

# **Course: Compiler Construction**

## **Week 12: Symbol Table Management**

**Lecturer:** Martha Gichuki

# Course Description

- The course begins with a coverage of basic Principles in Compiler Design with an introduction to Formal programming language translation and compiler design concepts; compilers and interpreters.
- A coverage of Language theory, Parsing Context-Free Languages (Top - down and bottom - up parsing), translation specifications and machine-independent code optimization will then follow.

# Course Description ...

- The main phases of compilation i.e. Lexical, Syntax and Semantic analysis will be taught.
- Code generation and Optimization, Symbol table design, Program compilation, Loading and execution will then be covered.
- Compilation techniques, Optimization, Design of a simple complete compiler will culminate the course.

# Learning outcomes Week 12: Symbol Table Management

At the end of the lecture, you will be able to:

- i. Describe Symbol table information
- ii. Describe the process of creating and managing symbol tables
- iii. Describe how to build a symbol table

# Definition: Symbol Table

- ✓ A symbol table is a central repository where a compiler stores **essential information about every symbol** contained in a program<sup>1</sup>. This information includes:-
  - Keywords
  - Variable names
  - Constants
  - Procedures
  - Functions
  - Operators
  - Labels
  - Data types among others.

# Symbol Table....

- ✓ It is a data-structure maintained throughout all the phases of a compiler and is an integral part of the compilers Intermediate Representation (IR)
- ✓ It stores all the identifier names along with their types making it easier for the compiler to quickly search and retrieve identifier records.
- ✓ It localizes information derived from various parts of the source code as well as aiding in scope management<sup>2</sup>.

# Roles of the Symbol Table

- i. Enables easy and efficient access to information about symbols
- ii. Simplifies the design and implementation of any code that must refer to information about variables derived earlier in compilation.
- iii. Eliminates the need to represent the declarations directly in the IR, therefore doing away with the overhead of making each reference contain a pointer to the declaration<sup>3</sup>.

- ✓ It uses a **computed mapping** from the *textual name* to the *stored information*.
- ✓ Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer<sup>4</sup>.

# How a Symbol Table interacts with the phases of a compiler

- ✓ Every compiler phase uses the symbol table
- 1. *Initialization Phase* places keywords, operators and standard identifiers in it
- 2. *The scanner* places user-defined identifiers and literals in it and returns the corresponding token
- 3. *The parser* uses these tokens to create the parse tree which is the outcome from syntactic program analysis
- 4. *The semantic action* routines place the data type in its entries and uses this information to perform basic type-checking<sup>5</sup>

# How a Symbol Table interacts with the phases of a compiler...

5. *Intermediate Code Generation phase* uses pointers to entries in the symbol table to create intermediate Representation (IR) of the program
6. *The object code generation phase* uses pointers to entries in the symbol table for two functions:-
  - ✓ *To allocate storage for its variables and constants*
  - ✓ *To store the addresses of its procedures and functions*<sup>6</sup>

# Symbol Table Interaction with Semantic Analyzer

✓ The symbol table contains the following information about the input strings in a source program<sup>7</sup>: -

- i. Lexeme (input string) itself*
- ii. Corresponding token*
- iii. Its semantic component e.g. variable, operators, constant, function, procedure etc.*
- iv. Data type*
- v. Pointers to other entries (where necessary)*

- ✓ The **semantic analyzer** uses the **syntax tree** and the information in the symbol table to check the source program for semantic consistency with the language definition.
- ✓ It also gathers type information and saves it in either **the syntax tree or the symbol table**, for subsequent use during intermediate-code generation<sup>8</sup>.

- ✓ Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information.
- ✓ Symbol tables typically need to support multiple declarations of the same identifier within a program<sup>9</sup>.

# Symbol Table Interaction with Lexical Analyzer

- ✓ When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- ✓ In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer.
- ✓ This helps in determining the proper token to be passed to the parser<sup>10</sup>.

# Building a Symbol Table

✓ The symbol table defines **two interface routines** for the rest of the compiler.

1. *LookUp(name)* returns the record stored in the table at  $h(name)$  if one exists.

Otherwise, it returns a value indicating that name was not found.

2. *Insert(name, record)* stores the information in record in the table at  $h.name/$ . It may expand the table to accommodate the record for name<sup>11</sup>.

# Building a Symbol Table...

- ✓ In processing declaration syntax, the compiler builds up a set of attributes for each variable.
- ✓ When the parser recognizes a production that declares some variable, it can enter the name and attributes into the symbol table using *Insert*.
- ✓ If a variable name can appear in only one declaration, the parser can call *LookUp* first to detect a repeated use of the name<sup>12</sup>.

