Type Checking and Run Time Environments

1. Type Checking:

Type systems, Specification of a simple type checker, Equivalence of type expressions, Type conversions

2. Run Time Environments:

Parameter passing by value, reference, and name; activation records, stack and static allocation of activation records; translating a function call, allocating offsets to variables, generating code for function prologue, function epilogue, call sequence, and return sequence.

TYPE CHECKING

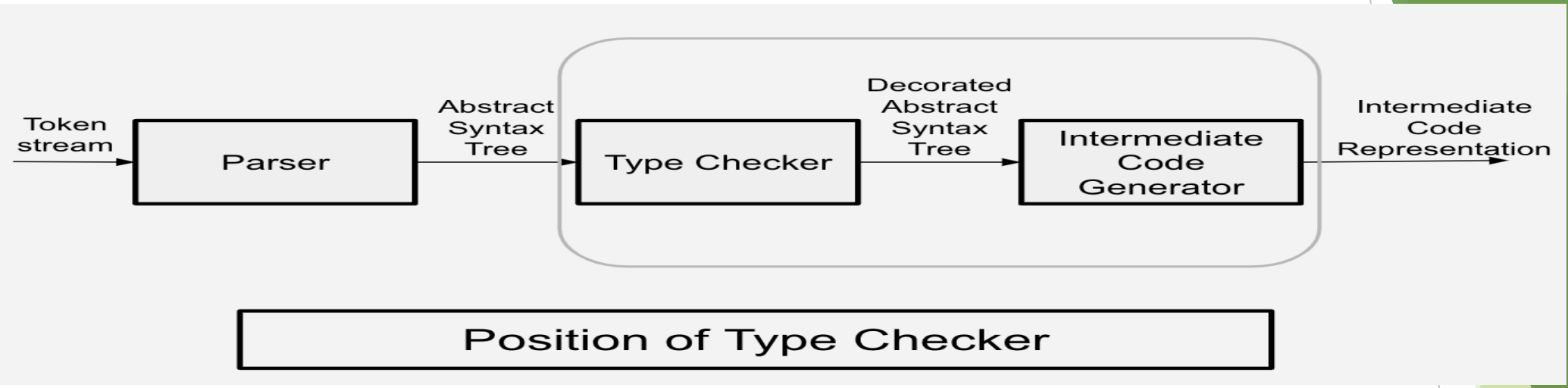
A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Static checks can be classified as:

1. Type checks - A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. Flow-of-control checks - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switch statement.

Position of type checker



A type checker verifies that the type of a construct matches that expected by its context.

For example : arithmetic operator ***"mod"*** in Pascal requires integer operands, so a type checker verifies that the operands of ***mod*** have type integer.

Type information gathered by a type checker may be needed at the time of code generation.

Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : “ if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer ”

- The type system is a set of rules for assigning type expressions to various program parts.
- A type system is implemented with the help of a type checker.
- It is syntax-directed.
- Compilers and processors for the same language may utilize different type systems.

Type Expressions

- The type of a language construct will be denoted by a "**type expression**."
- A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.
- Static type checking is performed by a compiler, whereas terminal dynamic type checking is performed when the target program is run.
- Type checking should have the property for error recovery.

The definitions of type expressions are as follows:

1.Type expressions are basic types such as boolean, char, integer, and real.

During type checking, a basic type “type_error” will signify an error; void, which denotes "the lack of a value," allows statements to be checked.

2.A type name is a type expression since type expressions can be named.

3.A type expression is a type constructor applied to the type expressions.

4.Variables whose values are type expressions can be found in type expressions.

EXAMPLE

Example for a type constructor applied to type expressions is a type expression.

Constructors include:

Arrays :

If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products :

If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

Records :

The difference between a record and a product is that the names.

