Paper on Symbol Table Implementation in Compiler Design- Dr.Jad Matta

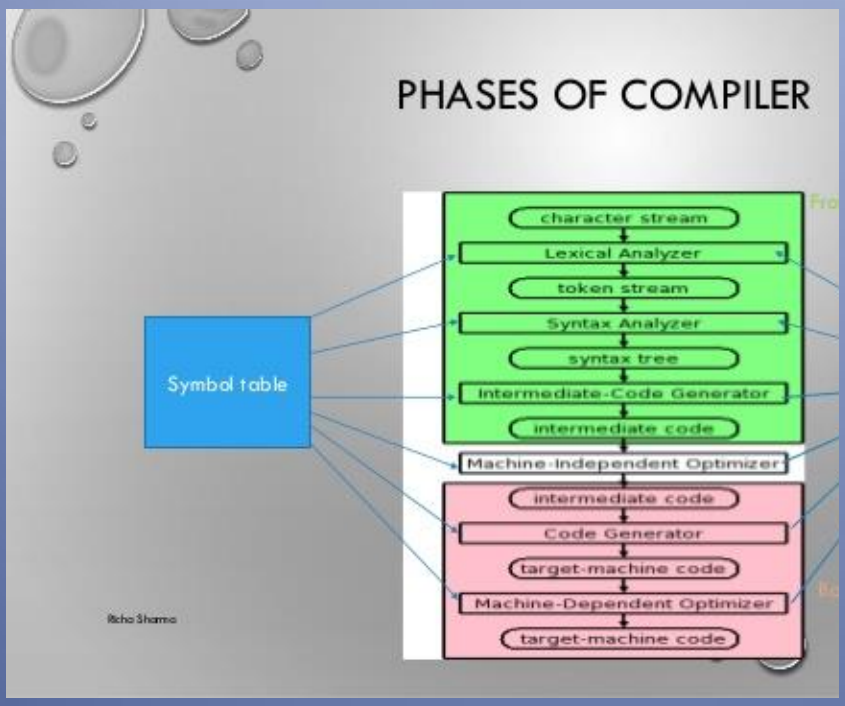
Method · December 2019

DOI: 10.13140/RG.2.2.14217.60005

1 author:

Jad Matta
American University of Beirut
15 PUBLICATIONS **0** CITATIONS

SEE PROFILE



Hash table implementation strategies in Symbol Table using JAVA and C++

Dr. Jad Matta

Abstract

This paper exposes different kinds of strategies in implementing compilers symbol table using JAVA and C++ as programming languages. At the beginning, the paper gives definition of a symbol table and it illustrates its role in identifying variables and identifiers in compiler design. Then, a complete implementation of a LinkedList of Hash tables is given in JAVA programming language. Then, the paper provides an implementation of a symbol table taking blocks and variables from a text file and outputting the scope and entry level of each variable. Finally, the paper concludes with a C++ source code that implements Open Addressing Scheme in hash tables.

Keywords:

Compiler Design, Hashtables, Symbol table, LinkedList of Hashtables, Java Programming, C++, blocks and scope.

Hash table implementation strategies in Symbol Table using JAVA and C++

Table of Contents

Table of Figures

1. What is a Symbol Table?

Symbol table is a data structure used by a compiler to keep track of semantic of variables and identifiers. This information will be used later on by the semantic analyzer and code generator. The symbol table stores the following information about identifiers and variable:

- The name as an array of characters
- The data type (int, float, double, long, bool)
- The block level
  - The block level in modern language is determined by the opening and closing of angle brackets { }
- The scope
  - Global
  - Local
  - Parameter
- The offset from the base pointer (local variables and parameters)\
- Function and methods name
  - Its return type (void, int, double, long etc…)
  - Number of formal parameters
  - Whether the function is calling itself (recursive)

