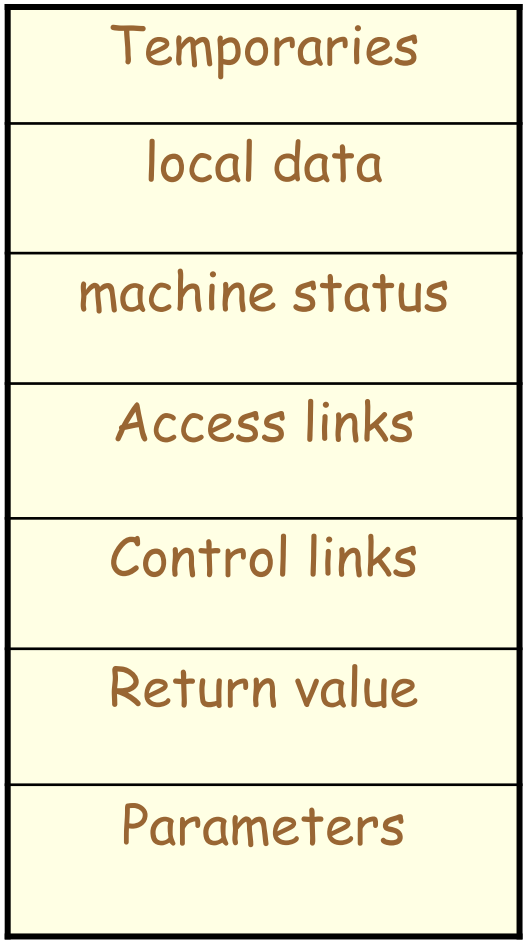
- What do we maintain at the procedure activation in stack?
  -

Control stack

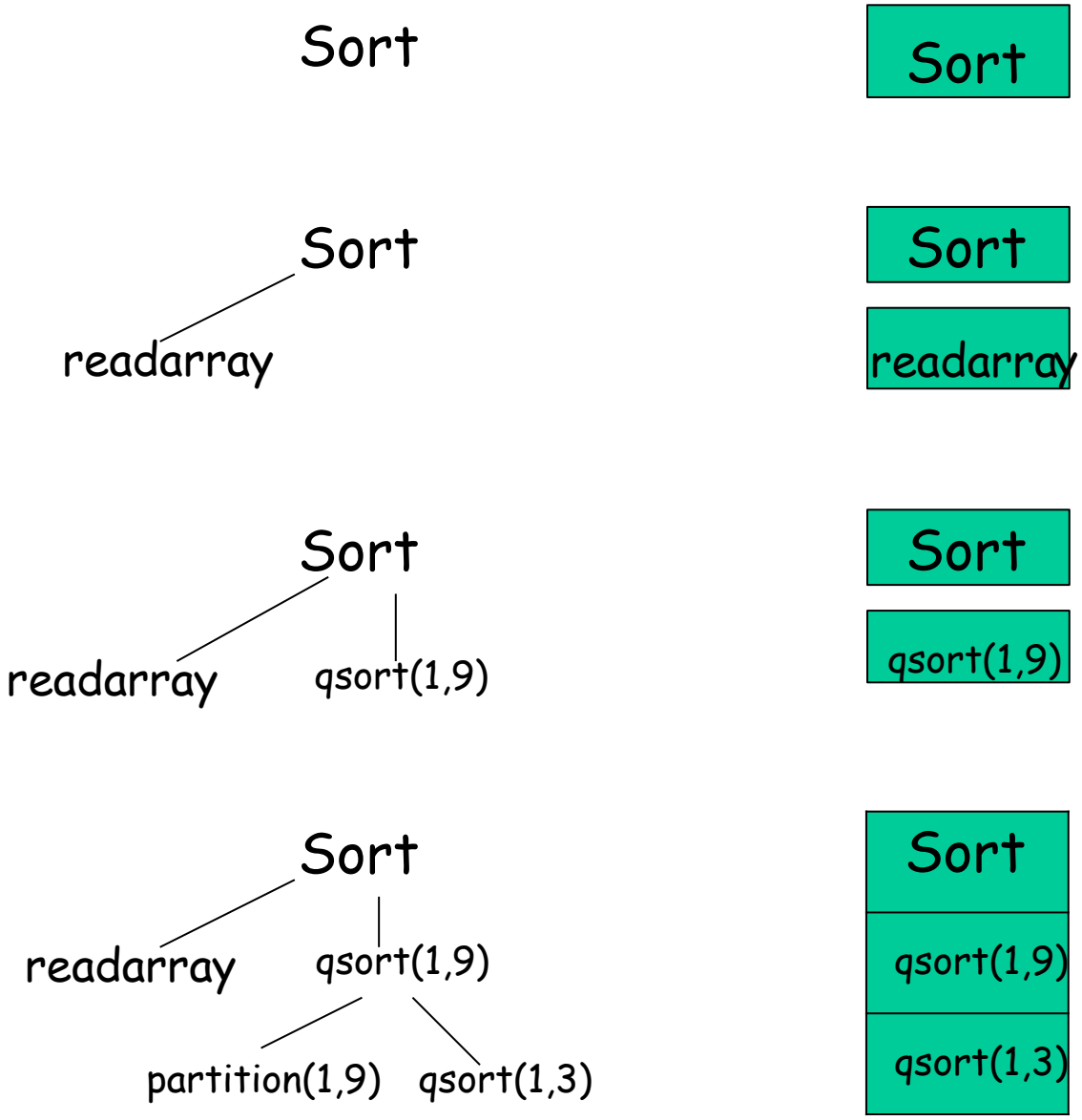
- What do we maintain at the procedure activation in stack?
  - Parameters
  - Local variables
  - Temporary variables
  - Return address
  - Return value..
  - ...

