

Chapter 7

Names and Binding

Overview

This chapter discusses the definition, implementation, and semantics of names. The meaning of a symbol or name in a programming language is its binding. In most languages, all sorts of entities can have names. Binding creates an association between a name and its storage object. Binding can be static; the name is bound to the object when the object is allocated and remains bound throughout the program. Or, binding can be dynamic. In this case, a name can be bound to one object, unbound, and rebound to another within the run of a program. Constant declarations bind a symbolic name to a literal value.

The scope of a name is that part of the program where a name is known by the translator. Naming conflicts occur when some name is accidentally used more than once within a linear program. Modularity and block structure allow the programmer to limit the scope of a name to a block and all its nested blocks. It is the job of the interpreter or compiler to determine the proper meaning of ambiguous names according to the semantics of the language.

7.1 The Problem with Names

This section concerns the ways that we define symbols, or *names*, in a programming language, give those names meaning (or meanings), and interpret references to names. In lambda calculus this issue is very simple; every name acquires a unique meaning in one of two ways:

1. Some names are defined, by declaration, to stand for formulas.
2. Parameter names acquire meaning during a reduction step. When a lambda expression is applied to an argument, that argument becomes the meaning of the parameter name.

Lambda calculus is referentially transparent: wherever a name appears in an expression, the defining formula can be substituted without changing the meaning of the expression. The reverse is also true; the meaning of an expression does not change if a defined name is substituted for a subexpression which matches its defining formula.

Thus lambda calculus makes a simple one-to-one correspondence between names and meanings. Most programming languages, however, are not so simple. This section tries to explain and straighten out all the myriad ways in which real languages complicate the naming problem.

7.1.1 The Role of Names

We use names to talk about objects in a computer. In simplest terms, a name is a string of characters that a programmer can write in a program. Different languages have different rules for the construction of names, but intuitively, a name is just like a word in English—a string of letters. A name must be given a meaning before it can be used. The *meaning of a name* is its binding, and we say the name is *bound* to that meaning.

While *objects* can be created dynamically in most languages, *names* cannot. Names are written in the program, and the text of the program does not change when the program is executed. Bindings, though, change when objects are created or destroyed. They attach the changing collection of objects to the fixed collection of names.

Naming would need little explanation if languages followed a “one object-one name” rule. However, the situation is not so simple. Languages permit a bewildering mismatch between the number of objects that exist and the number of names in the program. On the one hand, an object can have no name, one name, or multiple names bound to it. On the other hand, a name can be bound to no object (a dangling pointer), one object (the usual case), or several objects (a parameter name in a recursive function). This complexity comes about because of block structure, parameters, recursion, pointers, alias commands, KILL commands, and explicit binding commands. In this section, we explore the way names are used in writing programs and the binding mechanisms provided by various languages.

Symbolic names are not necessary for a computer to execute a program: compilers commonly remove names altogether and replace them by references. Nor are names necessary for a person to write a program: the earliest method for programming computers, writing absolute machine code, did not use names. Nonsymbolic programming requires considerable skill and extraordinary attention to detail, and “symbolic assemblers”, which permit the programmer to define names for locations, were a great leap forward because names help the programmer write correct code. It is much easier for a human to remember a hundred names than a hundred machine addresses.

In addition, names have an important semantic aspect that is appreciated by experienced programmers. A program that uses well-chosen names that are related to the meaning of the objects

Exhibit 7.1. Bad names.

You may enjoy the challenge of figuring out what this function does without using diagrams. At least one highly experienced Pascal programmer failed on his first try.

```
TYPE
  my_type = ↑x_type;
  x_type = RECORD next: char; prior: my_type END;
FUNCTION store_it (temp: my_type, jxq: char): my_type;
VAR jqx, first: my_type;
BEGIN
  first := temp;
  jqx := temp↑.prior;
  WHILE (jqx <> NIL) DO BEGIN
    IF jqx↑.next = jxq
    THEN jqx := NIL
    ELSE BEGIN
      first := jqx;
      jqx := jqx↑.prior
    END
  END;
  store_it := first;
END;
```

being named is much easier to debug than a program with randomly chosen, excessively general, or overused names like “J” and “temp”. A program that uses names inappropriately can be terrible to debug, since the human working on the program can be misled by a name and fail to connect the name with an observed error. It is even harder for another programmer, unfamiliar with the program, to maintain that code.

