Symbol Tables

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
Definitions

- **Symbol table:** A data structure used by a compiler to keep track of **semantics** of variables.
  - Data type.
  - When is used: **scope.**

  ▶ *The effective context where a name is valid.*

  - Where it is stored: storage address.

- **Possible implementations:**
  - Unordered list: for a very small set of variables.
  - Ordered linear list: insertion is expensive, but implementation is relatively easy.
  - Binary search tree: $O(\log n)$ time per operation for $n$ variables.
  - Hash table: most commonly used, and very efficient provided the memory space is adequately larger than the number of variables.
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.
- Uniform and randomized.

**Many design for** $h(n)$.
- 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 $\cdots$

**Resolving collisions:**
- Linear resolution: try $(h(n) + 1) \mod m$ for $m$ being a prime number.
- Chaining.
  - Open hashing.
  - Keep a chain on the items with the same hash value.
  - Most popular.
- 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., \textit{main}.
- Uniformly distributed.
Contents in symbol tables

Possible entries in a symbol table:

- **Name**: a string.
- **Attribute**:
  - `Reserved word`
  - `Variable name`
  - `Type name`
  - `Procedure name`
  - `Constant name`
  - ...
- **Data type**.
- **Scope information**: where it can be used.
- **Storage allocation, size**, ...
- ...
How to store names

- **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>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>readarray</td>
<td></td>
</tr>
<tr>
<td>i 2</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>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>length</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
sort $a$ readarray $i 2$
Handling block-structures

- Nested block means nested scope.
- Example (C language code)

```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 */
        ...
    }
}
```
Two common approaches (1/3)

- 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 match.
  - 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 */
      ...
    }
}
```

Compiler notes #5, Tsan-sheng Hsu, IIS
Two common approaches (2/3)

- 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.
    - Same names hash into the same location by adding at the front.
    - When a scope is closed, all entries of that scope are removed.

```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 */
    ...
  }
}
```

symbol table: hash with chaining

- parse point B
- parse point C
A second coding choice:
- Binary search tree:

```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 */
    ...
    } /* close an old scope */
    ...
    /* H used here is integer */
    ...
    } /* close an old scope */
    }
}
```

- 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.
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.

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.
Implementation of PASCAL "with" construct

- Example:

  ```pascal
  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, success only if it matches the current 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, mean more than one thing.
- Example:
  - operators:
    - \( I := I + 3; \)
    - \( X := Y + 1.2; \)
  - function call return value and recursive function call:
    - \( f := f + 1; \)
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 from the head of the chain.
Overloading (3/3)

- 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
```
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 it is declared.

GOTO labels:
- If labels must be defined before its usage, then one-pass compiler suffices.
- Otherwise, we need either multi-pass compiler or one with “back-patching”.
  - Avoid resolving a symbol until all its possible definitions 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.
Pointer types:
- determine the element type if possible;
- chaining together all references to a pointer to type $T$ until the end of the type declaration;
- all type names can then be looked up and resolved.

[PASCAL]
```pascal
type link = ^ cell;
cell = record
  info: integer;
  next: link;
end;
```
How to determine whether two types are equivalent?

- **Structural equivalence:**
  - Express a type definitions using a directed graph using nodes as entries.
  - Two types are equivalent if and only if their structures (graphs) are the same.
  - A difficult job for compilers.

- **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, might also be needed during debugging.
How to use?

- 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($name,scope$)

- Grammar productions:
  - Declaration:
    - $D \rightarrow TL \{ insert each name in $2.namelist into symbol table, allocate sizeof($1.type) bytes, error for duplicated names \}$
    - $T \rightarrow int\{$$\.type = int\}$
    - $L \rightarrow id,L \{insert the new name into $3.namelist and put it in $$\.namelist\} | id \{create a list of one name $$\.namelist\}$
  - Allocate global and temporary data space at the end of code.
    - $P \rightarrow program \ldots end \{printf("GDATA:\n"));
      printf(" nbytes %d\n",total_Gsize);
      printf("TDATA:\n");
      printf(" nbytes %d\n",total_Tsize); \}$
More issues on usage

Expressions:

- $S \rightarrow E + E \{ \text{generate code for adding data at $1.taddr$ and $3.taddr$} \}$
  - printf("load R_1,TDATA+%d\n",$1.taddr);
  - printf("load R_2,TDATA+%d\n",$3.taddr);
  - free $1.taddr$ and $3.taddr$ from temp space;
  - printf("add R_1,R_2\n");
  - current_t = allocate temp space;
  - printf("store TDATA+%d, R_1\n",current_t);
  - $$ .taddr = current_t$

- $E \rightarrow id \{ \text{find symbol table entry, allocate at global adta space gadd} \}$
  - printf("load R_1, GDATA+%d\n",gaddr);
  - current_t = allocate temp space;
  - printf("store TDATA+%d, R_1\n",current_t);
  - $$ .taddr = current_t$