The Symbol Table

The symbol table is used to store essential information about every symbol contained within the program.

This includes:
- Keywords
- Operators
- Variables
- Constants
- Data Types
- Functions
- Procedures
- Literals
Contents of the Symbol Table

The symbol table will contain the following types of information for the input strings in a source program:

- The lexeme (*input string*) itself
- Corresponding token
- Its semantic component (e.g., variable, operator, constant, functions, procedure, etc.)
- Data type
- Pointers to other entries (when necessary)

The interaction between the Symbol Table and the phases of a compiler

Virtually every phase of the compiler will use the symbol table:

- The initialization phase will place keywords, operators, and standard identifiers in it.
- The scanner will place user-defined identifiers and literals in it and will return the corresponding token.
- The parser uses these token to create the parse tree, the product of the syntactic analysis of the program.
- The semantic action routines place data type in its entries and uses this information in performing basic type-checking.
The interaction between the Symbol Table and the phases of a compiler (continued)

- The intermediate code generation phase uses pointers to entries in the symbol table in creating the intermediate representation of the program.
- The object code generation phase uses pointers to entries in the symbol table in allocating storage for its variables and constants, as well as to store the addresses of its procedures and functions.

Looking things up in the Symbol Table

Example - How would the scanner read a Pascal program header:

```pascal
int main(void)
```

After reading the first 4 characters, it assembles the lexeme `int main(void)`

Pushing back the blank into the input stream. Since `int` is a reserved word, it is already in the symbol table, which returns the token `int`
Looking things up in the Symbol Table (continued)

Next, the scanner assembles the lexeme

\[ \text{'m'} \text{'a'} \text{'i'} \text{'n'} \]

pushing back the \{ following it

The Symbol Table manager can’t find it when it looks it up. Therefore, it installs the \text{main} in the Symbol Table with the token \text{identifier}.
During semantic analysis, it will be given the semantic property of \text{int}.

Data stored in the Symbol Table

Let’s take a look at the kinds of information stored in the symbol table for different types of elements within a program.

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
<th>Symbol</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class</td>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>while</td>
<td>TokWhile</td>
<td>Keyword</td>
<td>None</td>
</tr>
<tr>
<td>+</td>
<td>TokPlus</td>
<td>Operator</td>
<td>None</td>
</tr>
<tr>
<td>x</td>
<td>TokIdentifier</td>
<td>Variable</td>
<td>Integer</td>
</tr>
<tr>
<td>8</td>
<td>TokConstant</td>
<td>Literal</td>
<td>Integer</td>
</tr>
</tbody>
</table>
The basic operations of the Symbol Table

• There are several operations that are performed on the Symbol Table, the most important being:
  • Adding symbols - The reserved words, standard identifiers and operators are placed in the Symbol Table during its initialization.
    – New lexemes are added when the scanner encounters them, and they are assigned a token class.
    – Similarly, the semantic analyzer adds the appropriate properties and attributes that belong to the lexeme.

The basic operations of the Symbol Table (continued)

• Deleting symbols - When the compiler is finished with a given scope of the program, all the symbols belonging to that scope must be effectively removed before beginning to process a different scope of the program.
• The data regarding these variables may be hidden from the compiler’s view or dumped into a temporary file.
• Searching for symbols - Looking up a lexeme in the symbol table and retrieving any and all of the properties belonging to that symbol.
Possible organization methods

There are several different ways to organize the symbol table:

The most obvious is to organize it as an **array of records**, but it would either require a linear search or constant sorting.