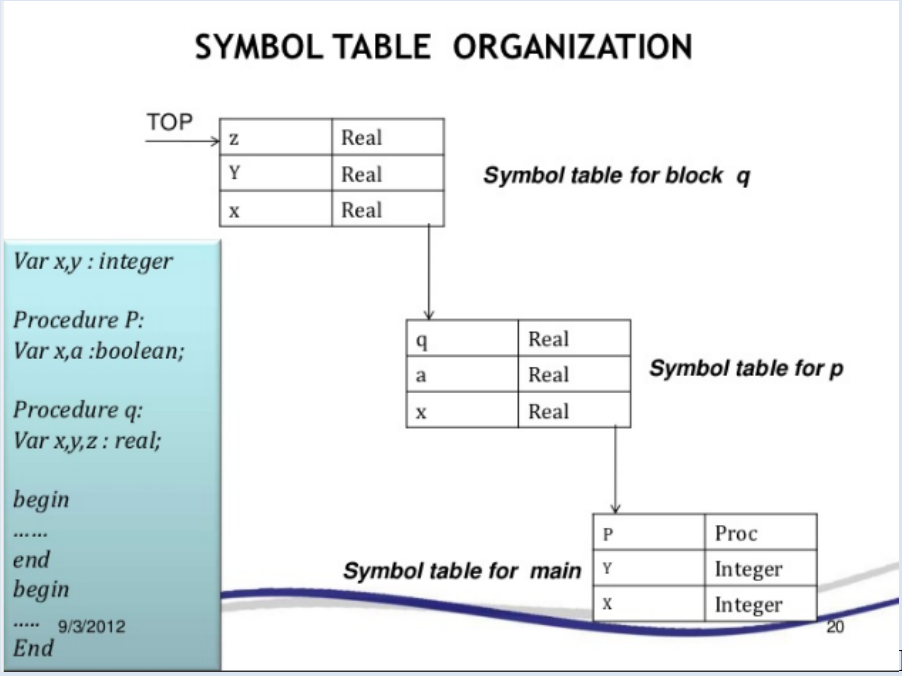


Fig.0

2. Symbol table implementations

2.1 Unordered list

Searching is very expensive since we have to scan on average half of the list to find the key element. It is mainly used for very small set of variables.

2.2 Ordered linear list

Insertion is expensive since we have to scan for the proper location inside the list to keep the array sorted.

We can use binary search for searching since the list is sorted as an average of $\log_2 n$.

2.3 Binary search tree

As long as we have none amortized tree, the average of searching and sorting is logarithmic.

2.4 Hash table

Hash table is the most efficient implementation of symbol table. It has two operations GET and PUT and performs well while having memory space adequately larger than the number of variables and identifiers

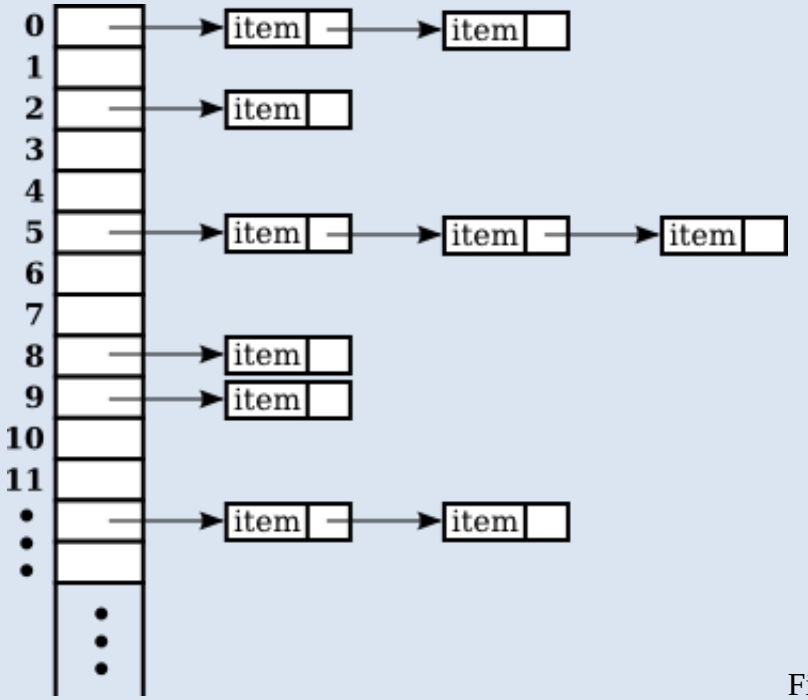


Fig.1

- Hash function h(n) returns a value between 0 and m-1, where n is the input and m is the hash table size.

A note on resolving collisions:

- Linear resolution:
  - (h(n) +1) mod m for m being a prime number
- Chaining
  - Open Hashing
  - Keep a chain on the items with the same hash value
- Quadratic-rehashing:
  - (h(n) + 1) mod m, and consecutively (h(n) + 2^2) mod m …….. till (h(n) + i^2) mod m.
- Double hashing
  - Distance between probes is calculated using another hash function

Performance of Hash Table:

1. Performance depends on collision resolution schemes
2. Hashtable size must be large enough than the number of entries
3. Keys should be distinct to avoid large collisions
4. Hashtable must be uniformly distributed

