Interpreters

There are two different kinds of interpreters that support execution of programs, *machine interpreters* and *language interpreters*.

Machine Interpreters

A machine interpreter simulates the execution of a program compiled for a particular machine architecture. Java uses a *bytecode interpreter* to simulate the effects of programs compiled for the JVM. Programs like SPIM simulate the execution of a MIPS program on a non-MIPS computer.

Language Interpreters

A language interpreter simulates the effect of executing a program without compiling it to any particular instruction set (real or virtual). Instead some IR form (perhaps an AST) is used to drive execution.

Interpreters provide a number of capabilities not found in compilers:

- Programs may be modified as execution proceeds. This provides a straightforward interactive debugging capability. Depending on program structure, program modifications may require reparsing or repeated semantic analysis. In Python, for example, any string variable may be interpreted as a Python expression or statement and executed.

- Interpreters readily support languages in which the type of a variable denotes may change dynamically (e.g., Python or Scheme). The user program is continuously reexamined as execution proceeds, so symbols need not have a fixed type. Fluid bindings are much more troublesome for compilers, since dynamic changes in the type of a symbol make direct translation into machine code difficult or impossible.
- Interpreters provide better diagnostics. Source text analysis is intermixed with program execution, so especially good diagnostics are available, along with interactive debugging.
- Interpreters support machine independence. All operations are performed within the interpreter. To

move to a new machine, we just recompile the interpreter.

However, interpretation can involve large overheads:

- As execution proceeds, program text is continuously reexamined, with bindings, types, and operations sometimes recomputed at each use. For very dynamic languages this can represent a 100:1 (or worse) factor in execution speed over compiled code. For more static languages (such as C or Java), the speed degradation is closer to 10:1.
- Startup time for small programs is slowed, since the interpreter must be load and the program partially recompiled before execution begins.

- Substantial space overhead may be involved. The interpreter and all support routines must usually be kept available. Source text is often not as compact as if it were compiled. This size penalty may lead to restrictions in the size of programs. Programs beyond these built-in limits cannot be handled by the interpreter.

Of course, many languages (including, C, C++ and Java) have both interpreters (for debugging and program development) and compilers (for production work).

Symbol Tables & Scoping

Programming languages use scopes to limit the range an identifier is active (and visible).

Within a scope a name may be defined only once (though overloading may be allowed).

A symbol table (or dictionary) is commonly used to collect all the definitions that appear within a scope.

At the start of a scope, the symbol table is empty. At the end of a scope, all declarations within that scope are available within the symbol table.

A language definition may or may not allow *forward references* to an identifier.

If forward references are allowed, you may use a name that is defined later in the scope (Java does this for field and method declarations within a class).

If forward references are not allowed, an identifier is visible only after its declaration. C, C++ and Java do this for variable declarations.

In CSX no forward references are allowed.

In terms of symbol tables, forward references require two passes over a scope. First all declarations are gathered. Next, all references are resolved using the complete set of declarations stored in the symbol table.

If forward references are disallowed, one pass through a scope suffices, processing declarations and uses of identifiers together.

Block Structured Languages

- Introduced by Algol 60, includes C, C++, CSX and Java.
- Identifiers may have a non-global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

Example (drawn from C):

```
int x,z;
void A() {
  float x,y;
  print(x,y,z);
                            int
}                         float
void B() {              float
  print (x,y,z)
                               int
}                        undeclared
                         int
```


Block Structure Concepts

- Nested Visibility

  No access to identifiers outside their scope.

- Nearest Declaration Applies

  Using static nesting of scopes.

- Automatic Allocation and Deallocation of Locals

  Lifetime of data objects is bound to the scope of the Identifiers that denote them.

Block-Structured Symbol Tables

Block structured symbol tables are designed to support the scoping rules of block structured languages.

For our CSX project we'll use class **`Symb`** to represent symbols and **`SymbolTable`** to implemented operations needed for a block-structured symbol table.

Class **`Symb`** will contain a method

```
public String name()
```

that returns the name associated with a symbol.

Class **SymbolTable** contains the following methods:

- **public void openScope()** {
  A new and empty scope is opened.
- **public void closeScope() throws EmptySTException**
  The innermost scope is closed. An exception is thrown if there is no scope to close.
- **public void insert(Symb s) throws DuplicateException, EmptySTException**
  A **Symb** is inserted in the innermost scope. An exception is thrown if a **Symb** with the same name is already in the innermost scope or if there is no symbol table to insert into.

- **public Symb localLookup(String s)**
  The innermost scope is searched for a **Symb** whose name is equal to **s**. Null is returned if no **Symb** named **s** is found.
- **public Symb globalLookup(String s)**
  All scopes, from innermost to outermost, are searched for a **Symb** whose name is equal to **s**. The first **Symb** that matches **s** is found; otherwise null is returned if no matching **Symb** is found.

Is Case Significant?

In some languages (C, C++, Java and many others) case *is* significant in identifiers. This means **aa** and **AA** are different symbols that may have entirely different definitions.

In other languages (Pascal, Ada, Scheme, CSX) case *is not* significant. In such languages **aa** and **AA** are two alternative spellings of the same identifier.