<table>
<thead>
<tr>
<th>begin</th>
<th>tokbegin</th>
<th>stkeyword</th>
<th>dtnone</th>
<th>...</th>
<th>global</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>tokcall</td>
<td>stkeyword</td>
<td>dtnone</td>
<td>...</td>
<td>global</td>
</tr>
<tr>
<td>called</td>
<td>tokidentifier</td>
<td>stvariable</td>
<td>dtreal</td>
<td>...</td>
<td>sample</td>
</tr>
<tr>
<td>procedure</td>
<td>tokprocedure</td>
<td>stkeyword</td>
<td>dtnone</td>
<td>...</td>
<td>global</td>
</tr>
<tr>
<td>xmin</td>
<td>tokidentifier</td>
<td>tokconstant</td>
<td>dtinteger</td>
<td>...</td>
<td>sample</td>
</tr>
</tbody>
</table>

Possible organization methods (continued)

We could use an **unordered list**. This would make efficient use of memory and it would be quick and easy to install names, but retrieval would be unacceptably slow.

An ordered list would make storing and retrieving names faster, but not by that much.
Possible organization methods (continued)

*Binary search trees* would provide some compromises on these structures above, but only when the tree is balanced. This rarely happens on its own and takes time for balancing to be done by the insertion routine.

```
+----------------+ +----------------+ +----------------+
| Sum            | I              | Average         |
+----------------+ +----------------+ +----------------+
| I              | Average        | Sum             |
+----------------+ +----------------+ +----------------+
| Average        | Balanced       | Unbalanced      |
+----------------+ +----------------+ +----------------+
| Unbalanced     | Sum            | Average         |
+----------------+ +----------------+ +----------------+
```

Possible organization methods (continued)

*Hash trees* are the most practical choice.

```
0 1 2 3 4 5 6 7 8 9
+----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+#
| sum            | i              | average         | x              | averg1          | y              | z              | h              | zzz             | sum            |
+----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+ +----------------+#
```

Organization of the Symbol Table

We need a method for storing the lexemes in an efficient manner. Using an array of strings is not efficient and would limit the length of key words and identifiers.

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>l</th>
<th>l</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>e</td>
<td>c</td>
<td>l</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>v</td>
<td>e</td>
<td>r</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FORTRAN and other earlier programming languages restricted identifiers to a specific length for exactly this reason. (FORTRAN limited identifiers to 6 letters; C originally ignored anything after the first 8 letters.)

The string table and name table

String table

```
begin call declare
```

Name table

<table>
<thead>
<tr>
<th>Start position</th>
<th>Length</th>
<th>Pointer to attribute entry</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
Hash Functions

- The hash function takes the lexeme and produces a nonunique value from it which is a starting point in a list of entries, one of which is the one that we seek.
- We want our hash value to produce as few hash collisions as possible.
- Our first temptation would be to use a hash function such as:
  \[ \text{Sum(ASCII(Lexeme)) MOD SomeValue} \]
This is not a good hash function because \textit{cat} and \textit{act} would have the same hash value
### Hash Function

```c
int symboltable::hashcode(char string[])
{
    int       i, numshifts, startchar, length;
    unsigned  code;

    length = strlen(string);
    numshifts = (int) min(length, (8*sizeof(int)-8));
    startchar = ((length-numshifts) % 2);
    code = 0;

    for (i = startchar;
         i <= startchar + numshifts - 1;  i++)
        code = (code << 1) + string[i];
    return(code % hashtablesize);
}
```

### The attribute table

- The attribute table can be considered to be the central structure of the symbol table where the attributes of each lexeme are stored.
- The attribute table is an array of records in which we store data relevant to the symbol:
  - Symbol Type - Data Type
  - Token class - Value
  - Pointer back to the Name Table
The attribute table (continued)

- The data in the attribute table is separated from the name table because the same name can be used in different contexts in different scopes of the program. For this reason, it is in our best interests to use a pointer to the attribute table entry directly in most cases.

- If a name that already is used in outer scope is used again, we will create another attribute table entry and link it to the outer scope’s entry.
Auxiliary Table

- If there are several values to store or more than several pointers to other entries in the symbol table, it may be necessary to add an auxiliary table to hold these values.
- The attribute table is linked to the auxiliary table by holding the starting position and number of entries in the auxiliary table.
- This would also allow us to implement structures such as array and records in a fairly straightforward fashion.

