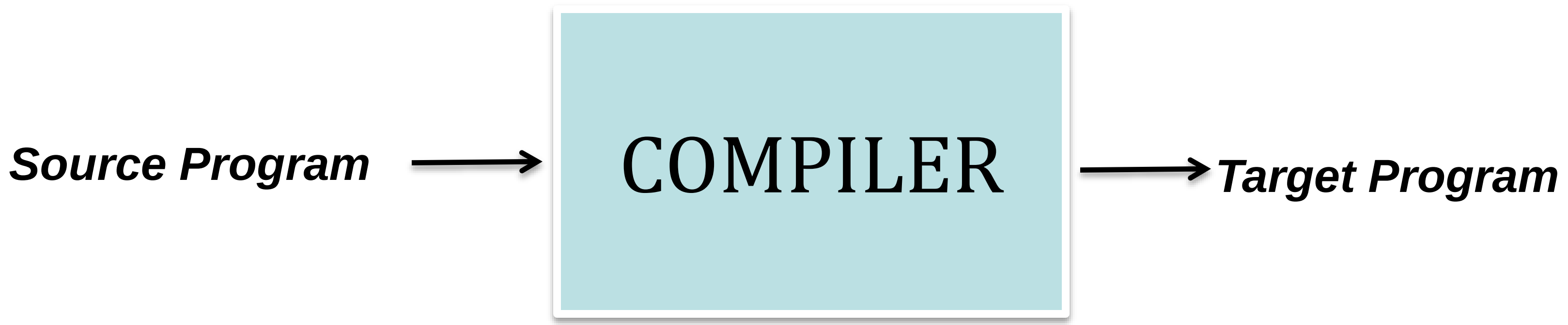
SYMBOL TABLE DESIGN

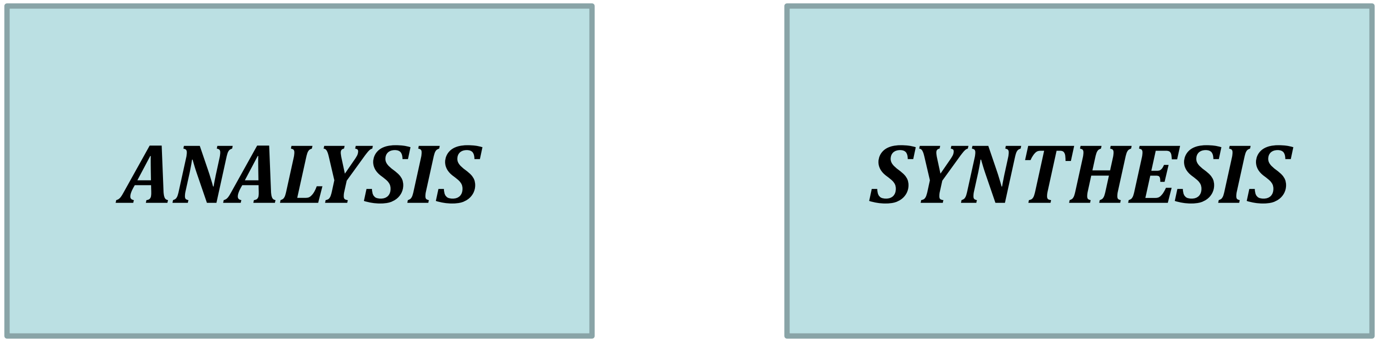
THE STRUCTURE OF A COMPILER

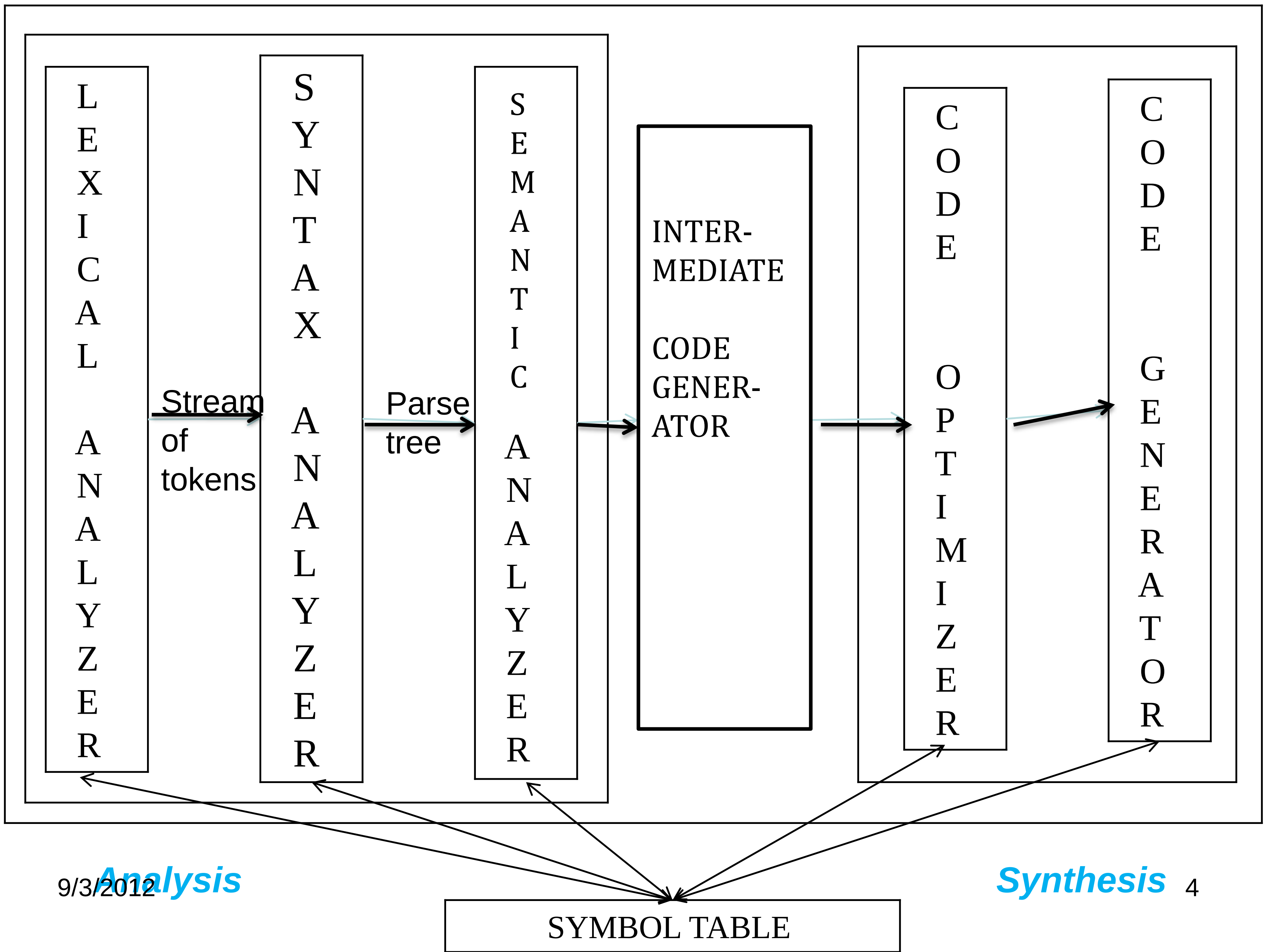
- Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.



WHAT'S INSIDE THIS BOX?

- If we open up this box a little, we see that there are two parts to this mapping:




LEXICAL ANALYZER
Stream of tokens
SYNTAX ANALYZER
Parse tree
SEMANTIC ANALYZER
INTER-MEDIATE CODE GENER-ATOR
CODE OPTIMIZER
CODE GENERATOR
Analysis
Synthesis
SYMBOL TABLE

ANALYSIS

Breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

The analysis part also collects information about the source program and stores it in a data structure called a ***symbol table,*** *which is passed along with the intermediate representation to the synthesis part.*

SYNTHESIS

- The synthesis part constructs the desired target program from the intermediate representation and the information in the ***symbol table***

ANALYSIS
- Front end of compiler

SYNTHESIS
- Back end of compiler

COMPILERS ROLE??

- An essential function of a compiler –

Record the variable names used in the source program and collect information about various attributes of each name.

- These attributes may provide information about the storage allocated for a name , its type and its scope , procedure names ,number and types of its arguments, the method of passing each argument and the type returned

SO , WHAT EXACTLY IS SYMBOL TABLE??