The function definition in Exhibit 7.1 was written with names purposely chosen to disguise and confuse its purpose. The English semantics of every name used are wrong for the usage of the corresponding object. A compiler would have no trouble making sense of this clear, concise code, but a human being will be hindered by ideas of what names are *supposed* to mean and will have trouble understanding the code. You may enjoy trying to decode it before reading further.

Several naming sins occur in Exhibit 7.1:

- Two names were used that have subtle differences: jxq, jqx.
- Nonsuggestive names were used: temp, my_type, x_type, jxq.
- Suggestive names were inappropriately used: “store_it” names a search routine that does no

Exhibit 7.2. Good names.

```

TYPE
  list_type = ↑cell_type;
  cell_type = RECORD value: char; next: list_type END;
FUNCTION search (letter_list: list_type, search_key: char): list_type;
VAR scanner, follower: list_type;
BEGIN
  follower := letter_list;
  scanner := letter_list↑.next;
  WHILE (scanner <> NIL) DO BEGIN
    IF scanner↑.value = search_key
    THEN scanner := NIL
    ELSE BEGIN
      follower := scanner;
      scanner := scanner↑.next
    END
  END
END;
search := follower;
END;

```

storing. “Next” names a value field, rather than the traditional pointer. “Prior” names a pointer field pointing at the next item in the list.

- A name was used that seemed appropriate on first use but did not reflect the actual usage of the object: “first” started as a pointer to the first thing in the list, but it is actually a scanning pointer.

A list of good name substitutions for the program in Exhibit 7.1 is:

```

my_type = list_type   prior = next       jxq = search_key
x-type = cell_type    next = value       jqx = scanner
store_it = search     temp = letter_list  first = follower

```

Rewritten with the names changed, the purpose of this code should be immediately apparent. Try reading the code in Exhibit 7.2. Anyone familiar with Pascal and with list processing should understand this code readily.

7.1.2 Definition Mechanisms: Declarations and Defaults

All sorts of entities can have names in most languages: objects and files (nouns), functions and procedures (verbs), types (adjectives), and more. Depending on the rules of the language, the

Exhibit 7.3. Predefined names in Pascal.

This is a list of *all* the names that are predefined in UCSD Pascal.

types	integer, real, Boolean, char, text
constants	NIL, TRUE, FALSE, MAXINT
files	input, output
functions	odd, eof, eoln, abs, sqr, sqrt, sin, cos, arctan, ln, exp, trunc, round, ord, chr, succ, pred
procedures	read, readln, write, writeln, get, put, rewrite, reset, page, new, dispose, pack, unpack

programmer might or might not be permitted to use the same name for entities in different classes. As in English, there must be some way to give meaning to a name and some way to find the meaning of a name when it is used. Declarations, defaults, and the language definition itself are the means used in programming languages to give meaning to names.

The *symbol table* is a data structure maintained by every translator that is analogous to a dictionary. It stores names and their definitions during translation.¹

A name must be defined before it can be used. In some languages this happens the first time it is used; in others all names must be explicitly declared. A *declaration* is a statement that causes the translator to add a new name to its list of defined names in the symbol table.

Many functions, types, and constants are named by the language designer, and their definitions are built into all implementations of the language. We call these *primitive symbols* [Exhibits 7.3 and 7.4]. These names are not like other reserved words. They do not occur in the syntax that defines the language, and the programmer may define more names in the same categories. They are a necessary part of a language definition because they provide a basic catalog of symbols in terms of which all other symbols must be defined.

¹This structure has also been called the *environment* or *dictionary*.

Exhibit 7.4. Predefined names in C.

These are the names that are predefined in C:

types	int, long, short, unsigned, float, double, char
constants	NULL (TRUE, FALSE, and EOF are also defined in many versions of <code><stdio.h></code> , the header file for the standard I/O package.)
functions	Every C implementation has a library, which contains I/O functions, numeric functions, and the like. The libraries are fairly well standardized from one implementation to another and are far too extensive to list here.