The record type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
address: integer;
    lexeme: array[1..15] of char
    end;
var table: array[1...101] of row;
```

declares the type name ***row*** representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable table to be an array of records of this type.

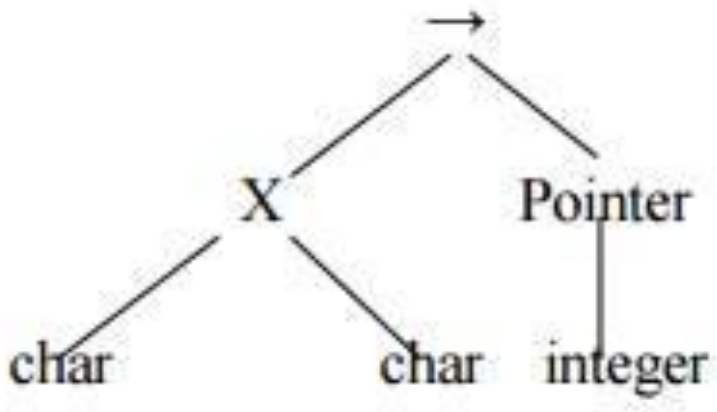
Constructors include (continue):

Pointers :
If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T".

For example, var p: ↑ row declares variable p to have type pointer(row).

Functions :
A function in programming languages maps a domain type D to a range type R.
The type of such function is denoted by the type expression D → R

Example for a type expressions may contain variables whose values are type expressions.



Tree representation for char x char → pointer (integer)

Types of Type Checking

There are two types of type checking:

- Static Type Checking
- Dynamic Type Checking

Static Type Checking

1. Static type checking in compiler design refers to the process of analyzing the source code during compilation to ensure that variables and expressions adhere to a consistent and predetermined set of data types.
2. The compiler checks for type-related errors and violations before generating the executable code.
3. This early detection at compile-time helps prevent many common type-related issues, improving program reliability and performance.
4. Languages like C, Java and C++ employ static type checking.

Dynamic Type Checking

1. Dynamic type checking in compiler design involves checking the data types of variables and expressions during program execution.
2. Instead of enforcing type constraints at compile-time, the compiler generates code that includes runtime checks for type compatibility.
3. This approach provides more flexibility, allowing variables to change types dynamically during program execution.
4. However, it may lead to runtime errors if unexpected type mismatches occur.
5. Languages like Python, PHP and JavaScript use dynamic type checking.

SPECIFICATION OF A SIMPLE TYPE CHECKER

- A type checker for a simple language checks the type of each identifier.
- The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions.
- The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E
D → D ; D
D → id : T { addtype (id.entry , T.type) }
T → char { T.type : = char }
T → integer { T.type : = integer }
T → ↑ T1 { T.type : = pointer(T1.type) }
T → array [num] of T1 { T.type : = array (1… num.val , T1.type) }

In the above language,

- There are two basic types : char and integer ;
- type_error is used to signal errors;
- the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression pointer (integer).

Type checking of expressions

E → literal | num | id | E mod E | E [E] | E ↑

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1. E → literal { E.type : = char } E→num { E.type : = integer }
Here, constants represented by the tokens literal and num have type char and integer.

2. E → id { E.type : = lookup (id.entry) }
lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 mod E2 { E.type : = if E1. type = integer and E2. type = integer then integer else type_error }
The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type_error.

4. E → E1 [E2] { E.type : = if E2.type = integer and E1.type = array(s,t) then t
else type_error }

In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

5. E → E1 ↑ { E.type : = if E1.type = pointer (t) then t
else type_error }

The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement: S→id: = E

S→id: = E	{ S.type : = if id.type = E.type then void else type_error }

2. Conditional statement: S→if E then S1

S→if E then S1	{ S.type : = if E.type = boolean then S1.type else type_error }

3. While statement: S → while E do S1

S → while E do S1 { S.type : = if E.type = boolean then S1.type
else type_error }

4. Sequence of statements:

S → S1 ; S2 { S.type : = if S1.type = void and S1.type = void then void else type_error }

S → S1 ; S2 { S.type : = if S1.type = void and
S1.type = void then void
else type_error }

RUN-TIME ENVIRONMENTS - SOURCE LANGUAGE ISSUES

Procedures:

A procedure definition is a declaration that associates an identifier with a statement.

The identifier is the procedure name, and the statement is the procedure body.

For example, the following is the definition of procedure named ***readarray*** :