*Symbol tables are **data structures** that are used by compilers to hold information about source-program constructs.*

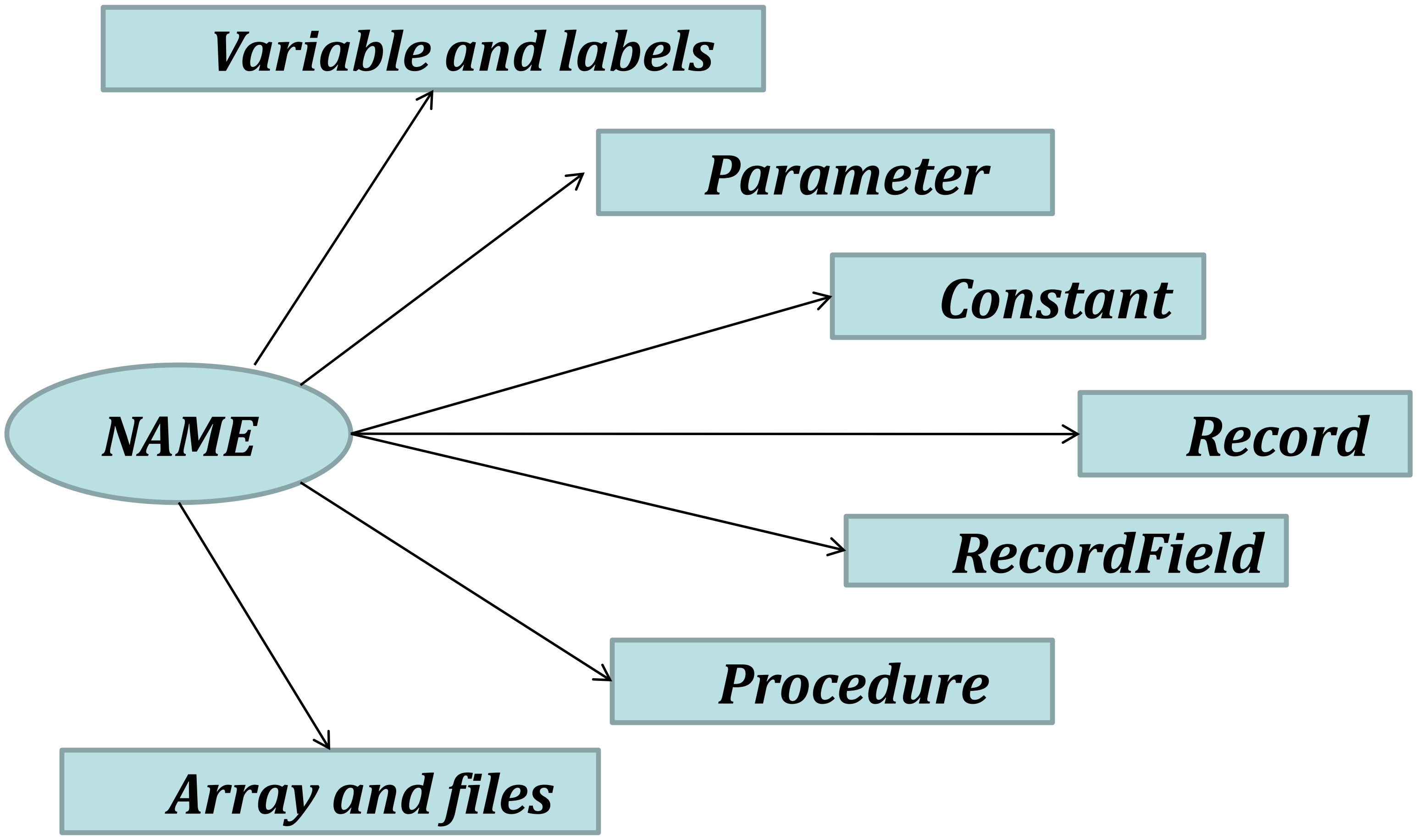
A symbol table is a necessary component because

- Declaration of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text

INFORMATION PROVIDED BY SYMBOL TABLE

- *Given an Identifier which name is it?*
- *What information is to be associated with a name?*
- *How do we access this information?*

SYMBOL TABLE - NAMES



SYMBOL TABLE-ATTRIBUTES

- Each piece of information associated with a name is called an ***attribute.***
- Attributes are language dependent.
- Different classes of Symbols have different Attributes

Variable, Constants	Procedure or function	Array
• Type , Line number where declared , Lines where referenced , Scope	• Number of parameters, parameters themselves, result type.	• # of Dimensions, Array bounds.