In many interpreted languages the programmer is not required to declare types, because allocation decisions do not have to be made in advance of execution, and at execution time, the type of a datum can often be determined by examining the datum itself. Names are generally added to the symbol table the first time they are mentioned. Thus names are typeless. These languages are sometimes called “typeless” because types are not declared and not stored in the symbol table with the names.

Objects, on the other hand, are never “typeless”. Every storage object has a fixed size, and size is one aspect of type. Every program object has a defined encoding, another aspect of type. In a “typeless” language, the type of an object must still be recorded. Since the type is not stored with the name, it must be encoded somehow as part of the object itself.

In a compiled language, the type of each name must be supplied either by a declaration or by a default so that the compiler can know how many bytes to allocate for the associated storage object. The type is stored in the symbol table with the name, and remains unchanged throughout the rest of translation. `Pascal` requires that a type be declared for each name. `FORTRAN` permits the programmer to write explicit declarations, but if an identifier does not appear in a declaration, a default type will be used which depends on the first letter of the name. The original `C` permitted function return types and parameter types, but not variable types, to be declared as “integer” by default.

7.1.3 Binding

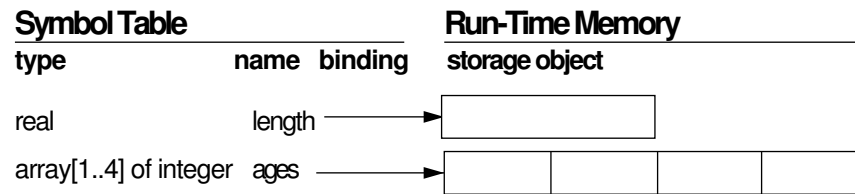
In compiled languages, a name exists *only* in the symbol table at translation time and objects exist only at run time. Names are gone before objects are created; they are not part of objects. In interpreted languages, names and objects coexist. In both cases, a name acquires one or more meanings during the course of translation, by a process called binding.

Binding creates an association between a name (in the symbol table) and a storage object (an area of memory). We can picture a binding as a pointer from the name to the storage object. A binding differs from an ordinary pointer, though, because it reaches from the system’s storage area into the programmer’s area. Moreover, in compiled languages, the binding spans time as well as space. At compile time it holds the location where an object will someday be allocated. Finally, bindings are unlike pointers because the translator automatically dereferences bindings but does not dereference pointers. We represent bindings in our diagrams as arrows (like pointers) but drawn in boldface, because they are not ordinary pointers.

Binding is invoked by the translator whenever a declaration is processed but can also be invoked explicitly by the programmer in many languages. A binding is *static* if it never changes during the lifetime of the program. Otherwise it is said to be *dynamic*. At any time a name might be *bound* to a particular object to which it *refers*, or it might be *unbound*, in which case it refers to nothing and is said to be *undefined*, or it might be *multiply bound* to different objects in different scopes.

Names of variables, types, pure values, and functions are identified and recorded in the symbol table during translation of a program. Another column of the symbol table records the bindings. Like allocation, binding can be static, block structured, or dynamic.

Exhibit 7.5. A symbol table with static bindings.



Typed Languages / Static Binding

Most of the familiar languages (COBOL, FORTRAN, ALGOL, C, Pascal, Ada) belong to a class called “typed languages”. In these languages each name defined in a program unit has a fixed data type associated with it, and often declared with it.

In the oldest and simplest of these languages, such as symbolic assemblers and COBOL, name binding is *static*. A name is bound to an object when the object is allocated and remains bound to the same storage object until the end of the program. In such languages, when a meaning is given to a name, that name retains the meaning throughout the program.

Static binding occurs in typed languages that are non-block structured. In a static language, there is no concept of a program block enclosed within another program block, producing a local program scope in which a name could be redefined.² A static binding associates a name with a storage object of fixed type and size at a fixed memory address.

Static binding can be implemented simply by using three columns in the symbol table to store the name, type, and binding [Exhibit 7.5]. We can describe this kind of symbol table as “flat”—it has the form of a simple one-dimensional list of entries, where each entry has three fields.

