CC Lecture 18

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

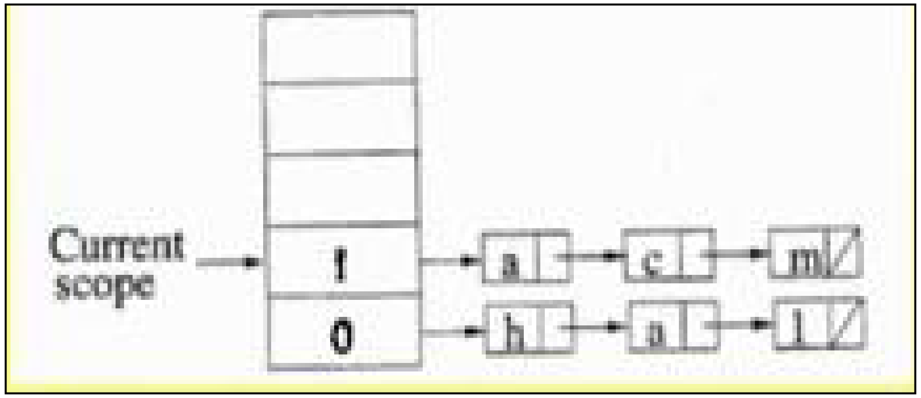
Nested Lexical Scoping

- So, hierarchically from one nesting level to the previous nesting level till it comes to the outermost level, the definition of a variable is checked/searched.
- **Visibility rules** are used to resolve conflicts arising due to the same variable being defined more than once.
- In this case, the innermost declaration closest to the reference is used.
- To implement the symbol tables with nested scope:
  1. **One table for each scope**
  2. **A single global table**

One Table per scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Here, **Lists, Trees, Hash tables** can be used.

Scoped Symbol Table: List



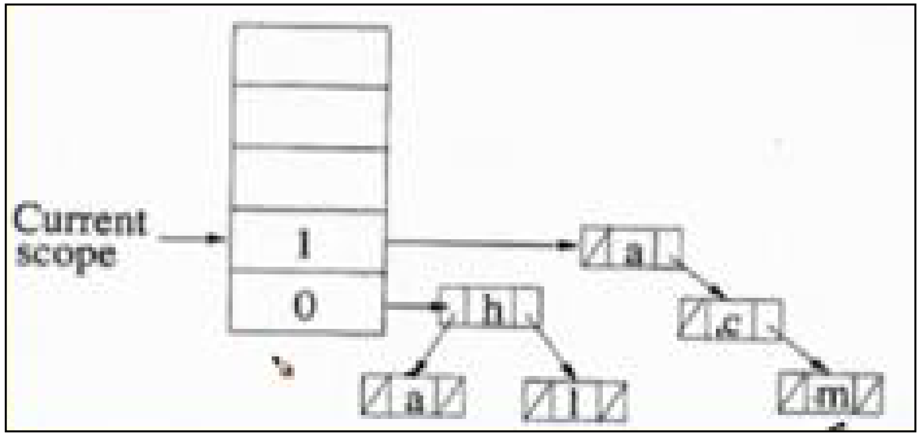
Current scope pointer points to current scope