The JASON programming language

JASON (Just Another Simple Original Notion) is a small programming language designed to illustrate the major concepts in programming language design and compiler construction.

JASON has two data types (Integers and Reals), Procedures (all parameters are passed by reference), and a structured syntax that is somewhat ALGOL-like.
An example of a JASON program

PROGRAM ProgName;  { This is a comment }
DECLARE
  INTEGER x, y, z;
PROCEDURE Test(INTEGER ed; INTEGER try;
  INTEGER can);
DECLARE
  INTEGER a, b;
BEGIN
  READ a;
END;
BEGIN  { main program }
  READ X;
  CALL Test(x, y, z);
  SET x = y/8 + x*y - 6;
  UNTIL x = 9  DO
    IF x < 3 THEN
      SET y = x*x*y
    ELSE
      SET y = 8;
    SET x = 77
    ENDIF;
  ENDUNTIL;
  WRITE x
END.
Enumerated Types For Tokens

// The tokens
enum tokentype {tokbegin, tokcall, tokdeclare, tokdo,
tokelse, tokend, tokendif, tokenduntil,
tokendwhile, tokif, tokinteger,
tokparameters, tokprocedure, tokprogram,
tokread, tokreal, tokset, tokthen,
tokuntil, tokwhile, tokwrite, tokstar,
tokplus, tokminus, tokslash, tokequals,
toksemicolon, tokcomma, tokperiod,
tokgreater, tokless, toknotequal,
tokopenparen, tokcloseparen, tokfloat,
tokidentifier, tokconstant, tokerror,
tokeof, tokunknown
};

Enumerated Types For Symbol And Data Types

// The semantic types, i.e, keywords, procedures,
// variables, constants
enum semanticstype {stunknown, stkeyword, stprogram,
stparameter, stvariable, sttempvar,
stconstant, stenum, ststruct,
stunion, stprocedure, stfunction,
stlabel, stliteral, stoperator
};

// The data types, i.e, real and integer
enum datatype {dtunknown, dtnone, dtprogram,
dtprocedure, dtinteger, dtreal
};
Type Definitions For The Name Table

// The structure for name table entries, i.e, a
// starting point in a long array, a pointer to the
// entry in the attribute table and the next lexeme
// with the same hash value.
typedef struct {
    int       strstart;
    int       symtabptr;
    int       nextname;
} nametabtype;

Type Definitions For The Attribute Table

typedef    struct { 
    semantictype       smtype;
    tokentype          tok_class;
    datatype           dataclass;
    int                owningprocedure;
    int                thisname;
    int                outerscope, scopenext;
    valrec             value;
    char               label[labelsize];  
} attribtabtype;
Declaration For the Class **Symbol**

// The definition of the class SymbolTable
class symboltable {
public:
    symboltable(void);
    void dump(void);
    // Functions that set up the symbol table
    bool installname(char string[], int &tabindex);
    bool ispresent(char string[], int &code,
                   int &nameindex);
    int installattrib(int nameindex);
    void setattrib(int tabindex,
                   semantictype symbol, tokentype token);
    void installdatatype(int tabindex,
                          semantictype stype, datatype dclass);

    // Functions that handle name scoping
    int openscope(int tabindex);
    void closescope(void);
    // Set and return the attribute table index for the
    // current procedure
    void setproc(int thisproc, int tabindex);
    int getproc(int tabindex);
    // Functions that set and return the constant value
    void setvalue(int tabindex, float val);
    void setvalue(int tabindex, int val);
    inline float getrvalue(int tabindex)
    {return(attribtable[tabindex].value.val.rval);}
    inline int getivalue(int tabindex)
    {return(attribtable[tabindex].value.val.ival);}
    void getlabel(int tabindex, char varlabel[]);
    int labelscope(int procindex);
// Sets up the label for the procedures's parameters
void paramlabel(int tabindex, char label[],
    int &bytecount);

// Print the lexeme and token in user-friendly
// formats
void printlexeme(int i);
void printtoken(int i);