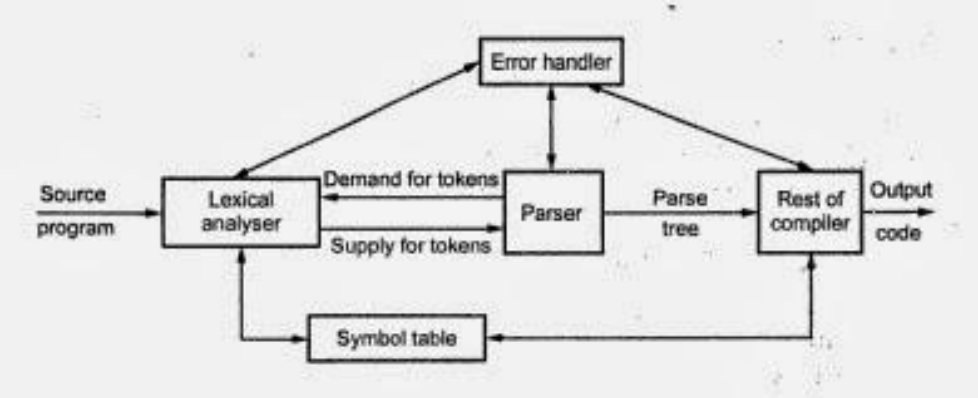


Fig.2

```
public class HashList
{
    private int key;
    private int value;
    private HashList next;

    public int getKey()
    {
        return key;
    }

    public int getValue()
    {
        return value;
    }

    public HashList(int key,int value)
    {
        this.key  =key;
        this.value = value;
    }

    public void setValue(int value)
    {
        this.value = value;
    }

    public void setKey(int key)
    {
        this.key = key;
    }

    public HashList getNext()
    {
        return next;
    }

    public void setNext(HashList next)
    {
        this.next = next;
    }

    public String toString()
    {
        return "[ "+ key + " ] --->"+ value;
    }
}
```

Fig.3

```
public void setSize(int size)
{
    this.size = size;
}

public int get(int key)
{
    int hashvalue = (key % size);
    if(hashtable[hashvalue] == null) // the value doesnt have an entry
         return -1;
    else
    {
        HashList bucket = hashtable[hashvalue];
        while(bucket!= null)
        {
            if(key == bucket.getKey())
                 return bucket.getValue();

            bucket = bucket.getNext();
        }

        return -1;
    }
}

public void put(int key,int value)
{
    int hashvalue = (key % size);
    if(hashtable[hashvalue] == null) // the value doesnt have an entry
         hashtable[hashvalue] = new HashList(key,value);
    else
    {
        HashList bucket = hashtable[hashvalue];
        while(bucket!= null)
        {
            if(key == bucket.getKey())
            {
                bucket.setValue(value);
                break;
            }
            bucket = bucket.getNext();
        }

        bucket.setNext(new HashList(key,value));
    }
}
}
```

Fig.4

Lexical Attributes and Token Objects

Besides the token's category, the rest of the compiler may several pieces of information about a token in order to perform semantic analysis, code generation, and error handling. These are stored in an object instance of class Token, or in C, a struct. The fields are generally something like:

```
struct token {
   int category;
   char *text;
   int linenumber;
   int column;
   char *filename;
   union literal value;
}
```

The union literal will hold computed values of integers, real numbers, and strings.

Lexical Analysis and the Symbol Table

In many compilers, the symbol table and memory management components of the compiler interact with several phases of compilation, starting with lexical analysis.

- Efficient storage is necessary to handle large input files.
- There is a colossal amount of duplication in lexical data: variable names, strings and other literal values duplicate frequently
- What token type to use may depend on previous declarations.

The below source code illustrates how to parse and resolve variables types, scope and block. The source code is written in JAVA programming language

```
public class Types
{
    public static boolean Variables(String var)
    {
        if(var.equals("int"))
            return true;
        else if(var.equals("char"))
            return true;
        else if(var.equals("boolean"))
            return true;
        else if(var.equals("long"))
            return true;
        else if(var.equals("double"))
            return true;
        else if(var.equals("float"))
            return true;
        else if(var.equals("byte"))
            return true;
        return false;
    }
}
```