- Nesting levels

```
{
    int h, a, l;
    {
        int a, c, m;
        …
    }
    …
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

Scoped Symbol Table: Tree

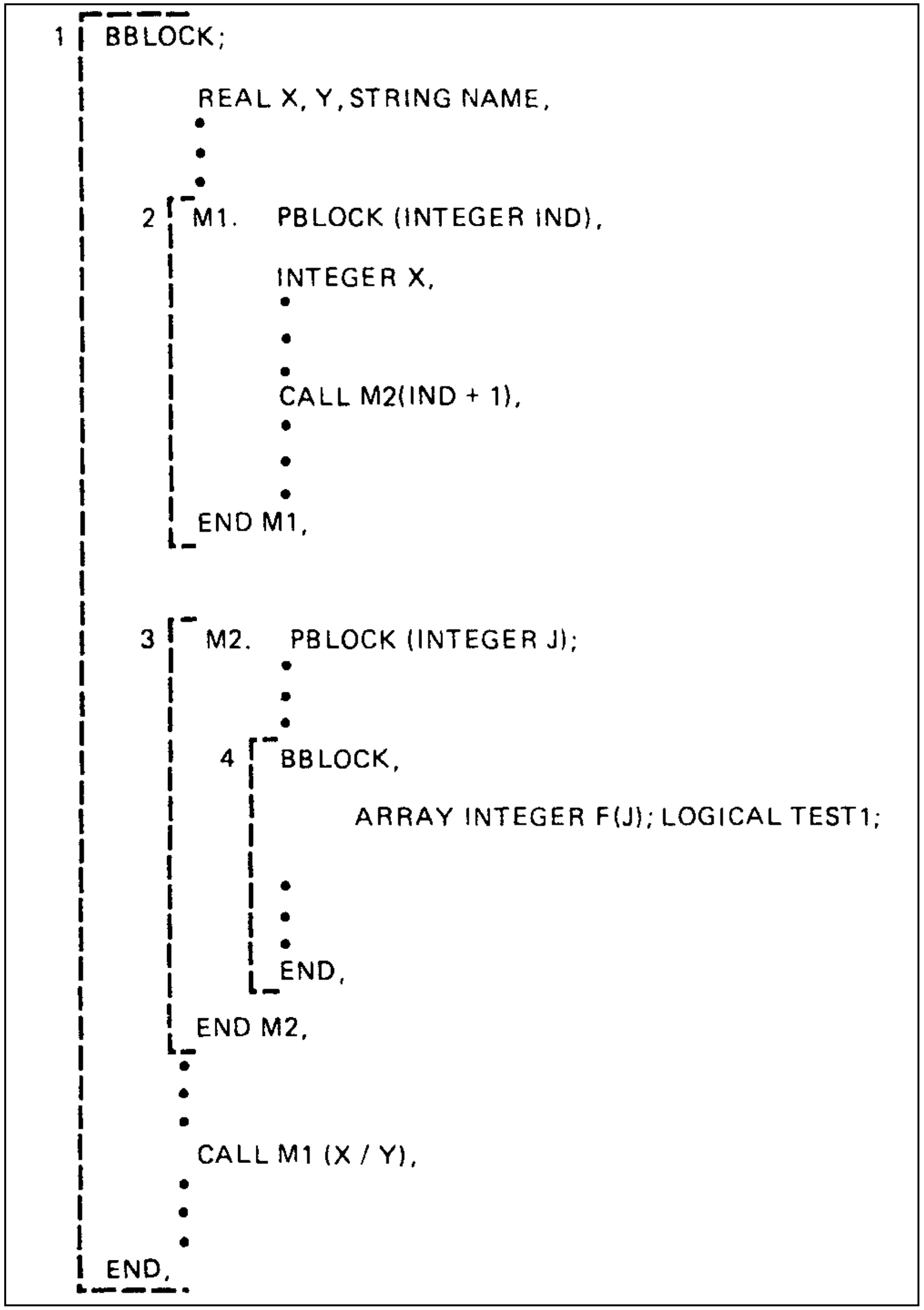


Current scope pointer points to current scope

- Nesting levels

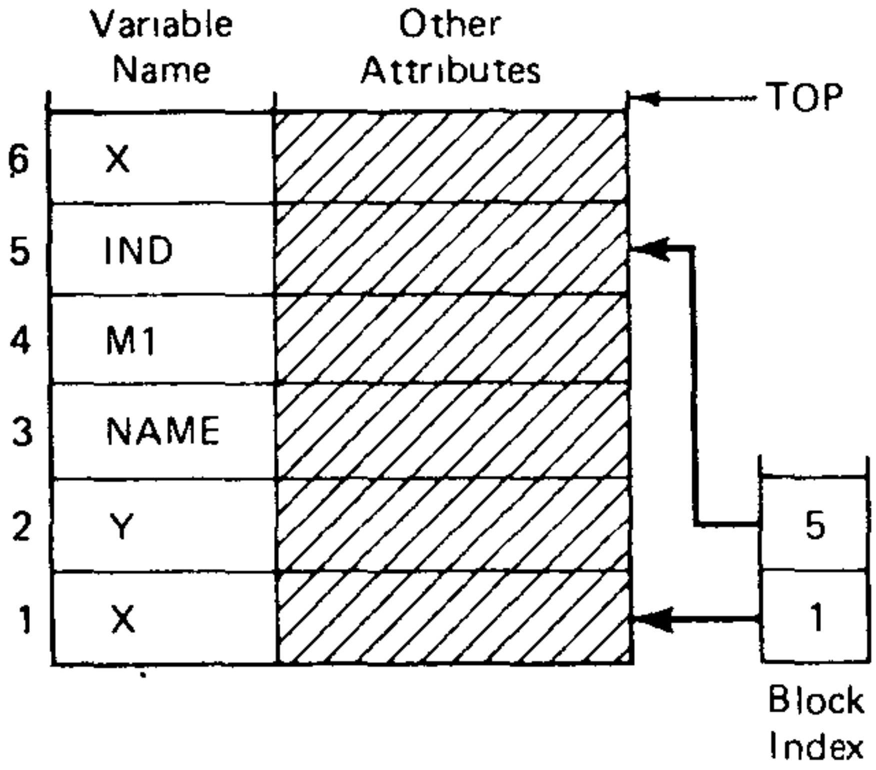
```
{
    int h, a, l;
    {
        int a, c, m;
        …
    }
    …
}
```

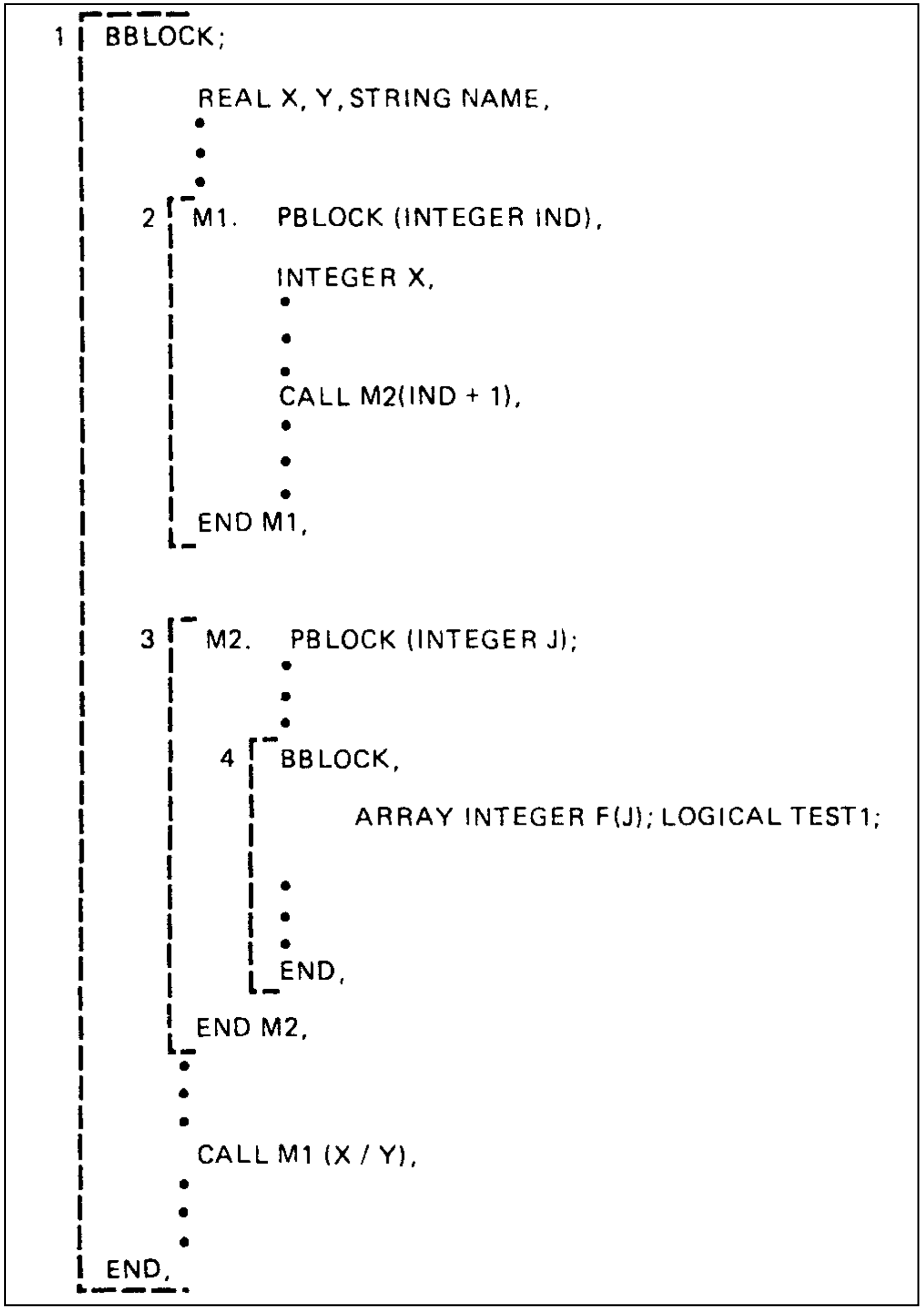
*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY



