SYMBOL TABLE ROUTINES

Choose an organization for your symbol table -- data structures for the identifier names and records -- and write the FIND and INSERT routines. Put these routines into a dummy main program to prove that they work.

Submit:
(1) Symbol table routines
(2) Test runs (input and output)

Grading:
Is the program correct? Is the program elegant? Well written? Is the program well tested?

Important routines:
FIND_in_CURRENT_SCOPE-1, FIND_IN_ALL_SCOPES-2, INSERT, DISPLAY

Driver program: to test FIND (both versions), INSERT, DISPLAY, you need a driver (main) program that can: (1) open and close scopes, (2) check if a name is already in the symbol table (ST), and (3) add it into the ST if not already there.

Requirements:
-- identifiers must be stored without truncation, but also without wasting space. This probably means string allocation routines (although in some languages, a string table must be used).

-- an important question is whether the high level language is case sensitive (is “a” the same as “A”?). For now, assume the language compiled IS case sensitive.

-- you may use any library or utility functions available (Java has a HashTable with findKey and put which are almost find and insert).

-- symbol table entries must be expandable. For this program, there is no information to store about each identifier name (unless you choose to store a "block number" to distinguish between two different occurrences of the same name).

-- the table must be able to handle multiple entries of the same name. This can be done by (1) having separate symbol tables for each scope (OPEN CLOSE) or (2) storing some means of differentiating between multiple copies of the same identifier, such as the "block number" of the declaration.

-- FIND should be an efficient routine; it will be called often. INSERT should be at least relatively efficient. Hash tables and balanced binary trees are the usual choices for a symbol table organization.

-- FIND must return more than a simple Boolean; if found, it must return some means of retrieving the information in the symbol table (pointer to or copy of the record).

-- FIND and INSERT must be separate routines; neither may be nested inside the other. Probably neither should call the other

Test your program by creating a dummy file of identifier names with some method of indicating that a block starts or stops. One dummy file is below, but it isn’t sufficient to test your program.

The OPEN and CLOSE will not be necessary in the final compiler, and you don't need to check that there are equal numbers of each, or that they occur in a legal order.

NO ERROR CHECKING IS NEEDED. You need FIND & INSERT routines that work on correct strings, but you don’t have to worry about what a correct string may be in your high level language.

You do need to DISPLAY the symbol table in some way (at one or more points in reading file) that shows that all entries were properly added.

See Chap. 8 (Parsons) for further information.   Be sure that your program follows good programming practices, is logical, easy to read, and thoroughly tested! And remember, the program should exhibit a clean, simple, easy-to-understand design.

ONE SAMPLE INPUT FILE:

The inputs below assume that scopes are opened with ‘{’ and closed with ‘}’.

```
 { a ab ba abbab ab  ba      //What opens a new scope?  Don’t really care. OPEN?
{ ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
}
{ a ab ba AB BA AB BA ab ba
{ a ab AB }
  a ab ba AB BA AABB
}
  a ba cd dc
}
```

SYMBOL TABLE DUMP (order of names not important)

```
Scope 1:  a ab ba abbab cd dc
Scope 2:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
Scope 3:  a ab ba AB BA AABB
Scope 4:  a ab AB
```

SECOND SAMPLE FILE

```
{ a  A  b  B
   {   }
   { a  A}
   { b  B  c}
   {  }
 a  A  c  C  }
```

Symbol Table Questions:

1.   Chained hash table, balanced binary tree, library hash data structure, or?
2.   For new scopes, will there be a new hash table or a block identifier?
3.   If one hash table per scope, what will you do with a table when you leave a scope? Put on Old_ST_Stack or? Do you want block identifiers just to make printing entire symbol table easier (at end of program)?
4.   If block identifiers, how to know what is the current block? One way: have a small (12 element array of integer) stack of current blocks – on entering a block, increment block_number and push the new value. On exiting a block, pop the top of stack (never never decrement block_number).

5. How is the best way to store identifier names without truncation and without wasting space? It's fine to have one long array as a buffer to read in a name from a file, but not OK to have long static arrays in each symbol table entry.
6. What error checking needs done? Essentially none, since the only routines that will be kept are FIND (two versions), INSERT, and DISPLAY.
7. What is the input source? Probably easiest way is from a file.
8. What languages will I use for this, and for the rest of the compiler? C++, Java, or? How do I want to organize the program? What classes will I want?

EXAMPLE 1:

```
OPEN a a a b b b  OPEN a b c  CLOSE OPEN a OPEN a CLOSE a a CLOSE a a b c CLOSE
```

EXAMPLE 2:

```
OPEN   //  block number = 0
aa1122*&^         a      a     a
***   ***  &&&&     // any string is a legal id as far as this program is concerned
OPEN
a          b          B   B   a   B
CLOSE
a
b
CLOSE
```

General outline of your program:

```
{  initialize
   while not at end of file
        get next string
        if string == OPEN   start a block
            else if string == CLOSE   end a block  (and print the block you are closing?)
            else   // string looks like an id
                { curr =  find_in_current (string, current block number)
                  if not curr   insert (string, current block number)
                  if not curr   find_in_all (string)   // to check if anywhere else in table
  print the whole symbol table
}
```

Start block
  a. sym tables: create a new ST and push it onto active symbol table stack
  b. block nums: increment BlockNum, push BlockNum onto stack of active block numbers

End block
  a. sym tables: pop off the ST (and push onto old_symbol_table_stack?)
  b. block nums: decrement top of stack for stack of active block numbers