WHO CREATES SYMBOL TABLE??

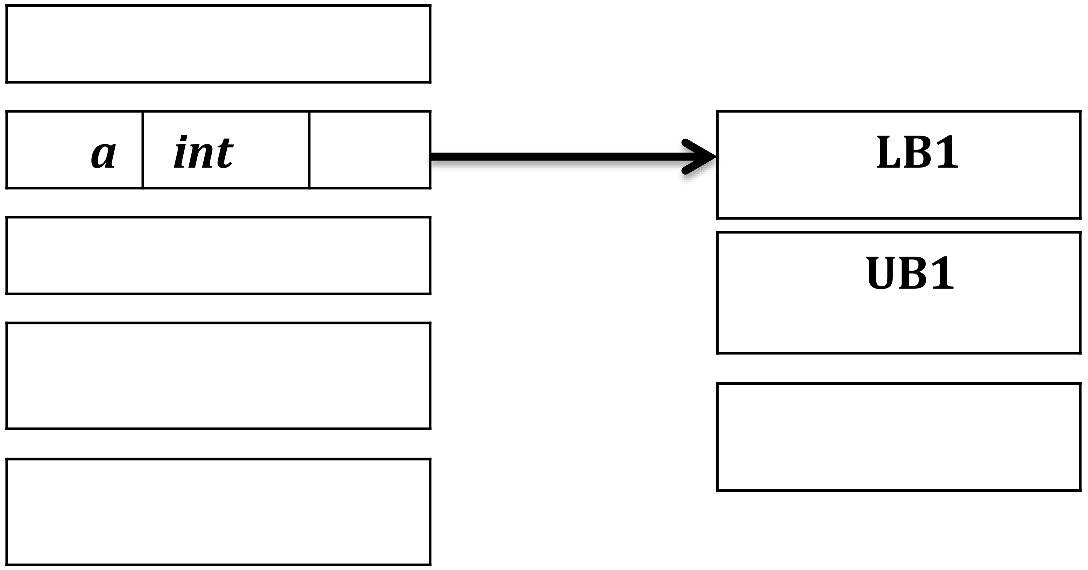
- Identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier
- In simple languages with only global variables and implicit declarations:
  - ✓ The scanner can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
  - ✓ The parser and/or semantic analyzer enter identifiers and corresponding attributes

USE OF SYMBOL TABLE

- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

IMPLEMENTATION OF SYMBOL TABLE

- Each entry in the symbol table can be implemented as a ***record*** consisting of several field.
- These fields are ***dependent*** on the information to be saved about the name
- But since the information about a name depends on the usage of the name the ***entries in the symbol table records will not be uniform.***
- Hence to keep the symbol tables records uniform some information are kept outside the symbol table and a pointer to this information is stored in the symbol table record.