```
1  BBLOCK;
         REAL X, Y, STRING NAME,
         •
         •
         •
   2  M1.    PBLOCK (INTEGER IND),
                 INTEGER X,
                 •
                 •
                 •
                 CALL M2(IND + 1),
                 •
                 •
                 •
      END M1,

   3  M2.    PBLOCK (INTEGER J);
                 •
                 •
                 •
      4  BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
                 •
                 •
                 •
         END,
      END M2,
      •
      •
      •
      CALL M1 (X / Y),
      •
      •
      •
   END,
```

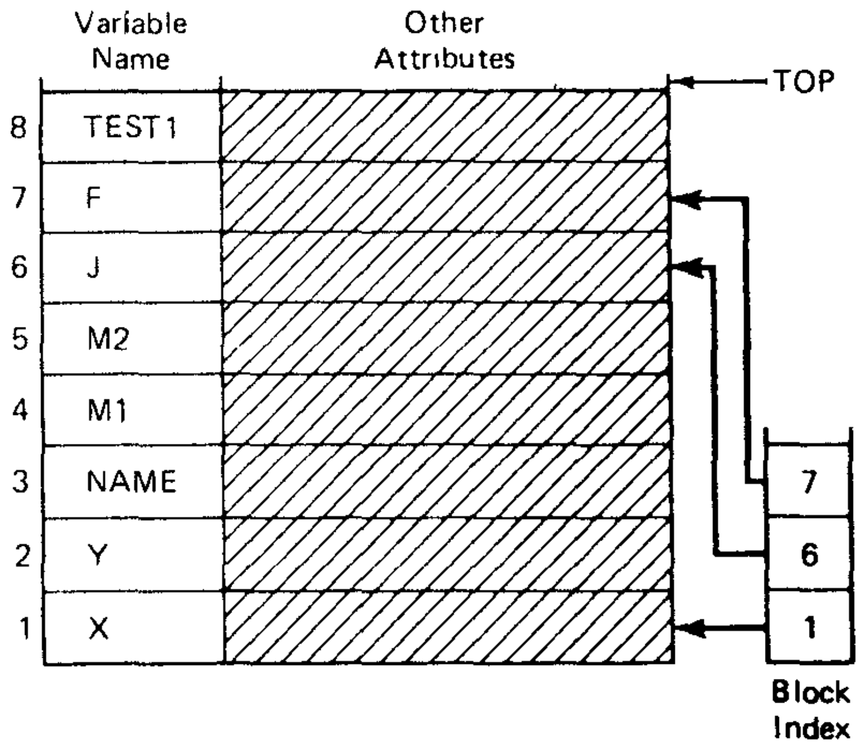
- Stack Symbol table just prior to completing the compilation of block 2





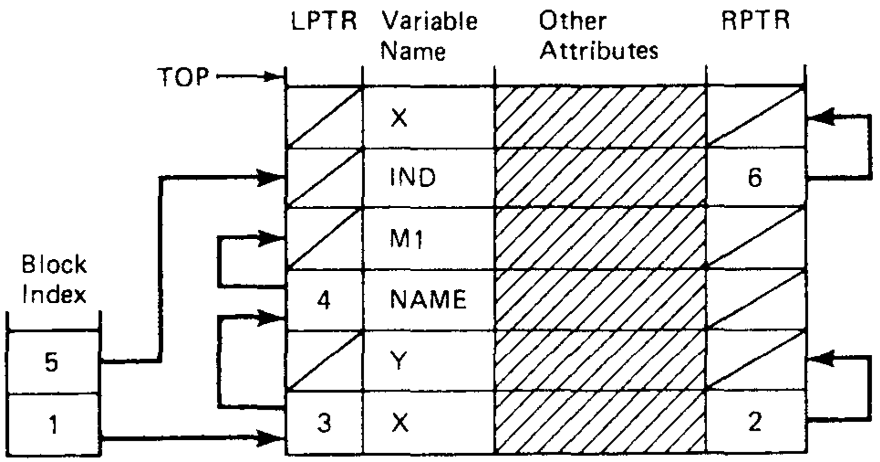
```
1  BBLOCK;
        REAL X, Y, STRING NAME,
        .
        .
        .
   2  M1.   PBLOCK (INTEGER IND),
              INTEGER X,
              .
              .
              .
              CALL M2(IND + 1),
              .
              .
              .
        END M1,