```
procedure readarray;
        var i : integer;

    begin
        for i : = 1 to 9 do read(a[i])
    end;
```

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

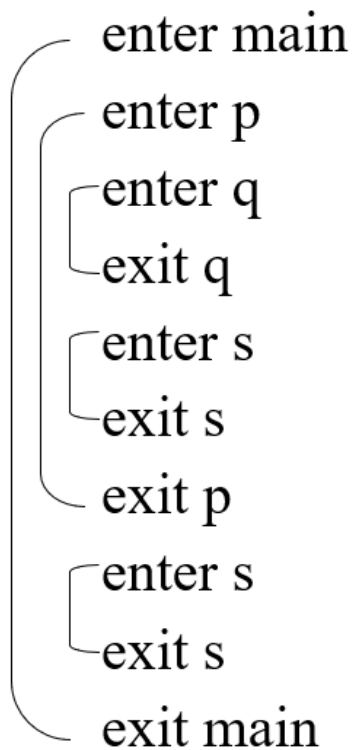
Activation trees

An activation tree is used to depict the way control enters and leaves activations.

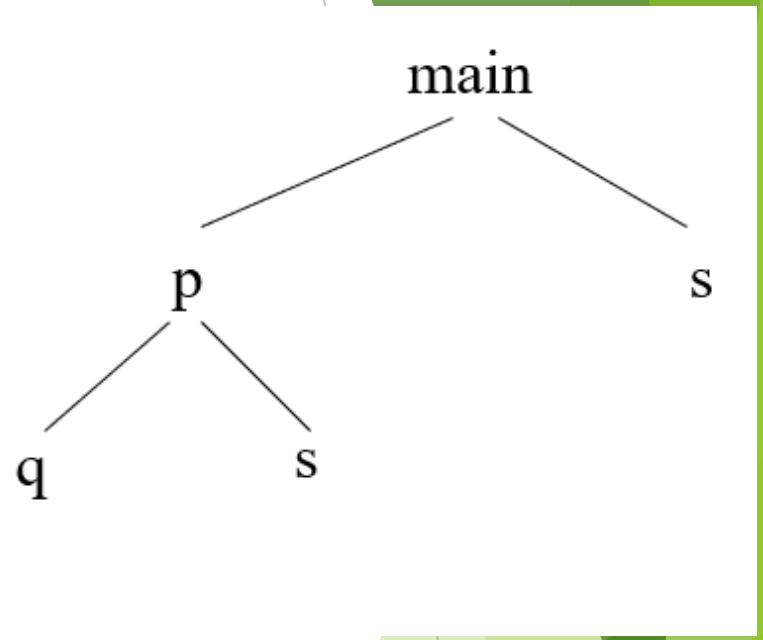
In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for "***a***" is