Fig.5

```
import java.io.*;
import java.util.*;

public class Driver
{
   private static Hashtable<String, Integer> sTable = new Hashtable<String ,
Integer>();
   private static Stack stack = new Stack();
   public static void main(String[] args)throws IOException
   {
       boolean iserror = false;
       BufferedReader input = new BufferedReader(new FileReader("data.txt"));
       String line = input.readLine();
       int scope=0;
       System.out.println();
       System.out.println("Variable" + " : "+ "Scope Level");
       System.out.println("-----------------------------------------------");
       while(line != null)
       {
           if(line.equals(""))
           {
               line = input.readLine().trim();
               continue;
           }

           else if(line.equals("{"))
           {
               stack.push("{");
               scope++;
           }

           else if(line.equals("}"))
           {
               if(!stack.isEmpty())
               {
                   stack.pop();
                   scope--;
               }
               else
               {
                   System.out.println("Unbalanced { } ");
                   iserror = true;
                   break;
                }

           }
           else
           {
               StringTokenizer tokens = new StringTokenizer(line);
               String type = tokens.nextToken();
               String var = tokens.nextToken();

               if(!Types.Variables(type))
               {
```

```
                    System.out.println("Wrong Type for Variable : " + var);
                    line = input.readLine().trim();
                    continue;
                }
                if(!var.endsWith(";"))
                {
                     System.out.println("Missing SemiColon for Variable : " +
var);
                     line = input.readLine().trim();
                     continue;
                }
                else
                     var = var.substring(0,var.length()-1);

                if(!sTable.containsKey(var))
                {
                    sTable.put(var,scope);
                }
                else
                {
                    int currentScope = scope;
                    int alreadyVarScope = sTable.get(var);

                    if(currentScope > alreadyVarScope)
                    {
                        System.out.println("Variable " + var + " is already
declared");
                        line = input.readLine().trim();
                        iserror=true;
                        break;
                    }
                    else
                    {
                        if(scope == 1)
                        {
                            System.out.println("Variable " + var + " is
already declared");
                            line = input.readLine().trim();
                            iserror=true;
                            break;
                        }
                        System.out.println(var +" :\t\t " + scope);
                        sTable.put(var,scope);
                    }
                }
            }

            try
            {
                line = input.readLine().trim();
             }catch(NullPointerException err)
             {
                 break;
             }
        }

        if(!stack.isEmpty()&&!iserror)
```

```
        {
            System.out.println("Unbalanced { } ");
        }

        if(!iserror)
        {
            Object[] v1 = sTable.values().toArray();
            Enumeration v2 = sTable.keys();
            for(int i=0;i<v1.length;i++)
            {
                System.out.println(v2.nextElement() + " :\t\t " + v1[i]);
            }
        }
    }
}
```

Sample Input:

```
{
char p;

      {
            int x;
            {
                  double z;
            }
      }

      {
            int x;
            {
                  double z;
                  {
                        char t;
                        char p;
                  }
            }
      }

}
```

Sample output:

```
Variable : Scope Level

-----------------------------------------------

x :          2

z :          3

Variable p is already declared
```

- **Open Addressing**

Hash tables based on open addressing is directly related to the proper choice of hash function. If the hash function is good and hash table is well-dimensioned, this will amortize the complexity of insertion, removal and lookup operations are constant.

```
#include "LinkedList.h"

class HashTable
{
private:

   LinkedList * array;

   int length;

   int hash( string Key );

public:

   HashTable( int tableLength = 100 );

   void put( Item * item );

   Item * get( string Key );

   int getLength();

   void print();

   ~HashTable();
};


##############################################
         HASH TABLE IMPLEMENTATION
##############################################


#include "HashTable.h"
HashTable::HashTable( int tableLength )
{
   if (tableLength <= 0) tableLength = 100;
   array = new LinkedList[ tableLength ];
```

```
    length = tableLength;
}

// Returns an array location for a given item key.
int HashTable::hash( string Key )
{
    int value = 0;
    for (int i = 0; i < itemKey.length(); i++)
        value += itemKey[i];
    return (itemKey.length() * value) % length;
}

// Adds an item to the Hash Table.
void HashTable::put( Item * item )
{
    int index = hash(item -> key );
    array[ index ].insertItem(item);
}


// Returns an item from the Hash Table by key.
Item * HashTable::get( string Key )
{
    int index = hash(Key);
    return array[index].getItem(Key);
}

void HashTable::print()
{
    cout << "\nHash Entries:\n";
    for (int i = 0; i < length; i++)
    {
        cout << "Bucket " << i+1 << ": ";
        array[i].printList();
    }
}

int HashTable::getLength()
{
    return length;
}

HashTable::~HashTable()
{
    delete [] array;
}
```

REFERENCES (LITERATURE CITED)

1. Implementing Programming Languages, Aarne Ranta ,February 6, 2012
2. http://arantxa.ii.uam.es/~modonnel/Compilers/04_SymbolTablesI.pdf
3. http://jcsites.juniata.edu/faculty/rhodes/lt/sytbmgmt.htm
4. http://www.csbeans.com/2015/04/how-to-program-compiler-design-symbol.html