   3  M2.   PBLOCK (INTEGER J);
              .
              .
              .
      4  BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
              .
              .
              .
              END,
        END M2,
        .
        .
        .
        CALL M1 (X / Y),
        .
        .
        .
   END,
```

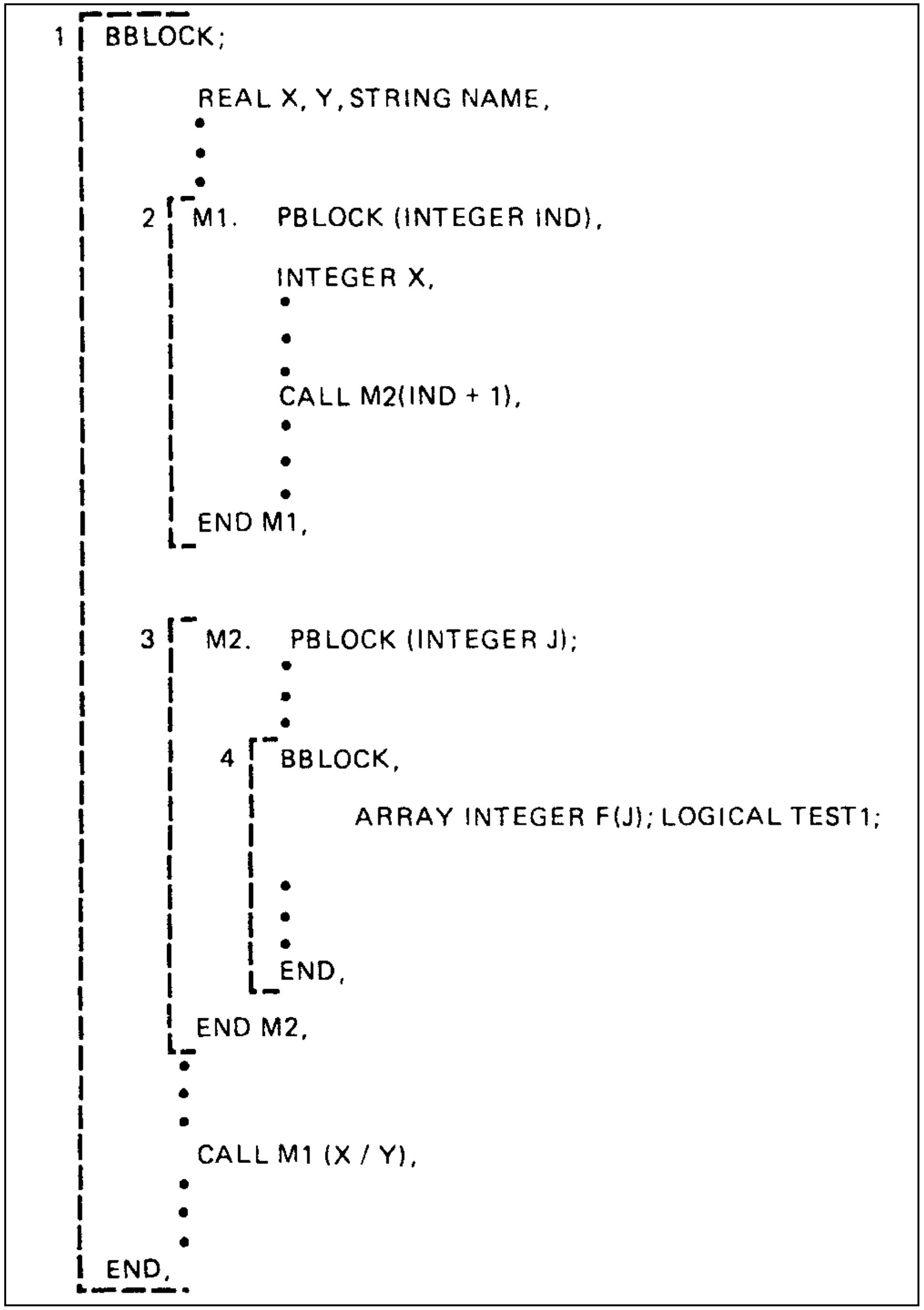
- Stack Symbol table just prior to completing the compilation of block 4



