Symbol Tables & Scoping

Programming languages use scopes to limit the range in which 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):

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

### 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.

---

### 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 linked lists, but rebalancing may be needed. (Entering identifiers in sorted order turns 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.

---

**Reading Assignment**

Read Chapter 3 of *Crafting a Compiler.*
**Scanning**

A scanner transforms a character stream into a token stream. A scanner is sometimes called a *lexical analyzer* or *lexer*. Scanners use a formal notation (*regular expressions*) to specify the precise structure of tokens. But why bother? Aren’t tokens very simple in structure? Token structure can be more detailed and subtle than one might expect. Consider simple quoted strings in C, C++ or Java. The body of a string can be any sequence of characters except a quote character (which must be escaped). But is this simple definition really correct?

Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash. C, C++ and Java allow escaped newlines in strings, Pascal forbids them entirely. Ada forbids all unprintable characters. Are null strings (zero-length) allowed? In C, C++, Java and Ada they are, but Pascal forbids them. (In Pascal a string is a packed array of characters, and zero length arrays are disallowed.) A precise definition of tokens can ensure that lexical rules are clearly stated and properly enforced.

**Regular Expressions**

Regular expressions specify simple (possibly infinite) sets of strings. Regular expressions routinely specify the tokens used in programming languages.

Regular expressions can drive a *scanner generator.*

Regular expressions are widely used in computer utilities:

- The Unix utility *grep* uses regular expressions to define search patterns in files.
- Unix shells allow regular expressions in file lists for a command.

- Most editors provide a “context search” command that specifies desired matches using regular expressions.
- The Windows Find utility allows some regular expressions.
Regular Sets

The sets of strings defined by regular expressions are called regular sets.

When scanning, a token class will be a regular set, whose structure is defined by a regular expression.

Particular instances of a token class are sometimes called lexemes, though we will simply call a string in a token class an instance of that token. Thus we call the string abc an identifier if it matches the regular expression that defines valid identifier tokens.

Regular expressions use a finite character set, or vocabulary (denoted Σ).

This vocabulary is normally the character set used by a computer. Today, the ASCII character set, which contains a total of 128 characters, is very widely used. Java uses the Unicode character set which includes all the ASCII characters as well as a wide variety of other characters.

An empty or null string is allowed (denoted λ, “lambda”). Lambda represents an empty buffer in which no characters have yet been matched. It also represents optional parts of tokens. An integer literal may begin with a plus or minus, or it may begin with λ if it is unsigned.