// Returns basic information about symbols
tokentype gettok_class(int tabindex);
inline enum datatype getdatatype(int tabindex)
    {return(attribtable[tabindex].dataclass);} 
inline semantictype getsmclass(int tabindex)
    {return(attribtable[tabindex].smitype);} 
inline bool isvalidtype(int tabindex)
    {return(attribtable[tabindex].dataclass == dtinteger 
        ||attribtable[tabindex].dataclass == dtreal);} 

// Returns the size of the attribute table
inline int tablesiz(void){ return(attribtablen); } 

private:
// Initializes the procedure stack's entry
procstackitem initprocentry(int tabindex);
int hashcode(char string[]);
void LexemeInCaps(int tabindex);

// Creates a label for the object code
void makelabel(int tabindex, char *label);

inline int min(int a, int b)
    {return ((a < b)? a : b);} 
char stringtable[stringtablesize];
nametabtype nametable[nametablesize];
attribtabtype attribtable[attribtablesize];
int hashtable[hashtablesize];
int strtablen, namtablen, attribtablen;
procstackitem thisproc;
stack <procstackitem> ps;
Initializing the Symbol Table

- Before we can start scanning the program, we must initialize the symbol table.
- Initialization involves installing all reserved words, standard identifiers, and operators.
- In production compilers, the symbol table has these stored in memory before execution.

The Symbol Table Constructor

```cpp
symboltable::symboltable(void)
{
    int i, nameindex;

    thisproc = initprocentry(-1);

    // Initialize the hash table, the name table's next field and the attribute table's fields as -1.
    for (i = 0; i < hashtablesize; i++)
        hashtable[i] = -1;

    for (i = 0; i < nametablesize; i++)
        nametable[i].nextname = -1;
}```
for (i = 0; i < attribtablesize; i++) {
    attribtable[i].smtype = stunknown;
    attribtable[i].tok_class = tokunknown;
    attribtable[i].thisname = -1;
    attribtable[i].value.tag = tint;
    attribtable[i].value.val.ival = 0;
    attribtable[i].owningprocedure = -1;
    attribtable[i].outerscope = -1;
    attribtable[i].scopenext = -1;
    attribtable[i].label[0] = '\0';
}

// Install the keywords in the name table and
// set their attributes to keyword
for (i = 0; i < numkeywords; i++) {
    installname(keystring[i], nameindex);
    setattrib(nameindex, stkeyword, (tokentype)i);
}

// Install the operators in the name table and
// set their attributes to operator
for (i = numkeywords; i < numtokens; i++) {
    installname(keystring[i], nameindex);
    setattrib(nameindex, stoperator, (tokentype)i);
}

installname(keystring[i], nameindex);
setattrib(nameindex, stfunction, (tokentype)i);
installdatatype(nameindex, stfunction, dtreal);

cout << "All initialized" << endl;
Searching for lexemes and installing them

• After the scanner assembles the lexeme, it must determine if it is already in the symbol table. If so and if it is declared within the current scope, its token is already determined.
• If the lexeme is not already in the symbol table, then it must be installed and assigned a token (usually **Identifier**). This also requires that we create an entry in the attribute table.
• If the lexeme is already in the symbol table but does not belong to the current scope, a new scope must be opened for it.

The **InstallName** function