```
1  BBLOCK;
          REAL X, Y, STRING NAME,
          .
          .
          .
   2  M1.    PBLOCK (INTEGER IND),
             INTEGER X,
             .
             .
             .
             CALL M2(IND + 1),
             .
             .
             .
          END M1,

   3   M2.   PBLOCK (INTEGER J);
             .
             .
             .
      4   BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
             .
             .
             .
             END,
          END M2,
          .
          .
          .
          CALL M1 (X / Y),
          .
          .
          .
   END,
```

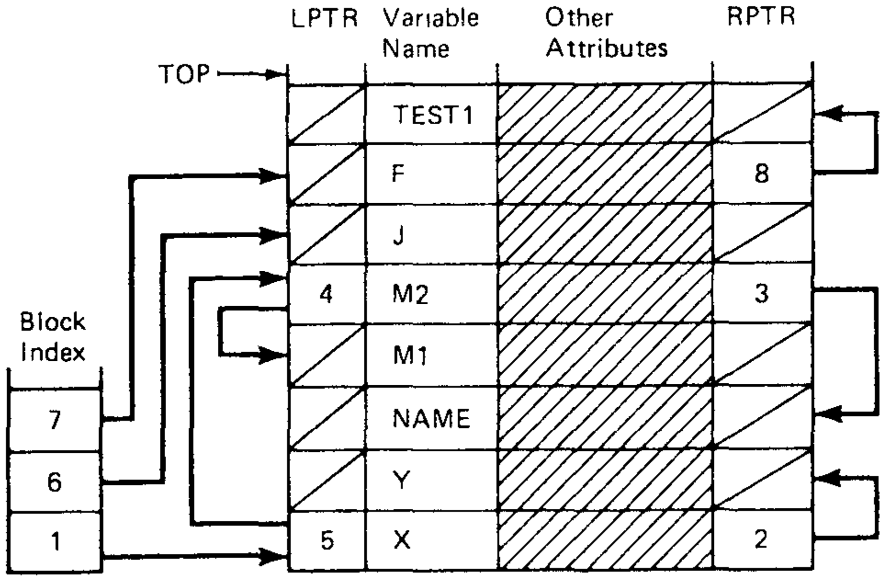
- Stack-implemented Tree-structured Symbol table just prior to completing the compilation of block 2





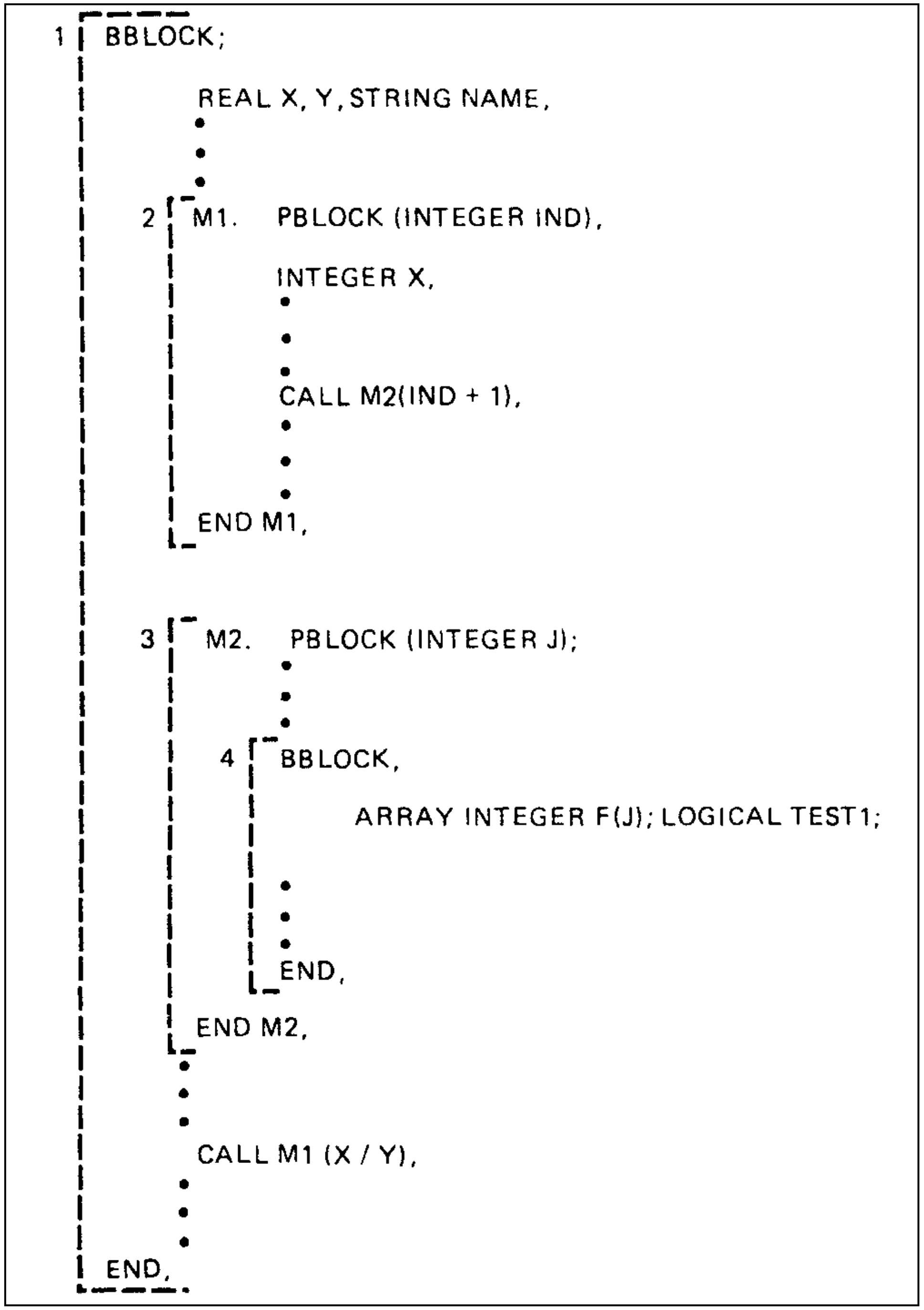
```
1  BBLOCK;
       REAL X, Y, STRING NAME,
       •
       •
       •
   2  M1.    PBLOCK (INTEGER IND),
             INTEGER X,
             •
             •
             •
             CALL M2(IND + 1),
             •
             •
             •
       END M1,

