Assignment 3

Problem Statement: Generate and populate appropriate Symbol Table.

Objective: To learn to generate symbol table

Outcome: Successful implementation of symbol table generation

Software Requirements: JDK tool kit

Hardware Requirements: 4GB RAM, 500 GB HDD

Theory:

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built in lexical and syntax analysis phases.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- It is used by compiler to achieve compile time efficiency.
- It is used by various phases of compiler as follows :-
    1. **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
    2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
    3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
    4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
    5. **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
    6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

The IR produced by the front end consists of two components:

1. Tables of information

2. An INTERMEDIATE CODE (IC) which is a description of the source program.

TABLES

Tables contain the information obtained during different analyses of SP. The most important table is the symbol table which contains information concerning all identifiers used in the SP. The symbol table is built during lexical analysis. Semantic analysis adds information concerning symbol attributes while processing declaration statements. It may also add new names designating temporary results.

INTERMEDIATE CODE (I C)

The IC is a sequence of IC units; each IC unit representing the meaning of one action in SP. IC units may contain references to the information in various tables.

Example

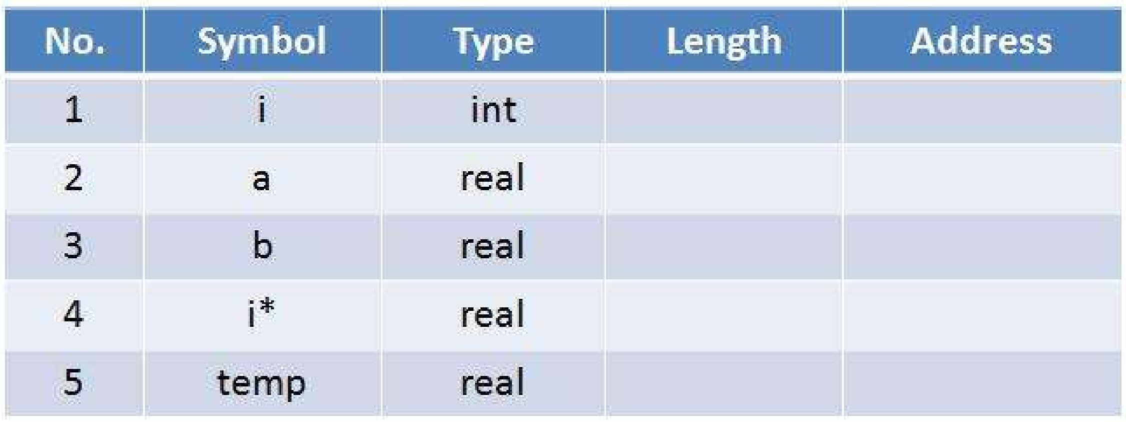
Figure shows the IR produced by the analysis phase for the program

i:integer;

a,b: real;

a := b+i;

Symbol Table:



No.	Symbol	Type	Length	Address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

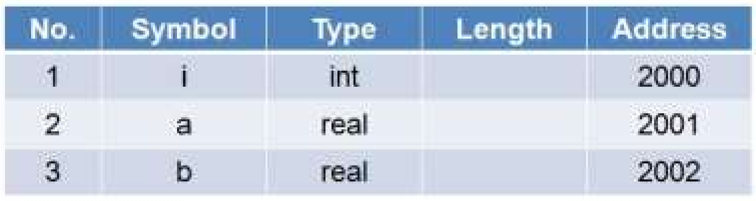
The symbol table contains information concerning the identifiers and their types.

The Back End

The back end performs memory allocation and code generation

Memory allocation

Memory allocation is a simple task given the presence of the symbol table. The memory requirement of an identifier is computed from its type, length and dimen-sionality, and memory is allocated to it. The address of the memory area is entered in the symbol table.
Example
After memory allocation, the symbol table looks as shown in Figure 3. The entries for i* and temp are not shown because memory allocation is not needed for these id’s.



No.	Symbol	Type	Length	Address
1	i	int		2000
2	a	real		2001
3	b	real		2002

Certain decisions have to precede memory allocation, for example, whether i* and temp should be allocated memory. These decisions are taken in the preparatory steps of code generation.