## Building a Symbol Table...

- ✓ The compiler *can use separate functions* for *LookUp* and *Insert*, or they can be combined by passing *LookUp* a flag that specifies whether or not to insert the name.
- ✓ This ensures that a *LookUp* of an undeclared variable will fail - a property useful for **detecting a violation** of the **declare-before-use** rule in Syntax-Directed Translation schemes or for supporting nested lexical scopes<sup>13</sup>.

- ✓ When the parser encounters a variable name outside the declaration syntax, it uses *LookUp* to obtain the appropriate information from the symbol table.
- ✓ *LookUp* fails on any **undeclared name**<sup>14</sup>.

- ✓ The compiler writer may need to **add** functions to **initialize** the table, to **store** it to and **retrieve it** from external media, and to **finalize** it.
- ✓ For a language with a single name space, this interface suffices<sup>15</sup>.

✓ Symbol tables maintain *name entries* in the following format:

*<symbol name, type, attribute>*

### **Example:-**

✓ Consider a symbol table that stores information about the following **variable declaration**:

*static int interest;*

✓ then it should store the entry such as:

*<interest, int, static>*

✓ The attribute clause contains the entries related to the name<sup>16</sup>.

- ✓ Symbol tables are created by the lexer and parser and later used by the code generator<sup>17</sup>.
- ✓ The following is a symbol table generated for some code:-

Name	Class	Type
X1	Variable	Integer
X2	Variable	Integer
xR	Variable	Real

# Symbol Table Implementation

- ✓ If a compiler is handling a small amount of data, then the symbol table can be implemented as an **unordered list**, which is easy to code, but it is only suitable for small tables only.
- ✓ A symbol table can be implemented in one of the following ways:
  - i. Linear (sorted or unsorted) list
  - ii. Binary Search Tree
  - iii. Hash table
- ✓ The most common method is hash tables, where the source code symbol itself is treated as a **key for the hash function** and the **return value** is the **information about the symbol**.
- ✓ A symbol table, whether linear or hash, should provide the **Insert and Lookup operations**<sup>18</sup>.

# Symbol Table Operations

## A. insert()

- ✓ The operation is frequently used in the analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table.
- ✓ The operation adds information in the symbol table about unique names present in the source code.
- ✓ The **format** or structure in which the names are stored depends upon the **compiler** being used
- ✓ A symbol attribute in the source code is the information associated with that symbol e.g. its value, state, scope, and type.
- ✓ The operation takes the symbol and its attributes as arguments e.g. *int a;* should be processed by the compiler as **insert(a, int)**<sup>19</sup>;

# Symbol Table Operations...

## B. lookup()

- ✓ The operation searches for a name in the symbol table to determine the following: -
  - i. Does the symbol exist in the table?
  - ii. Is it declared before use?
  - iii. Is the name used in the scope?
  - iv. Is the symbol initialized?
  - v. Is the symbol declared multiple times?
- ✓ The **format** of lookup() function varies according to the **programming language**. The basic format should be *lookup(symbol)*
- ✓ This method returns the symbol attributes if the symbol exists in the stored symbol table and zero(0) if the symbol *does not exist*<sup>20</sup>.