Activation Record

- **temporaries:** used in expression evaluation
- **local data:** field for local data
- **saved machine status:** holds info about machine status before procedure call (return address)
- **access link :** to access non local data
- **control link :**points to activation record of caller
- **actual parameters:** field to hold actual parameters
- **returned value**: field for holding value to be returned



Stack Allocation



Call Sequence

chalu hone se pehle cSP ko anya BP banao , and then we can continue

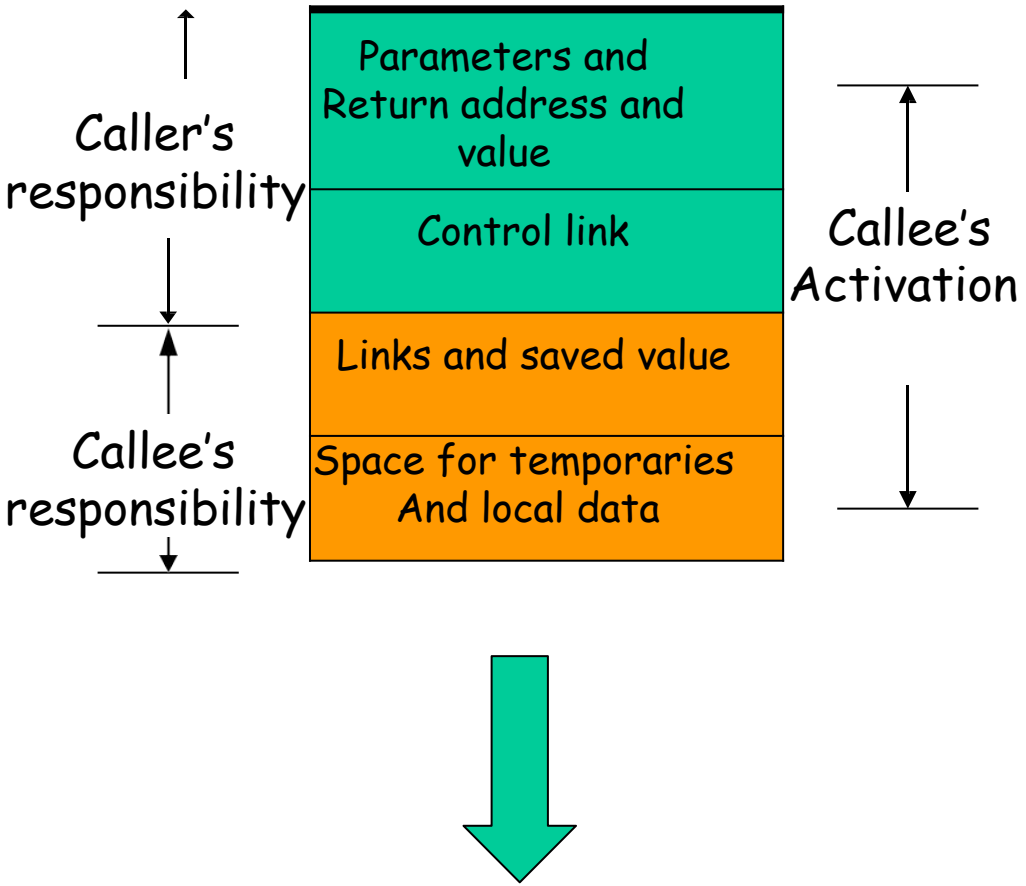
- Caller evaluates the actual parameters
- Caller stores return address and other values (control link) into callee's activation record
- Callee initializes its local data and begins execution

