Symbol Table

ASU Textbook Chapter 7.6, 6.5 and 6.3

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
Symbol table: A data structure used by a compiler to keep track of semantics of names.

- Data type.
- When is used: scope.

▷ The effective context where a name is valid.

- Where it is stored: storage address.

Operations:
- Search: whether a name has been used.
- Insert: add a name.
- Delete: remove a name when its scope is closed.
Some possible implementations

- **Unordered list:**
  - for a very small set of variables;
  - coding is easy, but performance is bad for large number of variables.

- **Ordered linear list:**
  - use binary search;
  - insertion and deletion are expensive;
  - coding is relatively easy.

- **Binary search tree:**
  - $O(\log n)$ time per operation (search, insert or delete) for $n$ variables;
  - coding is relatively difficult.

- **Hash table:**
  - most commonly used;
  - very efficient provided the memory space is adequately larger than the number of variables;
  - performance maybe bad if unlucky or the table is saturated;
  - coding is not too difficult.
Hash table

- **Hash function** \( h(n) \): returns a value from \( 0, \ldots, m - 1 \), where \( n \) is the input name and \( m \) is the hash table size.
  - Uniformly and randomly.
- **Many possible good designs.**
  - Add up the integer values of characters in a name and then take the remainder of it divided by \( m \).
  - Add up a linear combination of integer values of characters in a name, and then take the remainder of it divided by \( m \).

- **Resolving collisions:**
  - Linear resolution: try \( (h(n) + 1) \mod m \), where \( m \) is a large prime number, and then \( (h(n) + 2) \mod m \), \ldots, \( (h(n) + i) \mod m \).
  - Chaining: most popular.
    - Keep a chain on the items with the same hash value.
    - Open hashing.
  - Quadratic-rehashing:
    - try \( (h(n) + 1^2) \mod m \), and then
    - try \( (h(n) + 2^2) \mod m \), \ldots,
    - try \( (h(n) + i^2) \mod m \).
Performance of hash table

- Performance issues on using different collision resolution schemes.
- Hash table size must be adequately larger than the maximum number of possible entries.
- Frequently used variables should be distinct.
  - Keywords or reserved words.
  - Short names, e.g., $i$, $j$ and $k$.
  - Frequently used identifiers, e.g., $main$.
- Uniformly distributed.
Contents in a symbol table

- Possible entries in a symbol table:
  - Name: a string.
  - Attribute:
    - Reserved word
    - Variable name
    - Type name
    - Procedure name
    - Constant name
    - ...
  - Data type.
  - Storage allocation, size, ...
  - Scope information: where and when it can be used.
  - ...

How names are stored

- **Fixed-length name:** allocate a fixed space for each name allocated.
  - Too little: names must be short.
  - Too much: waste a lot of spaces.

<table>
<thead>
<tr>
<th>NAME</th>
<th>ATTRIBUTES</th>
<th>STORAGE ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>readarray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Variable-length name:**
  - A string of space is used to store all names.
  - For each name, store the length and starting index of each name.

<table>
<thead>
<tr>
<th>NAME</th>
<th>ATTRIBUTES</th>
<th>STORAGE ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>length</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Compiler notes #5, 20060512, Tsan-sheng Hsu
Handling block structures

```c
main() /* C code */
{ /* open a new scope */
   int H,A,L; /* parse point A */
   ...
   { /* open another new scope */
      float x,y,H; /* parse point B */
      ...
      /* x and y can only be used here */
      /* H used here is float */
      ...
   } /* close an old scope */
   ...
   /* H used here is integer */
   ...
   { char A,C,M; /* parse point C */
     ...
   }
}
```

- Nested blocks mean nested scopes.
- Two major ways for implementation:
  - Approach 1: multiple symbol tables in one stack.
  - Approach 2: one symbol table with chaining.
Multiple symbol tables in one stack

- An individual symbol table for each scope.
  - Use a stack to maintain the current scope.
  - Search top of stack first.
  - If not found, search the next one in the stack.
  - Use the first one matched.
- Note: a popped scope can be destroyed in a one-pass compiler, but it must be saved in a multi-pass compiler.

```c
main()
{
    /* open a new scope */
    int H,A,L; /* parse point A */
    ...
    {
        /* open another new scope */
        float x,y,H; /* parse point B */
        ...
        /* x and y can only be used here */
        /* H used here is float */
        ...
    } /* close an old scope */
    ...
    /* H used here is integer */
    ...
    { char A,C,M; /* parse point C */
        ...
    }
}
```
Pros and cons for multiple symbol tables

- Advantage:
  - Easy to close a scope.