# Scope Management...

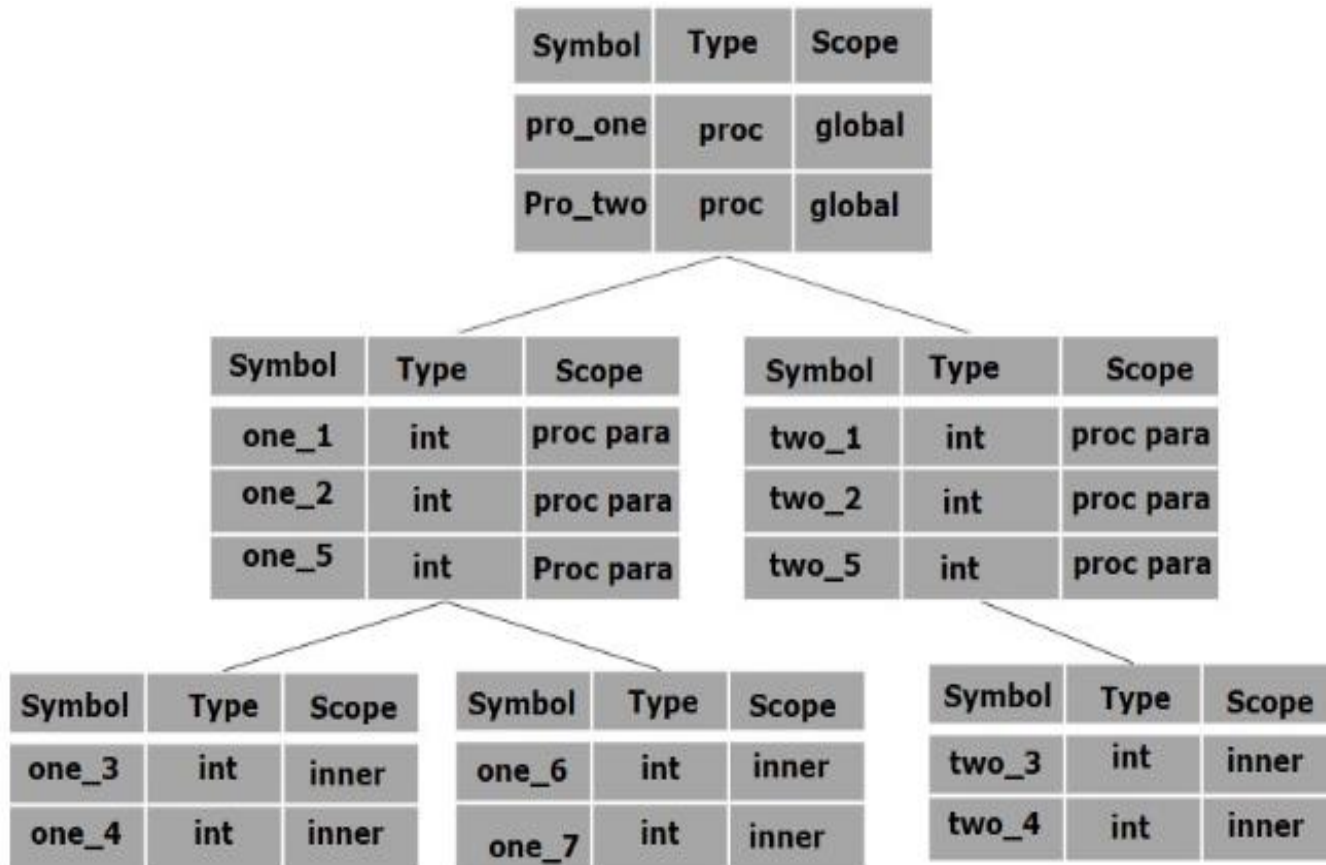
- ✓ The symbol table data structure hierarchy is stored in the semantic analyzer
- ✓ The following algorithm is used to search for a name in a symbol table:-
  - i. Search a symbol in the current scope, i.e. current symbol table.
  - ii. The search is completed if a name is found, else it will be searched in the parent symbol table until,
  - iii. either the name is found or global symbol table has been searched for the name<sup>21</sup>.

# Scope Management

- ✓ A compiler maintains two types of symbol tables:
  - i. A **global symbol table** which can be accessed by *all* the procedures
  - ii. A **scope symbol table** created *for each* scope in the program<sup>22</sup>.

# Scope Management

✓ To determine the scope of a name, symbol tables are arranged in a hierarchy structure as demonstrated in the example below<sup>23</sup>:-



# Three Basic Operations of a Symbol Table

## 1. *Adding Symbols*

- ✓ Reserved words, standard identifiers and operators are initialized while new lexemes, appropriate properties and attributes are *added* when the scanner encounters them.
- ✓ They are then assigned a token class<sup>24</sup>.

# Three Basic Operations of a Symbol Table ...

## 2. *Deleting Symbols*

- ✓ Upon finishing with a given scope of the program, the compiler *removes* all the symbols belonging to that scope effectively before processing a different program scope.
- ✓ This data may be hidden from the compilers' view or dumped into a temporary file<sup>25</sup>

# Three Basic Operations of a Symbol Table...

## 3. *Searching Symbols*

- ✓ Looking up a lexeme in the symbol table
- ✓ Retrieving any and all of the properties belonging to that symbol<sup>26</sup>

# Summary : Symbol Table Management

- ✓ A symbol table is a data structure that holds information about all symbols held in the source program.
- ✓ Information stored includes variable names, their size, types, etc.
- ✓ It does not form part of the final code generated<sup>27</sup>.

# Content Covered in Week 12: Symbol Table Management

At the end of the lecture, we were able to:

- i. Describe Symbol table information
- ii. Describe the process of creating and managing symbol tables
- iii. Describe how to build a symbol table

## Course Text Books

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; *Addison- Wesley Pub Co*, ISBN: 0201100886 (2007)
2. Compiler Construction: Principles and practices; Kenneth C. Loudon; *Cengage Learning*; 1st edition, ISBN-10 : 0534939724 (1997)
3. Basics of Compiler Design: Torben Mogensen; *DIKU University of Copenhagen Universitetsparken 1 DK-2100 Copenhagen DENMARK* (2007)
4. Engineering a Compiler: 2nd edition; Keith D. Cooper and Linda Torczon; *Morgan Kaufmann Publishers* ISBN: 978-0-12-088478-0 (2003)
5. Compiler Design: Santanu Chattopadhyay; *PHI Learning Publishers*, ISBN 812032725X, 9788120327252 (2005)