Data structures commonly used to implement symbol tables usually treat different cases as different symbols. This is fine when case is significant in a language. When case is insignificant, you probably will need to *strip case* before entering or looking up identifiers.

This just means that identifiers are converted to a uniform case before they are entered or looked up. Thus if we choose to use lower case uniformly, the identifiers **aaa**, **AAA**, and **AaA** are all converted to **aaa** for purposes of insertion or lookup.

BUT, inside the symbol table the identifier is stored in the form it was declared so that programmers see the form of identifier they expect in listings, error messages, etc.

How are Symbol Tables Implemented?

There are a number of data structures that can reasonably be used to implement a symbol table:

- An Ordered List
  Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.
- A Binary Search Tree
  Lookup is much faster than in a linked list, but rebalancing may be needed. (Entering identifiers in sorted order can turn a search tree into a linked list.)
- Hash Tables
  The most popular choice.

Implementing Block-Structured Symbol Tables

To implement a block structured symbol table we need to be able to efficiently open and close individual scopes, and limit insertion to the innermost current scope.

This can be done using one symbol table structure if we tag individual entries with a "scope number."

It is far easier (but more wasteful of space) to allocate one symbol table for each scope. Open scopes are stacked, pushing and popping tables as scopes are opened and closed.

Be careful though—many preprogrammed stack implementations don't allow you to

"peek" at entries below the stack top. This is necessary to lookup an identifier in all open scopes.

If a suitable stack implementation (with a peek operation) isn't available, a linked list of symbol tables will suffice.

More on Hashtables

Hashtables are a very useful data structure. Java provides a predefined **`Hashtable`** class. Python includes a built-in *dictionary* type.

Every Java class has a **`hashCode`** method, which allows any object to be entered into a Java **`Hashtable`**.

For most objects, hash codes are pretty simple (the address of the corresponding object is often used).

But for strings Java uses a much more elaborate hash function: $\sum_{i=0}^{n-1} c_i \times 37^i$

n is the length of the string, c_i is the i-th character and all arithmetic is done without overflow checking.

Why such an elaborate hash function?

Simpler hash functions can have major problems.

Consider $\sum_{i=0}^{n-1} c_i$ (add the characters).

For short identifiers the sum grows slowly, so large indices won't often be used (leading to non-uniform use of the hash table).

We can try $\prod_{i=0}^{n-1} c_i$ (product of characters), but now (surprisingly) the size of the hash table becomes an issue. The problem is that if even one character is encoded as an even number, the product *must* be even.

If the hash table is even in size (a natural thing to do), most hash table entries will be at even positions. Similarly, if even one character is encoded as a multiple of 3, the whole product will be a multiple of 3, so hash tables that are a multiple of three in size won't be uniformly used.

To see how bad things can get, consider a hash table with size 210 (which is equal to $2\times3\times5\times7$). This should be a particularly bad table size if a product hash is used. (Why?)

Is it? As an experiment, all the words in the Unix spell checker's dictionary (26000 words) where entered. Over 50% (56.7% actually) hit position 0 in the table!

Why such non-uniformity?

If an identifier contains characters that are multiples of 2, 3, 5 and 7, then their hash will be a multiple of 210 and will map to position 0.

For example, in **`Wisconsin`**, **`n`** has an ASCII code of 110 (2×55) and **`i`** has a code of 105 ($7\times5\times3$).

If we change the table size ever so slightly, to 211, no table entry gets more than 1% of the 26000 words hashed, which is very good.

Why such a big difference? Well 211 is *prime* and there is a bit a folk-wisdom that states that prime numbers are good choices for hash table sizes. Now our product hash will cover table entries far more uniformly

(small factors in the hash don't divide the table size evenly).

Now the reason for Java's more complex string hash function becomes evident—it can uniformly fill a hash table whose size isn't prime.

How are Collisions Handled?

Since identifiers are often unlimited in length, the set of possible identifiers is infinite. Even if we limit ourselves is short identifiers (say 10 of fewer characters), the range of valid identifiers is greater than 26^{10}.

This means that all hash tables need to contend with *collisions*, when two different identifiers map to the same place in the table.

How are collisions handled?

The simplest approach is *linear resolution*. If identifier **`id`** hashes to position **p** in a hash table of size **s** and position **p** is already filled, we try **(p+1) mod s**, then **(p+2) mod s**, until a free position is found.

As long as the table is not too filled, this approach works well. When we approach an almost-filled situation, long search chains form, and we degenerate to an unordered list.

If the table is 100% filled, linear resolution fails.

Some hash table implementations, including Java's, set a *load factor* between 0 and 1.0. When the fraction of filled entries in the table exceeds the load factor, table size is increased and all entries are rehashed.

Note that bundling of a **`hashCode`** method within all Java objects makes rehashing easy to do automatically. If the hash function is external to the symbol table entries, rehashing may need to be done manually by the user.

An alternative to linear resolution is *chained resolution*, in which symbol table entries contain pointers to chains of symbols rather than a single symbol. All identifiers that hash to the same position appear on the same chain. Now overflowing table size is not catastrophic—as the table fills, chains from each table position get longer. As long as the table is not too overfilled, average chain length will be small.