   3  M2.    PBLOCK (INTEGER J);
             •
             •
             •
       4  BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
             •
             •
             •
             END,
       END M2,
       •
       •
       •
       CALL M1 (X / Y),
       •
       •
       •
   END,
```

- Stack-implemented Tree-structured Symbol table just prior to completing the compilation of block 4



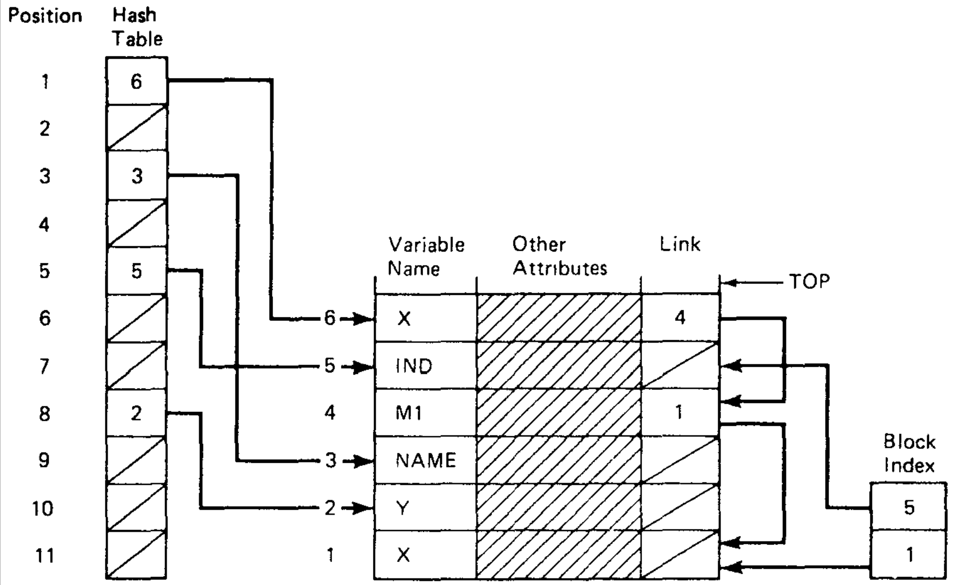
LPTR	Variable Name	Other Attributes	RPTR
	TEST1	▨	
	F	▨	8
	J	▨	
4	M2	▨	3
	M1	▨	
	NAME	▨	
	Y	▨	
5	X	▨	2

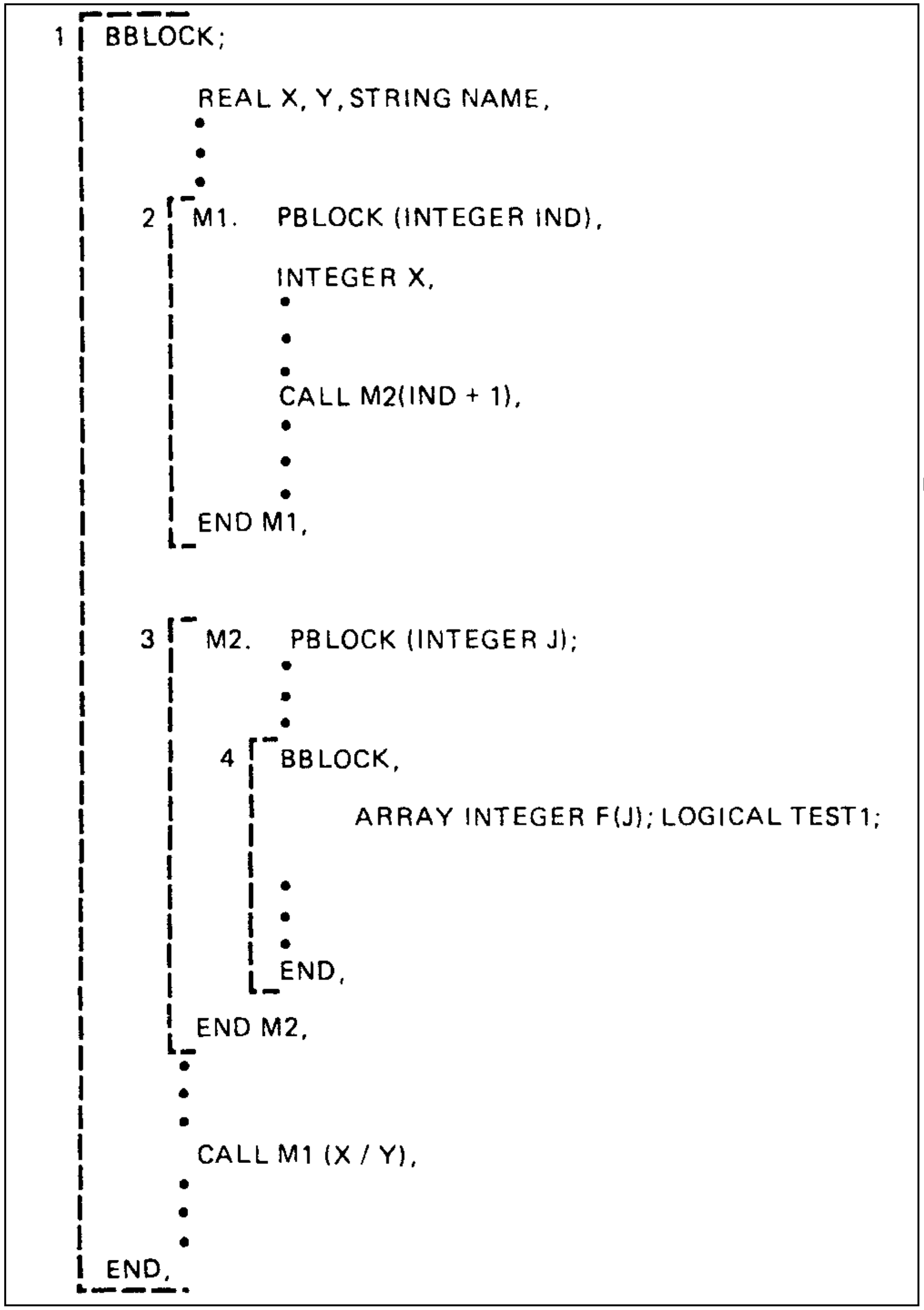
NOTE: for LPTR of X = 5: When M2 is added, tree becomes left heavy, apply double rotation to balance the tree.