Return Sequence

- Callee places a return value next to activation record of caller
- Branch to return address
- Caller copies return value into its own activation record

Calling Sequence

- A call sequence allocates an activation record and enters information into its field
- A return sequence restores the state of the machine so that calling procedure can continue execution



Access to non-local names

```
Program sort;
 var a: array[1..n] of integer;
        x: integer;
 procedure readarray;
    var i: integer;
    begin
         x = x+ i
    end;
 procedure exchange(i,j:integer)
    begin


    end;
```

Access to non-local names

```
   {
S1:   int x
S2:   X=20
      {
S3:      int X
S4:      X=15
S5:      printf("%d", x)
      }
      {
S6:         printf("%d", x)
      }
S7:   printf("%d", x)
   }
```

Output of S_5*?*

Output of S_6*?*

Output of S_7*?*

Block

- Statement containing its own data declarations
- Blocks can be nested
  - also referred to as *block structured*
- Scope of the declaration is given by *most closely nested* rule
  - The scope of a declaration in block B includes B
  - If X is not declared in B then an occurrence of X in B is in the scope of declaration of X in B′ such that
    - B′ has a declaration of X
    - B′ is most closely nested around B

Example

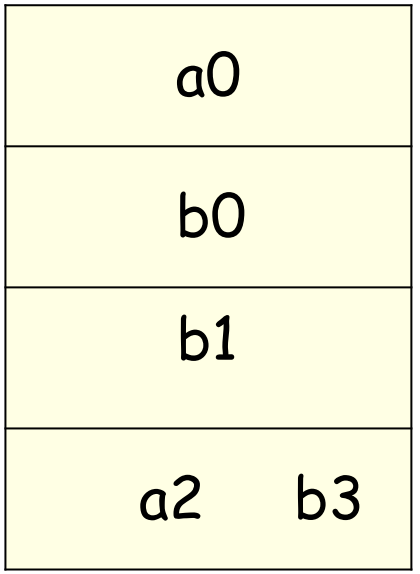
```
main()
{                                   BEGINNING of B0
    int a=0                                                    Scope B0, B1, B3
    int b=0                                                    Scope B0
    {                               BEGINNING of B1
        int b=1                                                Scope B1, B2
        {                           BEGINNING of B2
            int a=2                                            Scope B2
            print a, b
        }                           END of B2

        {                           BEGINNING of B3
            int b=3                                            Scope B3
            print a, b
        }                           END of B3
        print a, b
    }                               END of B1
    print a, b
}                                   END of B0
```

Blocks …

- Blocks are simpler to handle than procedures
- Blocks can be treated as parameter less procedures
- Either use stack for memory allocation
- OR allocate space for complete procedure body at one time

```
{ // a0
    { // b0
        { // b1
            { // a2
            }
            { //b3
            }
        }
    }
}
```



Access to non-local names

```
{  var X: integer
    X=50
    procedure Bar
   begin
       print X ---> S
    end
    procedure Foo
   begin
       var X: integer
            X=10
            Bar()
       end
    end
    Foo ()
end
```

Output of X at S?