the parent of the node for ***"b"*** if and only if control flows from activation "***a***" to ***"b"***.
4. The node for "***a***" is to the left of the node for "***b***" if and only if the lifetime of "***a***" occurs before the lifetime of "***b***".

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

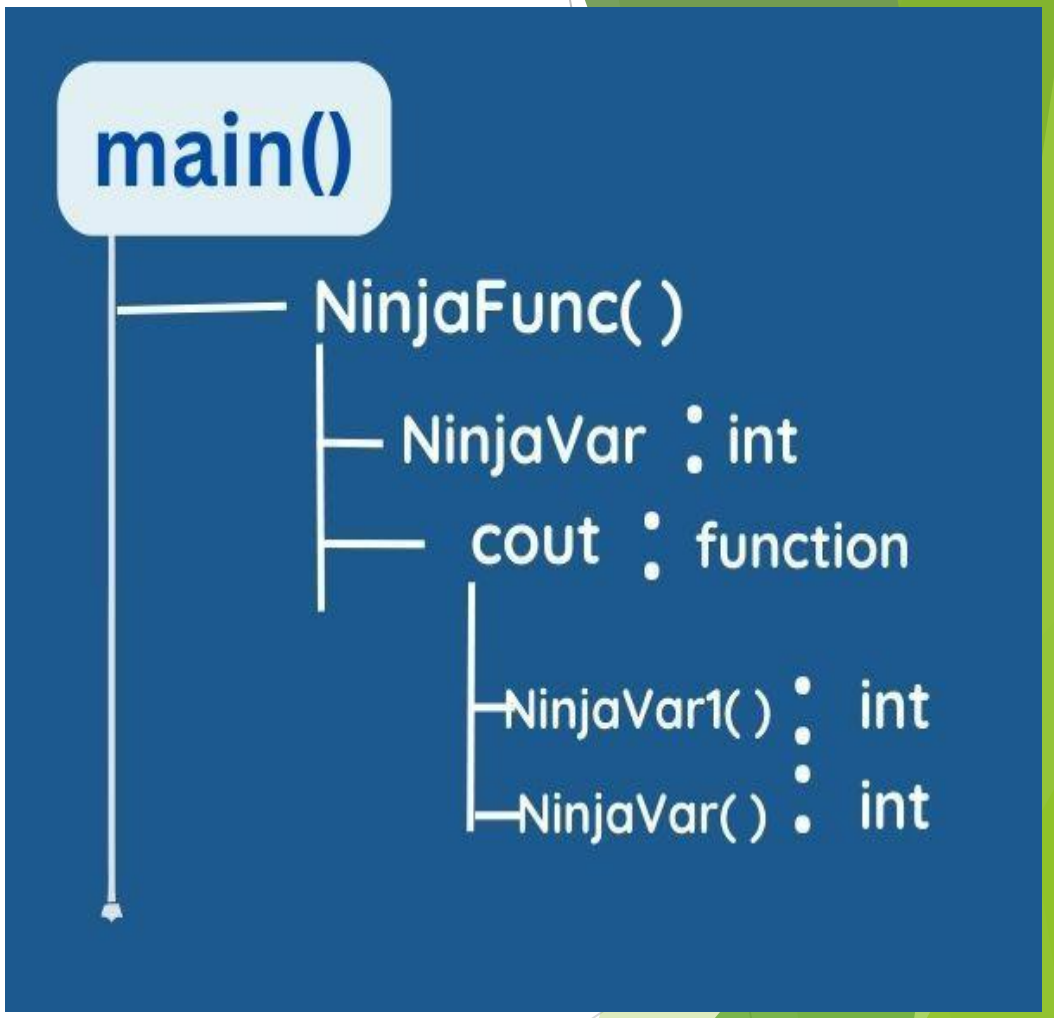


A Nested Structure

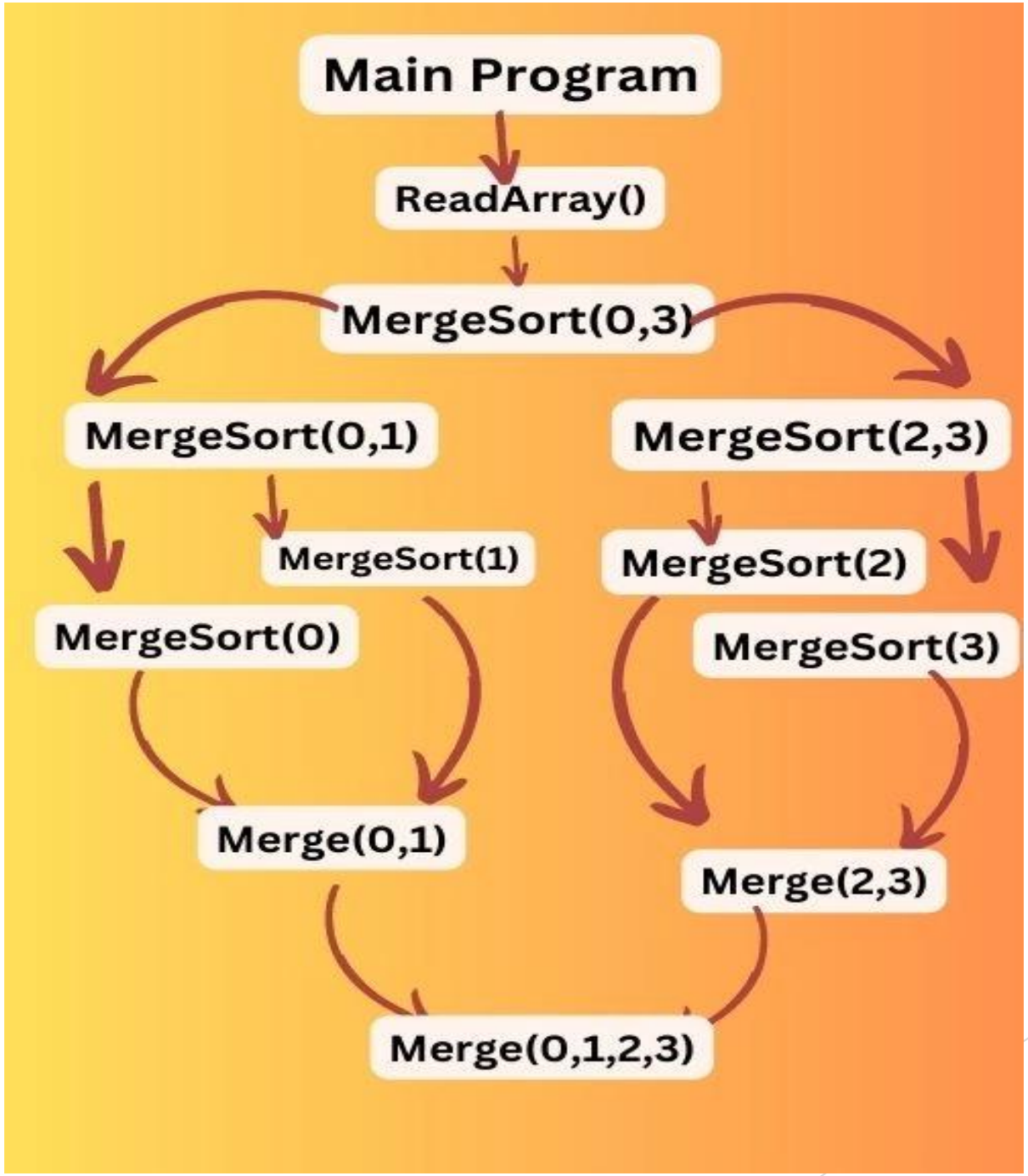


Code