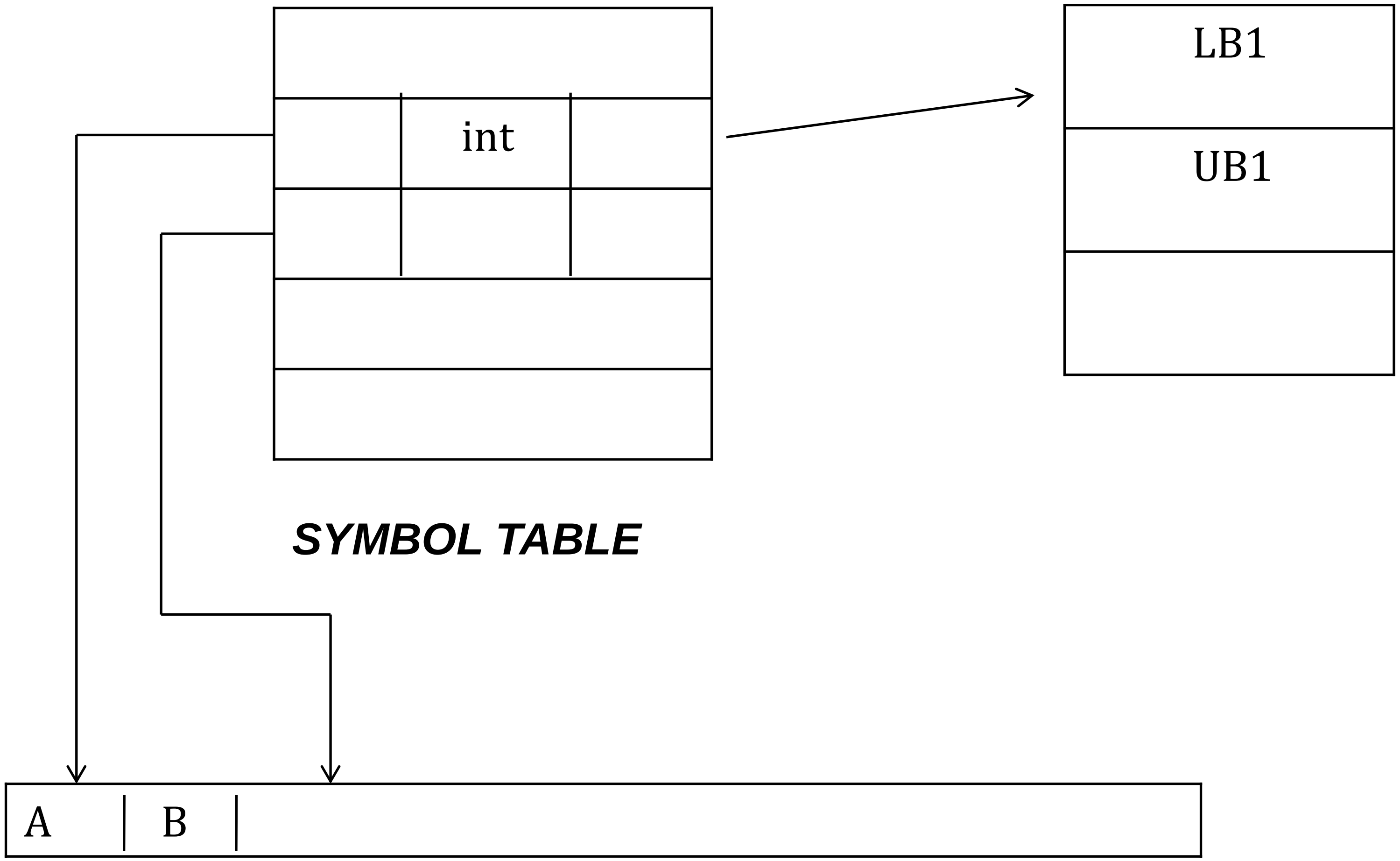


SYMBOL TABLE

A pointer steers the symbol table to remotely stored information for array a.

WHERE SHOULD NAMES BE HELD??

- If there is modest upper bound on the length of the name , then the name can be stored in the symbol table record itself.
- But If there is no such limit or the limit is already reached then an indirect scheme of storing name is used.
- A separate array of characters called a ‘***string table***’ is used to store the name and a pointer to the name is kept in the symbol table record



SYMBOL TABLE

STRING TABLE

SYMBOL TABLE AND SCOPE

- Symbol tables typically need to support multiple declarations of the same identifier within a program.

The scope of a declaration is the portion of a program to which the declaration applies.

- We shall implement scopes by setting up ***a separate symbol table for each scope.***

- The rules governing the scope of names in a block-structured language are as follows
  1. A name declared within a block B is valid only within B.
  2. If block B1 is nested within B2, then any name that any name that is valid for B2 is also valid for B1,unless the identifier for that name is re-declared in B1.
- The scope rules required a more complicated symbol table organization than simply a list association between names and attributes.
- Each table is list names and there associated attributes and the tables are organized into a stack.

SYMBOL TABLE ORGANIZATION

TOP →

z	Real
Y	Real
x	Real

Symbol table for block q

q	Real
a	Real
x	Real

Symbol table for p

Symbol table for main

P	Proc
Y	Integer
X	Integer

Var x,y : integer

Procedure P:
Var x,a :boolean;

Procedure q:
Var x,y,z : real;

begin
……
end
begin
…..
End

NESTING DEPTH

- Another technique can be used to represent scope information in the symbol table.
- We store the nesting depth of each procedure block in the symbol table and use the [***procedure name , nesting depth***] pairs as the key to accessing the information from the table.

A nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure.

- This number is basically a count of how many procedures are there in the referencing environment of the procedure .

```
Var x,y : integer

Procedure P:
Var x,a :boolean;

Procedure q:
Var x,y,z : real;

begin
……
end
begin
…..
End
```

x	3	Real
y	3	Real
z	3	Real
q	2	Proc
a	2	Boolean
x	2	Boolean
P	1	Proc
y	1	Integer
z	1	integer

SYMBOL TABLE DATA STRUCTURES

- Issues to consider : Operations required
    - Insert
        - Add symbol to symbol table
    - Look UP
        - Find symbol in the symbol table (and get its attributes)
- Insertion is done only once
- Look Up is done many times
- Need Fast Look Up
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

LINKED LIST

- A linear list of records is the easiest way to implement symbol table.
- The new names are added to the symbol table in the order they arrive.
- Whenever a new name is to be added to be added it is first searched linearly or sequentially to check if or the name is already present in the table or not and if not , it is added accordingly.
- Time complexity – O(n)
- Advantage – less space , additions are simple
- Disadvantages - higher access time.

UNSORTED LIST

```
01  PROGRAM Main
02          GLOBAL a,b
03          PROCEDURE P (PARAMETER x)
04                  LOCAL a
05          BEGIN {P}
06                  …a…
07                  …b…
08                  …x…
09          END  {P}
10  BEGIN{Main}
11          Call P(a)
12 END {Main}
```

Look up Complexity $O(n)$

Name	Characteristic Class	Scope	Other Attributes		
			Declared	Referenced	Other
Main	Program	0	Line 1		
a	Variable	0	Line 2	Line 11	
b	Variable	0	Line 2	Line 7	
P	Procedure	0	Line 3	Line 11	1, parameter, x
x	Parameter	1	Line 3	Line 8	
a	Variable	1	Line 4	Line 6	

SORTED LIST

```
01  PROGRAM Main
02          GLOBAL a,b
03          PROCEDURE P (PARAMETER x)
04                  LOCAL a
05          BEGIN {P}
06                  …a…
07                  …b…
08                  …x…
09          END  {P}
10  BEGIN{Main}
11          Call P(a)
12 END {Main}
```

Look up Complexity $O(\log_2 n)$

If stored as array (complex insertion)