- Disadvantage: Difficulties encountered when a new scope is opened.
  - Need to allocate adequate amount of entries for each symbol table if it is a hash table.
    - Waste lots of spaces.
    - A block within a procedure does not usually have many local variables.
    - There may have many global variables, and many local variables when a procedure is entered.
One symbol table with chaining (1/2)

- A single global table marked with the scope information.
  - Each scope is given a unique scope number.
  - Incorporate the scope number into the symbol table.

- Two possible codings (among others):
  - Hash table with chaining.

  ```c
  main() /* Chaining at the front when names hashed into the same location. */
  {
    /* open a new scope */
    int H,A,L; /* parse point A */
    ...
    {
      /* open another new scope */
      float x,y,H; /* parse point B */
      ...
      /* x and y can only be used here */
      /* H used here is float */
      ...
    } /* close an old scope */
    ...
    /* H used here is integer */
    ...
    { char A,C,M; /* parse point C */
      ...
    }
  }
  ```
A second coding choice:

- Binary search tree with chaining.

Use a doubly linked list to chain all entries with the same name.

```c
main()
{
    /* open a new scope */
    int H,A,L; /* parse point A */
    ...
    /* open another new scope */
    float x,y,H; /* parse point B */
    ...
    /* x and y can only be used here */
    /* H used here is float */
    ...
    } /* close an old scope */
    ...
    /* H used here is integer */
    ...
    { char A,C,M; /* parse point C */
      ...
    }
}
```

Compiler notes #5, 20060512, Tsan-sheng Hsu
Pros and cons for a unique symbol table

- **Advantage:**
  - Does not waste spaces.
  - Little overhead in opening a scope.

- **Disadvantage:** It is difficult to close a scope.
  - Need to maintain a list of entries in the same scope.
  - Using this list to close a scope and to reactive it for the second pass if needed.
The “with” construct in PASCAL can be considered an additional scope rule.
- Field names are visible in the scope that surrounds the record declaration.
- Field names need only to be unique within the record.

Another example is the “using namespace” directive in C++.

Example (PASCAL code):

```pascal
A, R: record
  A: integer
  X: record
    A: real;
    C: boolean;
  end
end

... R.A := 3;  /* means R.A := 3; */
with R do
  A := 4;  /* means R.A := 4; */
...```
Implementation of field names

- **Two choices for handling field names:**
  - Allocate a symbol table for each record type used.
  - Associate a record number within the field names.
    - Assign record number #0 to names that are not in records.
    - A bit time consuming in searching the symbol table.
    - Similar to the scope numbering technique.
Locating field names

- Example:

```plaintext
with R do
begin
  A := 3;
  with X do
    A := 3.3
end
```

- If each record (each scope) has its own symbol table,
  - then push the symbol table for the record onto the stack.

- If the record number technique is used,
  - then keep a stack containing the current record number;
  - During searching, succeed only if it matches the name and the current record number.
  - If fail, then use next record number in the stack as the current record number and continue to search.
  - If everything fails, search the normal main symbol table.
Overloading (1/3)

- A symbol may, depending on context, have more than one semantics.
- Examples.
  - operators:
    - $\triangleright I := I + 3$;
    - $\triangleright X := Y + 1.2$;
  - function call return value and recursive function call:
    - $\triangleright f := f + 1$;
Implementation:

- Link together all possible definitions of an overloading name.

- Call this an overloading chain.

- Whenever a name that can be overloaded is defined:
  
  ▶ if the name is already in the current scope, then add the new definition in the overloading chain;
  
  ▶ if it is not already there, then enter the name in the current scope, and link the new entry to any existing definitions;
  
  ▶ search the chain for an appropriate one, depending on the context.

- Whenever a scope is closed, delete the overloading definitions defined in this scope from the head of the chain.
Example: PASCAL function name and return variable.
- Within the function body, the two definitions are chained.
  ▶ i.e., function call and return variable.
- When the function body is closed, the return variable definition disappears.