```c
bool symboltable::installname(char string[],
                               int &stabindex)
{
    int i, code, length, nameindex;
    // Use the function ispresent to see if the token
    // string is in the table. If so, return a pointer
    // to its attribute table entry.
    length = strlen(string);
    if (ispresent(string, code, nameindex)) {
        if (nametable[nameindex].symtabptr == -1) {
            tabindex = installattrib(nameindex);
            return(false);
        }
    } else {
        tabindex = nametable[nameindex].symtabptr;
        return(true);
    }
}
```
// If not create entries in the name table, copy the
// name into the string table and create a hash table
// entry(linking it to its previous entry if
// necessary) and create an entry in the attribute
// table with the bare essentials.
  nametable[nameindex = namtablen++].strstart =
    strtablen;

  for (i = 0;  i < length;  i++)
    stringtable[strtablen++] = string[i];
  stringtable[strtablen++] = '\0';
  nametable[nameindex].nextname = hashtable[code];
  hashtable[code] = nameindex;
  tabindex = installattrib(nameindex);
  return(false);
}

The IsPresent function

// IsPresent() - After finding the hash value, it
// traces through the hash list, link
// by link looking to see if the
// current token string is there.
//
// bool symboltable::ispresent(char string[], int &code,
// int &nameindex)
{
  bool found = false;
  int oldnameindex, k;

  // Initialize the old name's index to -1;
  // it may not be there
  oldnameindex = -1;

  // Find the hash value
code = hashcode(string);
// Starting with the entry in the hash table, trace
// through the name table's link list for that hash
// value.
for (nameindex = hashtable[code];
    !found && nameindex != -1;
    oldnameindex = nameindex,
    nameindex = nametable[nameindex].nextname)
{
    k = nametable[nameindex].strstart;
    found = !strcmp(string, stringtable+k);
}

// If it's there, we actually went right past it.
if (found)
    nameindex = oldnameindex;
return(found);

Creating an attribute table entry and setting attributes

- Once the lexeme is installed in the name table, an entry
  must be created in the attribute table for it.
- The function *InstallAttrib* will create the entry, initialize a
  pointer back to the name table, and set the symbol type and
  the data type as unknown.
- The function *SetAttrib* will set the symbol type, token class
  and the data type as well as link the entry to any others for
  that scope.
The function \textit{InstallAttrib}

```c
int symboltable::installattrib(int nameindex)
{
    int tabindex;
    tabindex = nametable[nameindex].symtabptr
    = attribtablelen++;
    attribtable[tabindex].thisname = nameindex;
    attribtable[tabindex].smttype = stunknown;
    attribtable[tabindex].dataclass = dtunknown;

    return(tabindex);
}
```

The function \textit{SetAttrib}

```c
void symboltable::setattrib(int tabindex,
    semantictype symbol, tokentype token)
{
    attribtable[tabindex].smttype = symbol;
    attribtable[tabindex].tok_class = token;

    if (attribtable[tabindex].smttype == stkeyword
            || attribtable[tabindex].smttype == stoperator)
        attribtable[tabindex].dataclass = dtnone;
    else
        attribtable[tabindex].dataclass = dtunknown;
```
if (gettok_class(tabindex) == tokidentifier
    & thisproc.proc != -1)
    if (thisproc.sstart == -1)
        {
            thisproc.sstart = tabindex;
            thisproc.snex = tabindex;
        }
    else
        {
            attribtable[thisproc.snex].scopenext = tabindex;
            thisproc.snex = tabindex;
        }
}

Installing data type

• This is done by the semantic analyzer during the processing of declarations.
• Until this point, identifiers have their data type listed as unknown.
• The same function, InstallDataType, installs the the correct symbol type as well as the correct data.

void symboltable::installdatatype(int tabindex,
    semantictype stype, datatype dclass)
{
    attribtable[tabindex].smtyp = stype;
    attribtable[tabindex].dataclass = dclass;
}
Scope rules

- Block-structured languages, such as ALGOL, PL/I, Pascal and C, have to allow for the fact that the same identifier may mean completely different things in different scopes.
- Strictly speaking, a block is a program structure in which there can be declarations. E.g., Although it is unusual to place declarations in C blocks (excluding the main block of a function), it is perfectly legal.
- Because such languages resolve name conflicts using a most closely nested scope rule, languages with static scoping rules must have a mechanism for temporarily hiding a variable in an outer scope.
- In implementing a symbol table, this mean either:
  - Separate symbol tables for each scope or
  - identifying the scope to which each identifier belongs.

Keeping track of scope

We create a linked list of identifiers belonging within the scope. This will simply the process of closing the scope when we are finished.
Keeping track of scope

We also keep a linked list of identifiers sharing the same name pointing from the most closely nested scope outward.

Additionally, each identifier is marked with a pointer to the symbol table entry of the procedure which “owns” it.

Opening and closing the scope

Opening the scope involves:
• Creating a new entry in the attribute table.
• Readjusting the name table pointer to point to the new entry (and vice versa).
• Linking the new entry to the other entries in the new scope as well as to the other entries sharing its name.

Closing the scope involves:
• Readjusting the pointer in the name table to point to the next scope outwards for that identifier and possibly,
• writing the data on disk and clearing the entry.
The function `OpenScope`