```
#include <iostream>
using  namespace std;

int NinjaVar1=5;
void NinjaFunc(){
   int NinjaVar=20;
   cout<<"NinjaFunc() called in main():"<<NinjaVar1+NinjaVar;
}

int main() {

   NinjaFunc();

   return 0;
}
```



```
enter main(){
    enter  ReadArray()
    leave  ReadArray()
    enter  MergeSort(arr,0,3){

        enter  MergeSort(0,1){

            enter  MergeSort(0,0)
            leave  MergeSort(0,0)

            enter  MergeSort(1,1)
            leave  MergeSort(1,1)

            enter  Merge(0,1)
            leave  Merge(0,1)
        }

        leave  MergeSort(0,1)

        enter  MergeSort(2,3){
            enter  MergeSort(2,2)
            leave  MergeSort(2,2)

            enter  MergeSort(3,3)
            leave  MergeSort(3,3)

            enter  Merge(2,3)
            leave  Merge(2,3)
        }

        leave  MergeSort(2,3)

        enter  Merge(0,1,2,3)
        leave  Merge(0,1,2,3)
    }
    leave  MergeSort(0,3)
}
leave main()
```


Main Program
ReadArray()
MergeSort(0,3)
MergeSort(0,1)
MergeSort(2,3)
MergeSort(1)
MergeSort(2)
MergeSort(0)
MergeSort(3)
Merge(0,1)
Merge(2,3)
Merge(0,1,2,3)

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.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a ‘n’
Declarations may be explicit, such as: var i : integer ;

They may be implicit.
Example, any variable name starting with I is assumed to denote an integer.
The portion of the program to which a declaration applies is called the scope of that declaration.