```
1 BBLOCK;
         REAL X, Y, STRING NAME,
         .
         .
         .
    2  M1.   PBLOCK (INTEGER IND),
               INTEGER X,
               .
               .
               .
               CALL M2(IND + 1),
               .
               .
               .
         END M1,

    3   M2.   PBLOCK (INTEGER J);
               .
               .
               .
        4  BBLOCK,
                  ARRAY INTEGER F(J); LOGICAL TEST1;
                  .
                  .
                  .
               END,
         END M2,
         .
         .
         .
         CALL M1 (X / Y),
         .
         .
         .
  END,
```

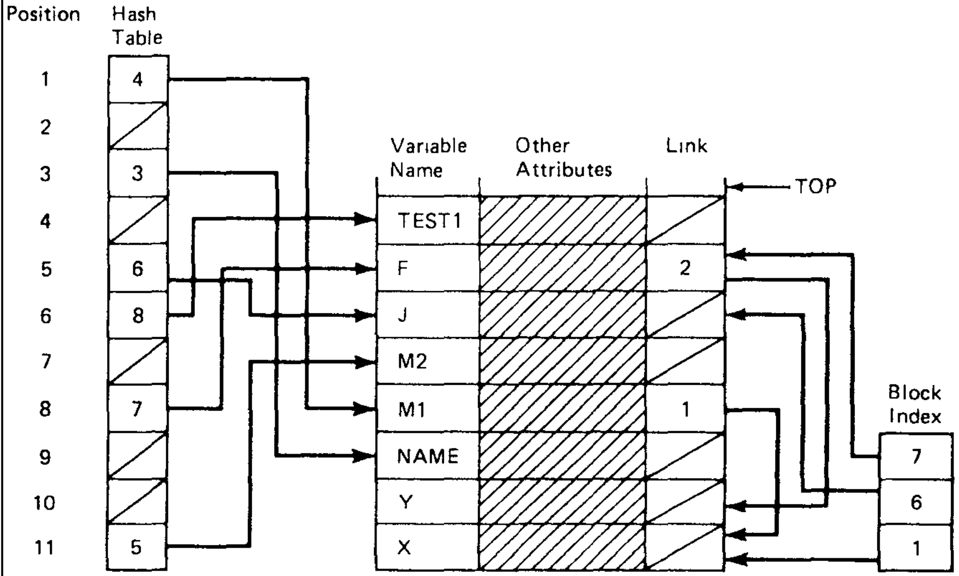
- Stack-implemented Hash Symbol table just prior to completing the compilation of block 2





```
1  BBLOCK;
         REAL X, Y, STRING NAME,
         •
         •
         •
   2  M1.   PBLOCK (INTEGER IND),
            INTEGER X,
            •
            •
            •
            CALL M2(IND + 1),
            •
            •
            •
      END M1,