Each declaration (explicit or default) specifies a name and a type. It causes the compiler to select and set aside an area of storage appropriate for an object of that type. Although this storage will not exist until run time, its address can be computed at compile time and stored in the symbol table as the binding for the name. Note, in Exhibit 7.5, that the run-time memory contains only the storage object; the symbol table no longer needs to be present. It was used to generate machine code and discarded at the end of translation.³

A Typed Language with Dynamic Binding

FORTH is an interactive, interpretive language embedded in a program development system. A complete system contains an editor, an assembler, an interpreter, and a “compiler”. This compiler does not generate machine code, rather, it lexes and parses function definitions and produces an

²However, additional names can be bound to a COBOL object, by using `REDEFINES`, as explained in Section 7.1.4.

³Some translators are embedded in systems that provide a symbolic debugger. These systems must keep the symbol table and load it along with the object code for the program.

Exhibit 7.6. Names, types, and bindings in FIG FORTH.

This dictionary segment contains two words, an integer and an array of four integers. The right-hand column has 4 bytes of memory per line in the diagram.

name :	6len gth	(Length of name followed by name.)
link :	▶ ?	(Pointer to previous word in dictionary.)
type :	▶ int	(Pointer to run-time code for integer variables.)
body :		(4 bytes, properly called the "parameter field".)
name :	4age	
link :	▶ length	
type :	▶ int	(Pointer to run-time code for integer variables.)
body :	(16 bytes of storage, enough for four variables.)
	
	

intermediate program form that can be interpreted efficiently.

FORTH is a typed language. Its symbol table, called the “dictionary”, is only a little more complex than the simple, flat symbol table used for a static language. The dictionary is organized into several “vocabularies”, each containing words for a different subsystem. Each vocabulary is implemented by a simple, flat symbol table. Unlike COBOL and assembler, though, FORTH is an interactive language system. A user wishing to create a new application subsystem is permitted to create a new vocabulary or to add to an existing vocabulary. The user may alternate between defining objects and functions, and executing those functions. The dictionary may thus grow throughout a session.

The dictionary contains an entry for each defined item [Exhibit 7.6]. Function names, variable names, and constant names are all called “words”. Entries for all the primitive words are loaded into the dictionary when you enter the FORTH system. A dictionary entry is created for a user-defined word when a declaration is processed, and it will remain in the dictionary until the user gives the command to FORGET the symbol. The FORTH dictionary is stack-structured; new items are added at the top of the stack and can be defined in terms of anything below them on the stack.

Each entry has four fields:

- The *name field* holds the name of the word, stored as a string whose first byte contains the length of the name.
- The *link field* is used to organize the dictionary into a data structure that can be searched efficiently. Searching must be done during translation, when the definition of a function refers to a symbol, or at run time when the interpreter evaluates a symbolic expression interactively. The implementation of the link field and its position relative to the name field varies among

different versions and implementations of FORTH. Exhibit 7.6 shows the relationships defined for FIG FORTH.

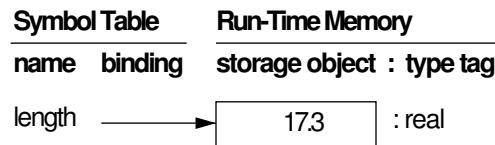
- The *code field* is the functional equivalent of a type field. It identifies, uniquely, the kind of object this word represents (function, variable, constant, or programmer-defined type).
- The *parameter field*, or body, contains the specific meaning of the word. For constants, it is a pure value. For variables it is a storage object. For functions, it contains code that can be interpreted.

FORTH maintains a rudimentary sort of type information in the “code field” of each dictionary entry. This field is a pointer to a run-time routine that determines the semantics of the name. It is actually a pointer to some code which will be run whenever this word is used at run time. This code defines the interpretation method for objects of this type. Thus constants, variables, and user-defined types can be interpreted differently. Initially only the types “function”, “variable”, and “constant” are built in, but others can be added. When a new type declarator is defined, two pieces of code are given: one to allocate and initialize enough storage for an object of the new type, and a second to interpret run-time references to the name of an object of this type. A pointer to this second piece of code becomes the unique identifier for the new type, and also becomes the contents of the code field for all objects declared with the new type.