```
[PASCAL]
function f: integer;
begin
  if global > 1 then f := f +1;
  return
end
```
Forward reference

Definition:
- A name that is used before its definition is given.
- To allow mutually referenced and linked data types, names can sometimes be used before they are declared.

Possible implementations:
- Multi-pass compiler.
- Back-patching.
  ▶ Avoid resolving a symbol until all possible places where symbols can be declared have been seen.
  ▶ In C, ADA and languages commonly used today, the scope of a declaration extends only from the point of declaration to the end of the containing scope.

If names must be defined before their usages, then one-pass compiler with normal symbol table techniques suffices.

Some possible usages for forward referencing:
- GOTO labels.
- Recursively defined pointer types.
- Mutually or recursively called procedures.
**GOTO labels**

- **Some language like C uses labels without declarations.**
  - Implicit declaration.
- **Example:**

```
[C]
L0:
  ...
  goto L0;
  ...
  goto L1;
  ...
L1:
  ...
```
Recursively defined pointer types

- Determine the element type if possible;
- Chaining together all references to unknown type names until the end of the type declaration;
- All type names can then be looked up and resolved.
  - Names that are unable to resolved are undeclared type names.
- Example:

```
[PASCAL]
type link = ^ cell;
cell = record
  info: integer;
  next: link;
end;
```
Mutually or recursively called procedures

- Need to know the specification of a procedure before its definition.
  - Some languages require prototype definitions.
- Example:

```plaintext
procedure A()
{
    ...
    call B();
    ...
}

...

procedure B()
{
    ...
    call A();
    ...
}
```
How to determine whether two types are equivalent?

- **Structural equivalence.**
  - Express a type definition via a directed graph where nodes are the elements and edges are the containing information.
  - Two types are equivalent if and only if their structures (labeled graphs) are the same.
  - A difficult job for compilers.

```plaintext
entry = record
  info : real;  +-----> [info] <real>
  coordinates : record  +-----> [coordinates]
    x : integer;  +----> [x] <integer>
    y : integer;  +----> [y] <integer>
end
```

- **Name equivalence.**
  - Two types are equivalent if and only if their names are the same.
  - An easy job for compilers, but the coding takes more time.

Symbol table is needed during compilation, and might also be needed during debugging.
Usage of symbol table in YACC

- Define symbol table routines:
  - **Find_in_symbol_table**(name,scope): check whether a name within a particular scope is currently in the symbol table or not.
    - Return not found or an entry in the symbol table;
  - **Insert_into_symbol_table**(name,scope)
    - Return the newly created entry.
  - **Delete_from_symbol_table**(name,scope)

- For interpreters:
  - Use the attributes associated with the symbols to hold temporary values.
  - Use a structure with maybe some unions to record all attributes.

```
struct YYTYPE {
    char type;    /* data type of a variable */
    int value;
    int addr;
    char * namelist; /* list of names */
}
```
**Hints on YACC coding (1/2)**

**Declaration:**

- **$D \rightarrow TL$**
  - \{ insert each name in $2.namelist$ into symbol table, i.e., use `Find_in_symbol_table` to check for possible duplicated names; \}
  - use `Insert_into_symbol_table` to insert each name in the list with the type $1.type$;
  - allocate `sizeof($1.type)` bytes;
  - record the storage address in the symbol table entry; \}

- **$T \rightarrow int$**
  - \{ `$$\.type = int;` \}

- **$L \rightarrow L, id$**
  - \{ insert the new name `yytext` into $1.namelist$; \}
  - return `$$\.namelist` as `$1.namelist$;` \}
  - `id` \{ the variable name is in `yytext`; \}
  - create a list of one name, i.e., `yytext`, `$$\.namelist;` \}
Usage of variables:

- **Assign** \(_S \rightarrow L\_var := Expression\):
  - \{ $1.addr is the address of the variable to be stored; $3.value is the value of the expression; generate code for storing $3.value into $1.addr; \}

- **L\_var \rightarrow id**
  - \{ use **Find\_in\_symbol\_table** to check whether yytext is already declared; $$.addr = storage address; \}

- **Expression \rightarrow Expression + Expression**
  - \{ $$\value = $1.value + $3.value; \}

- **Expression \rightarrow Expression - Expression**
  - \{ $$\value = $1.value - $3.value; \}

- \ldots

- **Expression \rightarrow id**
  - \{ use **Find\_in\_symbol\_table** to check whether yytext is already declared; if yes, error ... if not, $$\value = the value of the variable yytext}