   3   M2.   PBLOCK (INTEGER J);
             •
             •
             •
      4  BBLOCK,
               ARRAY INTEGER F(J); LOGICAL TEST1;
               •
               •
               •
         END,
      END M2,
      •
      •
      •
      CALL M1 (X / Y),
      •
      •
      •
   END,
```

- Stack-implemented Hash Symbol table just prior to completing the compilation of block 4



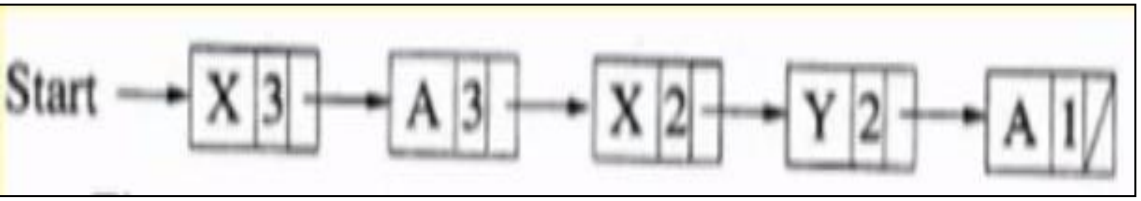
Limitations of One Table for each scope

- For a single-pass compiler, the table can be popped out and destroyed when the scope is closed but same is not true for multi-pass compiler.
  - **Closing of scope** means deleting all the variables declared in that scope when that scope ends.
- Search may be expensive if the variable is defined much above in the hierarchy.
- What about the table size allotted to each block??
  - Underutilized or insufficient table size if estimation is not proper.

One Table for All Scopes

- All identifiers are stored in a single table.
- Each entry in the symbol table has an extra field for identifying the scope.
- To search for an identifier, start with the highest scope number, then try the entries with next lesser scope numbers and so on.
- When a scope gets closed, all the identifiers with that scope number are removed from the table.
- More suitable for single pass compilers.
- Here also table can be represented as list, tree or hash table.

One Table for All Scopes -List



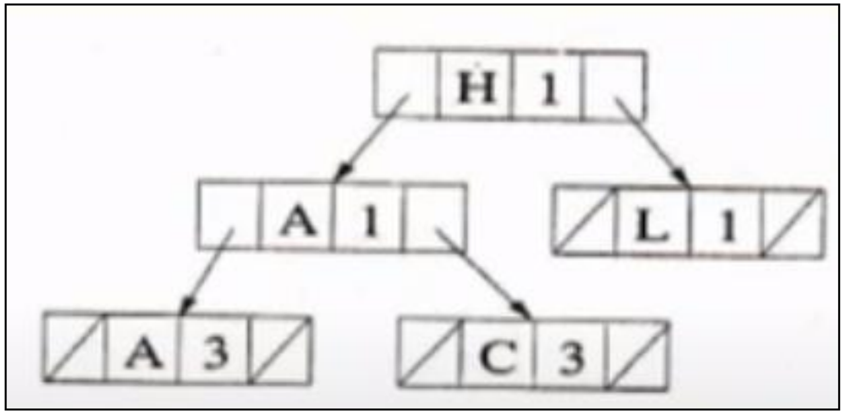
- Nesting Levels

```
{
    int A;
    {
        int X, Y;
        {
            int X, A;
            A = …
        }
    }
}
```

Start searching from Start and find the first occurrence

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

One Table for All Scopes -Tree

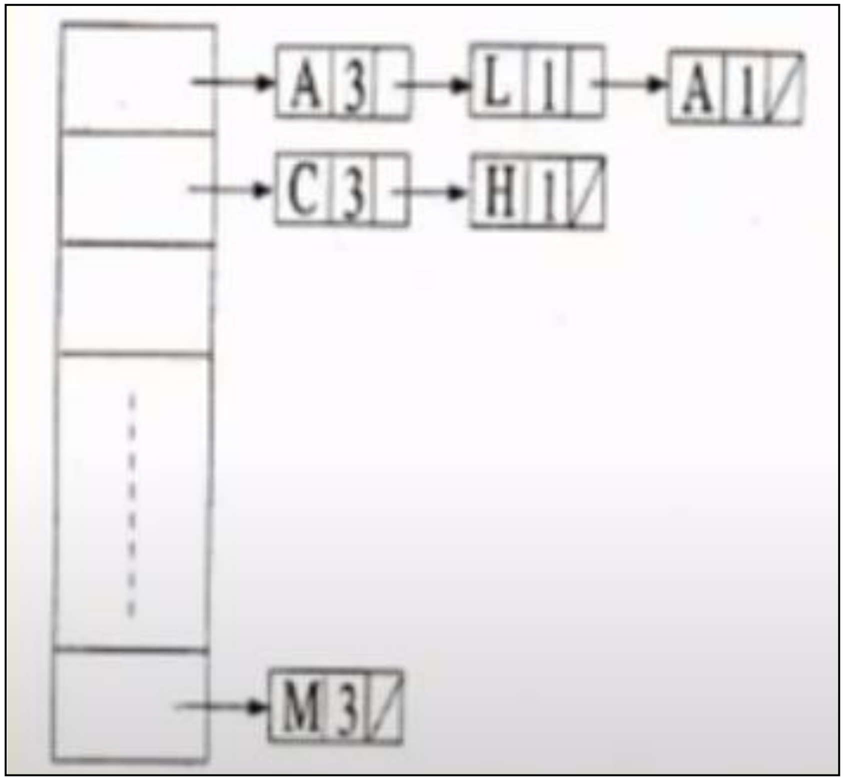


- Nesting Levels

```
{
    int H, A, L;
    {
        {
                int A, C;
        }
    }
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

One Table for All Scopes – Hash



- Nesting Levels

```
{
    int L, A, H;
    {
        {
                int A, C, M;
        }
    }
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

Advantage of using a Single Table

- Multiple tables need not be maintained.
- All information can be found from one table.
- But it totally depends on the compiler designer to choose between one table for all scope or one table for each scope.