Look up Complexity $O(n)$

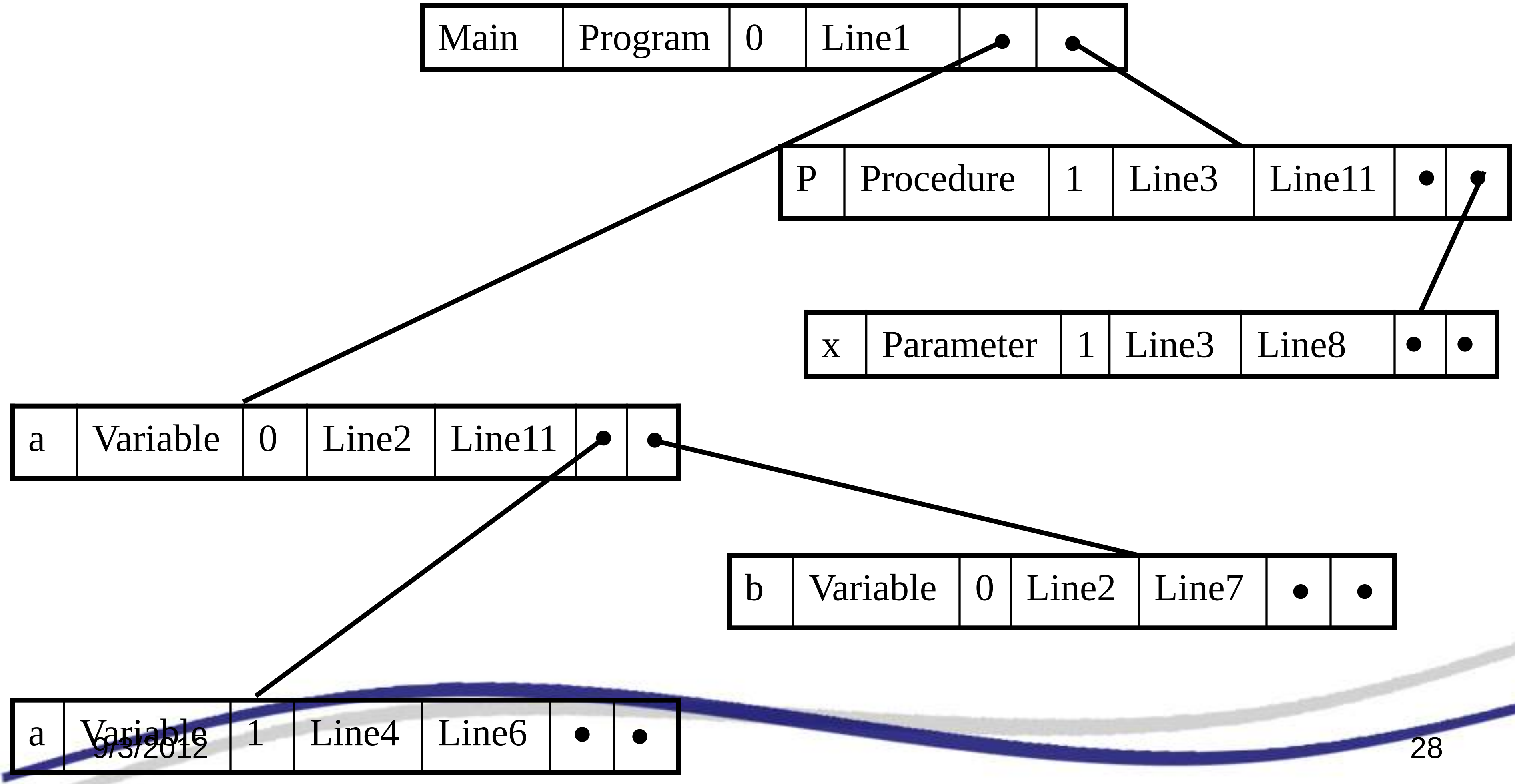
If stored as linked list (easy insertion)

Name	**Characteristic Class**	**Scope**	**Other Attributes**		
			Declared	**Reference**	**Other**
a	Variable	0	Line 2	Line 11	
a	Variable	1	Line 4	Line 6	
b	Variable	0	Line 2	Line 7	
Main	Program	0	Line 1		
P	Procedure	0	Line 3	Line 11	1, parameter, x
x	Parameter	1	Line 3	Line 8	

SEARCH TREES

- Efficient approach for symbol table organisation
- We add two links left and right in each record in the search tree.
- Whenever a name is to be added first the name is searched in the tree.
- If it does not exists then a record for new name is created and added at the proper position.
- This has alphabetical accessibility.

BINARY TREE


Main | Program | 0 | Line1
P | Procedure | 1 | Line3 | Line11
x | Parameter | 1 | Line3 | Line8
a | Variable | 0 | Line2 | Line11
b | Variable | 0 | Line2 | Line7
a | Variable | 1 | Line4 | Line6