Code generation

Code generation uses knowledge of the target architecture, viz. knowledge of instructions and
addressing modes in the target computer, to select the appropriate instructions.

Example

For the sequence of actions for the assignment statement
a :* b+i;

(i) Convert i to real, giving i*.
(ii) Add i* to b, giving temp,
(iii) Store temp in a.
the synthesis phase may decide to hold the values of i* and temp in machine registers and may generate the assembly code:

```
CONV_R
AREG, I
ADD_R
AREG, B
MOVEM
AREG, A
```

where CONV_R converts the value of I into the real representation and leaves the result in AREG. ADD_R performs the addition in real mode and MOVEM puts the result into the memory area allocated to A.

Conclusion:

Thus we’ve implemented program to generate and populate symbol table.

Code:

```
import java.io.*;
class P1
{
public static void main(String ar[])throws IOException
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
int i;
String a[][]={{"","START","101",""},
{"","MOVER","BREG","ONE"},
{"AGAIN","MULT","BREG","TERM"},
{"","MOVER","CREG","TERM"},
{"","ADD","CREG","N"},
{"","MOVEM","CREG","TERM"},
{"N","DS","2",""},
{"RESULT","DS","2",""},
{"ONE","DC","1",""},
{"TERM","DS","1",""},
{"","END","",""}};
int lc=Integer.parseInt(a[0][2]);
String st[][]=new String[5][2];
int cnt=0,l;
for (i=1;i<11;i++)
{
```

```
if (a[i][0]!="")
{
st [cnt][0]=a[i][0];
st[cnt][1]=Integer.toString(lc);
cnt++;
if(a[i][1]=="DS")
{
int d=Integer.parseInt(a[i][2]);
lc=lc+d;
}
else
{
lc++;
}
}
else
{
lc++;
}
}
System.out.print("***SYMBOL TABLE****\n");
System.out.println("_______________________");
for(i=0;i<5;i++)
{
for(cnt=0;cnt<2;cnt++)
{
System.out.print(st[i][cnt]+"\t");
}
System.out.println();}
String
inst[]={"STOP","ADD","SUB","MULT","MOVER","MOVEM","COMP","BC","DIV","R
EAD","PRINT"};
String reg[]={"NULL","AREG","BREG","CREG","DREG"};
int op[][]=new int[12][3];
int j,k,p=1,cnt1=0;
for(i=1;i<11;i++)
{
for(j=0;j<11;j++)
{
if(a[i][1].equalsIgnoreCase(inst[j]))
{
op[cnt1][0]=j;
}
else
if(a[i][1].equalsIgnoreCase("DS"))
{
p=Integer.parseInt(a[i][2]);
}
else if(a[i][1].equalsIgnoreCase("DC"))
{
op[cnt1][2]=Integer.parseInt(a[i][2]);
}
}
for(k=0;k<5;k++)
{
if(a[i][2].equalsIgnoreCase(reg[k]))
{
op[cnt1][1]=k;
}
}
for(l=0;l<5;l++)
{
if(a[i][3].equalsIgnoreCase(st[l][0]))
{
int mn=Integer.parseInt(st[l][1]);
op[cnt1][2]=mn;
}
}
cnt1=cnt1+p;
}
System.out.println("\n *****OUTPUT*****\n");
System.out.println("**********MOT TABLE**********");
int dlc=Integer.parseInt(a[0][2]);
for(i=0;i<12;i++)
{
System.out.print(dlc+++"\t");
for(j=0;j<3;j++)
{
System.out.print(" "+op[i][j]+" ");
}
System.out.println();
}
System.out.println("");}
}

/*
```

OUTPUT-

```
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment3$ javac
P1.java
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment3$ java P1
***SYMBOL TABLE****
______________________
AGAIN       102
N       106
```

```
RESULT      108
ONE   110
TERM        111

 *****OUTPUT*****

**********MOT TABLE**********
101     4  2  110
102     3  2  111
103     4  3  111
104     1  3  106
105     5  3  111
106     0  0  0
107     0  0  0
108     0  0  0
109     0  0  0
110     0  0  1
111     0  0  0
112     0  0  0

(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment3$

*/
```