The Symbol Table

Lecture 13
Section 7.6

Robb T. Koether

Hampden-Sydney College

Wed, Feb 25, 2009
Outline

1. The Symbol Table
2. Symbol Table Entries
3. Symbol Table Functions
4. The Symbol Table Structure
5. Hash Tables
   - The put() Function
   - The get() Function
6. Assignment
The Symbol Table

- When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers.
- This information will be used later by the semantic analyzer and the code generator.

Diagram:

```
Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Code Generator
|
|
Symbol Table
```

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Code Generator
Symbol Table

Lexical
Analyzer
Syntax
Analyzer
Semantic
Analyzer
Code
Generator
Symbol
Table

When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers.

This information will be used later by the semantic analyzer and the code generator.
Symbol Table Entries

- We will store the following information about identifiers.
  - The name (as a string).
  - The data type.
  - The block level.
  - The scope (global, local, or parameter).
  - The offset from the base pointer (for local variables and parameters only).
Symbol Table Entries

- This information is stored in an object called an `IdEntry`.
- This information may not all be known at once.
- We may begin by knowing only the name and data type, and then later learn the block level, scope, and the offset.
Symbol Table Functions

- **IdEntry install(String s, int blkLev)**
  Install a new symbol in the symbol table.

- **IdEntry idLookup(String s, int blkLev)**
  Return a reference to a symbol that is in the table.

The two most basic symbol table functions are the ones that insert a new symbol and lookup an old symbol.
Inserting a Symbol

- The `install()` function will insert a new symbol into the symbol table.
- Each symbol has a block level.
  - Block level 1 contains keywords.
  - Block level 2 contains global variables.
  - Block level 3 contains parameters and local variables.
- `install()` will create an `IdEntry` object and store it in the table.
Inserting a Symbol

- When the symbol is first encountered by the lexer, we do not yet know the scope or type.
- That is determined later by the parser.
- For example, we could first encounter the symbol `count` in any of several contexts.
Looking up a Symbol

- Whenever a symbol is encountered, we must look it up in the symbol table.
  - If it is the first encounter, then `idLookup()` will return `null`.
  - If it is not the first encounter, then `idLookup()` will return a reference to the `IdEntry` for that identifier found in the table.

- Once we have the `IdEntry` object entered in the symbol table, we may add information to it.
Looking up a Symbol

- Since a variable should be declared when it first appears,
  - If the parser is parsing a declaration, then it expects `idLookup()` to return `null`.
  - If the parser is not parsing a declaration, then it expects `idLookup()` to return non-null.
- In each case, anything else is an error.
Keywords, global variables, and local variables are stored at different block levels.

C and C++ recognize further levels (blocks) within functions, delimited by braces `{ }`.

However, in C, variables local to a block must be declared at the beginning of the block.

Every time we enter a block, the block level increases by 1 and every time we leave a block, it decreases by 1.
We will implement the symbol table as a linked list of hash tables, one hash table for each block level.
Initially, we create a null hash table at level 0.
Then we increase the block level and install the keywords in the symbol table at level 1.
Then we increase the block level and install the globals at level 2.
When we enter a function, we create a level 3 hash table and store parameters and local variables there.
When we leave the function, the hash table of local variables is deleted from the list.
If we enter another function, a new level 3 hash table is created.
When we look up an identifier, we begin the search at the head of the list.
Structure of the Symbol Table

- If it is not found there, then the search continues at the lower levels.
Keywords are found in the level 1 hash table.
Looking up a Symbol

- If an identifier is declared both globally and locally, which one will be found when it is looked up?
- If an identifier is declared only globally and we are in a function, how will it be found?
- How do we prevent the use of a keyword as a variable name?
Distinguishing Between Keywords and Identifiers

- The keywords are installed at level 1 before the lexer begins.
- When the lexer finds an “identifier,” it looks it up in the symbol table.
- If it finds it at level 1, it returns the appropriate keyword token.
- Otherwise, it returns an identifier token.
Distinguishing Between Keywords and Identifiers

- The benefit of this is that it greatly simplifies the lexer.
- For example, imagine writing a regular expression that accepts \textit{i} and \textit{iff}, but not \textit{if}.
- The regular expression for true identifiers would be immensely complicated.
- However, in \texttt{JLex}, we could list the regular expressions of the keywords first, then the regular expression for identifiers.
Hash Tables

Definition (Hash Table)

A hash table is a list in which each member is accessed through a key.

- The key is used to determine where to store the value in the table.
- The function that produces a location from the key is called the hash function.
- For example, if it were a hash table of strings, the hash function might compute the sum of the ASCII values of the first 5 characters of the string, modulo the size of the table.
The numerical value of the hashed key gives the location of the member.

Thus, there is no need to search for the member; the hashed key tells where it is located.

For example, if the string were "return", then the key would be

\[(114 + 101 + 116 + 117 + 114) \mod 100 = 62.\]

Thus, "return" would be located in position 62 of the hash table.
Clashes and Buckets

Definition (Clash)

A clash occurs when two entries have the same key.

- Clearly, there is the possibility of a clash.
- In that case, the hash table creates a list, called a bucket, of those values in the table with that same location.
- When that location comes up, the list is searched.
- However, it is generally a very short list, especially if the table size has been chosen well.
The two parameters that determine how efficiently the hash table performs are

- **The capacity** - The number of buckets.
- **The load factor** - How full the hash table is.
For a given hash table capacity,

- If there are too many buckets, then many buckets will not be used, leading to space inefficiency.
- If there are too few buckets, then there will be many clashes, causing the searches to degenerate into predominately sequential searches, leading to time inefficiency.
Java has a `Hashtable` class.
Look it up on the web to see what its member functions are.
The Java `Hashtable` class will use its own hash function to hash the keys.
It will manage its capacity and load factor automatically.
The two most important `Hashtable` functions for us are `put()` and `get()`.
Hash Tables in Java

The **put** () Function

Object put(Object key, Object value);

- When we make the function call `put(key, value)`, the hashtable will
  - Hash `key` to compute the location.
  - Insert `value` into the appropriate bucket.
  - Handle any clashes.
Hash Tables in Java

The `get()` Function

```java
Object get(Object key);
```

- When we make the function call `get(key)`, the hashtable will:
  - Hash `key` to compute the location.
  - Return the `value` that matches the key.
Compilers generally create a table of strings. These strings are the “names” of the identifiers, keywords, and other strings used in the program. Thus, if the same string is used for several different identifiers, the string will be stored only once in the string table. Each symbol table entry will include a pointer to the string in the string table. For simplicity, we will not use a string table.
Assignment

Homework

- Read Section 7.6, pages 429 - 440.