BINARY TREE

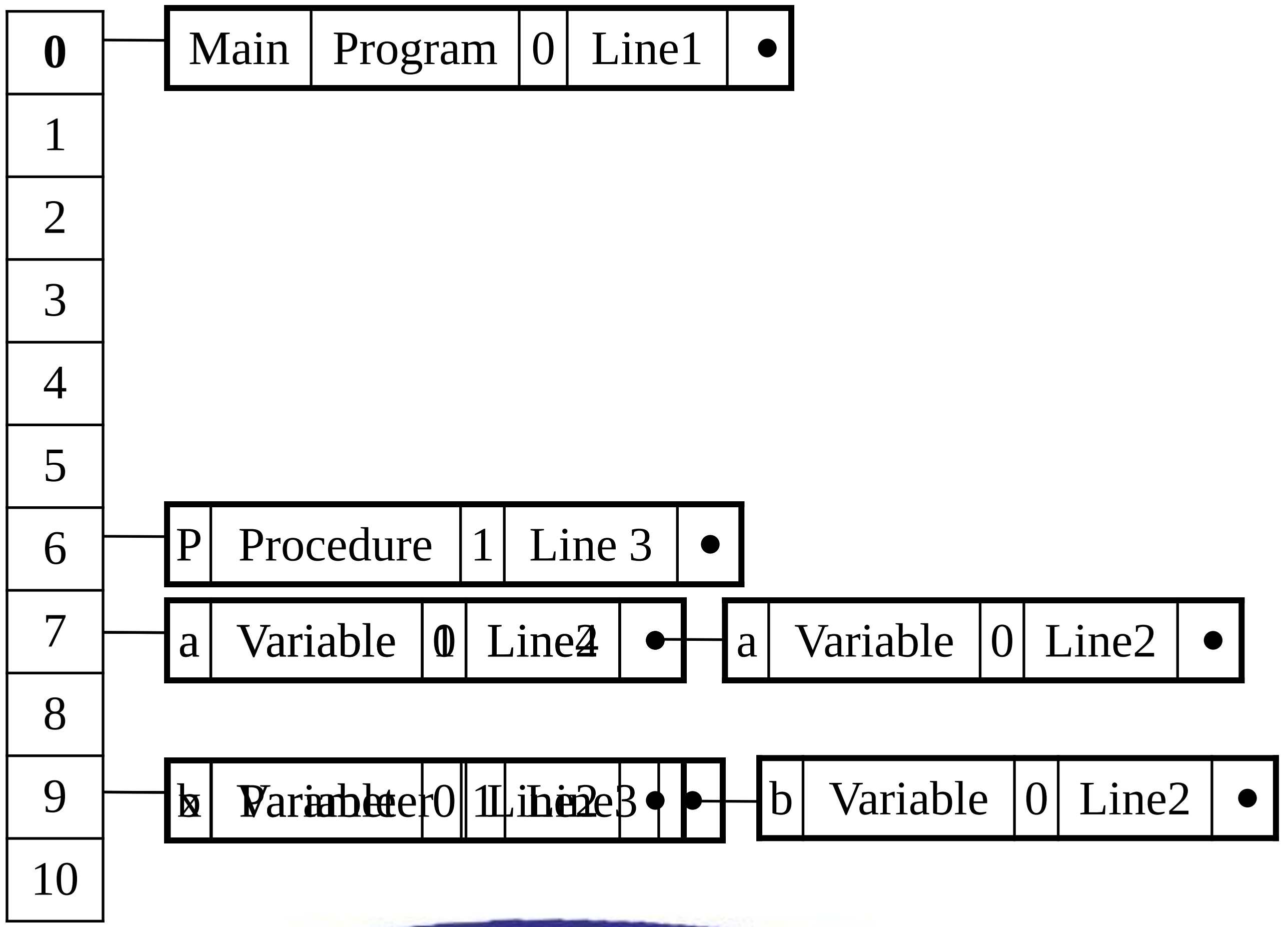
Lookup complexity if tree balanced	$O(\log_2 n)$
Lookup complexity if tree unbalanced	$O(n)$

HASH TABLE

- Table of k pointers numbered from zero to k-1 that points to the symbol table and a record within the symbol table.
- To enter a name in to the symbol table we found out the hash value of the name by applying a suitable hash function.
- The hash function maps the name into an integer between zero and k-1 and using this value as an index in the hash table.

HASH TABLE - EXAMPLE

M	n	a	b	P	x
77	110	97	98	80	120


0
1
2
3
4
5
6
7
8
9
10
Main | Program | 0 | Line1 | •
P | Procedure | 1 | Line 3 | •
a | Variable | 0 | Line2 | •
a | Variable | 1 | Line4 | •
x | Parameter | 1 | Line3 | •
b | Variable | 0 | Line2 | •
b | Variable | 0 | Line2 | •

```
PROGRAM Main
GLOBAL a,b
PROCEDURE P(PARAMETER x)
LOCAL a
BEGIN (P)
...a...
...b...
...x...
END (P)
BEGIN  (Main)
Call P(a)
End (Main)
```

H(Id) = (# of first letter + # of last letter) mod 11

REFERENCES

- *O.G.Kakde - Compiler design*
- *Compilers Principles, Techniques & Tools - Second Edition : Alfred V Aho , Monica S Lam, Ravi Sethi, Jeffery D Ullman*

THANK YOU…