```c
// OpenScope() - Open a new scope for this identifier
int symboltable::openscope(int tabindex)
{
    int newtabindex, nameindex;

    // Get the index in the name table
    nameindex = attribtable[tabindex].thisname;
    // Create a new attribute table entry and
    // initialize its information
    newtabindex = installattrib(nameindex);
    setattrib(newtabindex, stunknown, tokidentifier);
    // Have this entry point to the outer scope's entry
    attribtable[newtabindex].outerscope = tabindex;
    return(newtabindex);
}
```

The function `CloseScope`

```c
// CloseScope() - Close the scope for ALL the identifiers for the scope
void symboltable::closescope(void)
{
    int nmptr, symptr;

    // Start at the first identifier that belongs
    // to the procedure and for each identifier
    for (symptr = thisproc.sstart;  symptr != -1;
        symptr = attribtable[symptr].scopenext) {
        // Have it point to the outer scope's
        // attribute table entry
        nmptr = attribtable[symptr].thisname;
        nametable[nmptr].symtabptr =
            attribtable[symptr].outerscope;
    }
}
```
Miscellaneous functions

These are used primarily to set values within the attribute table or obtain or print these values. These include:
- gettok_class
- getproc
- LexemeInCaps
- printlexeme
- printtoken
- setproc
- setvalue

// GetTok_Class()  Returns the token class for the symbol
tokentype symboltable::gettok_class(int tabindex) {
    return(attribtable[tabindex].tok_class);
}

// PrintToken() - Print the token class's name given the token class.
void symboltable::printtoken(int i)
{
    cout << tokclstring[(int)gettok_class(i)];
}
// GetProc() - Returns the identifier's owning procedure
int symboltable::getproc(int tabindex)
{
    return(attribtable[tabindex].owningprocedure);
}

// SetProc() - Set the identifier's owning procedure
void symboltable::setproc(int thisproc, int tabindex)
{
    attribtable[tabindex].owningprocedure = thisproc;
}

// PrintLexeme() - Print the lexeme for a given token
void symboltable::printlexeme(int tabindex)
{
    int i, j;
    char*s;

    i = attribtable[tabindex].thisname;

    j = nametable[i].strstart;
    s = stringtable + j;
    cout << s ;
}
// LexemeInCaps() - Print the lexeme in capital letters
//                  This makes it more distinctive
void symboltable::LexemeInCaps(int tabindex)
{
    int i, j;

    // Get the index within the string table
    // where the lexeme starts
    i = attribtable[tabindex].thisname;

    // Until you encounter the ending null byte,
    // Print the character in upper case.
    for (j = nametable[i].strstart;
         stringtable[j] != '\0';  j++)
        cout << toupper(stringtable[j]);
}

// SetValue() - Set the value for a real identifier
void symboltable::setvalue(int tabindex, float val)
{
    attribtable[tabindex].value.tag = treal;
    attribtable[tabindex].value.val.rval = val;
}

// SetValue() - Set the value for an integer
// identifier
void symboltable::setvalue(int tabindex, int val)
{
    attribtable[tabindex].value.tag = tint;
    attribtable[tabindex].value.val.ival = val;
}
Procedure stack

- This is necessary because we need to be able to handle procedures within procedures within procedures.
- A stack provides the LIFO (last-in-first-out) structure that we need.