FORTH differs from the simple static languages in one important way: it permits the user to redefine a word that is already in the dictionary. The translator will provide a warning message, but accept the redefinition. Henceforth the new definition will be used to compile any new functions, but the old one will be used to interpret any previously compiled functions. This opens up the possibility of redefining primitive symbols. The new definition can call the original definition and, in addition, do more elaborate processing. The simple relationship between a name and its meaning no longer holds at all.

The FORGET command is an unusual feature that has no counterpart in most language translators. It does not just remove one item from the dictionary, it pops the entire dictionary stack back to the entry before its argument, forgetting everything that has been defined since! This is a rudimentary form of symbol table management which does not have either the same purpose or the same power as the stack-structured symbol tables used to implement block structure. A FORTH programmer alternates between compiling parts of his or her code and testing them. FORGET lets the programmer erase the results of part of a compilation, correct an error in that part, and recompile just one part. Thus FORGET is an important program-development tool.

Typed Languages / Block Structured Binding. The connection between a name and its meaning is further complicated by block structure. FORTH permits a new definition of a name to be given, and it will permanently replace the old version (unless it is explicitly “forgotten”). A block structured language permits this same kind of redefinition, but such a language will restore the original definition after exit from the block containing the redefinition. Block structure and the semantic mechanisms that implement it are taken up in Section 7.4.

Exhibit 7.7. A symbol table with dynamic binding.


Explicit Dynamic Binding. Fully dynamic binding is available only in interpreted languages or ones such as LISP with simple, uniform type structures. In such a language, types can be associated with objects, not names, and are stored with the object in memory, rather than in the symbol table. The symbol table has only two columns, the name and its current binding. The type must be stored with or encoded into the object in memory, or discarded altogether as in assembly language. This is illustrated in Exhibit 7.7.

With fully dynamic binding, a name can be unbound from one object and rebound to any other at any time, even in the middle of a block, by explicit programmer command. In such a language, the type of the object bound to a name may change dramatically, and these languages are sometimes called “typeless” because no definite type is associated permanently with a name. SNOBOL and APL are examples of this language class.

These “typeless” languages nevertheless commonly do implement objects of different types. For example, in APL there are two basic types, number and character. These types are implemented by attaching a type tag to the object itself, rather than to the name in the symbol table [Exhibit 7.8]. The symbol table contains only the name and the binding, and the programmer is permitted to bind a name to any object. Thus at different times a name may be bound to storage areas of different sizes, each with an associated type tag.

In such languages, binding often serves the same purpose as does assignment in Pascal and is often mistaken for assignment. The essential difference is that assignment does not change the storage object to which a name is bound, but changes the program object which is the contents of

Exhibit 7.8. Names, types, and binding in APL.

APL is a “typeless” language and so has no permanent association of types with names. Rather, a type tag is associated with each storage object, and the combination may be bound to any name.

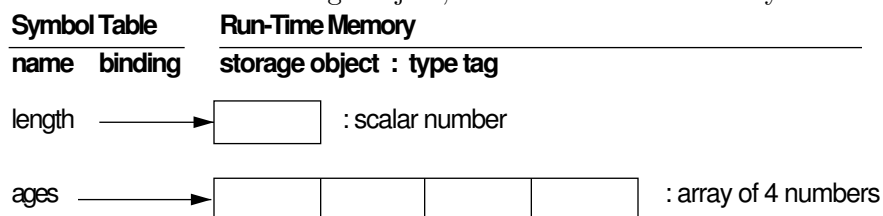


Exhibit 7.9. Dynamic binding in APL.

On two successive executions of the input statement

$$Q \leftarrow \square$$

one could legally supply the following inputs: 'aeiouy' (which is a character array) and 37.1 (which is a number). Thus we would get first the following binding:

Symbol Table	Run-Time Memory						
name binding	storage object : type tag						
Q —————→	<table style="display: inline-table; border-collapse: collapse;"> <tr> <td style="border: 1px solid black; padding: 2px 5px;">a</td> <td style="border: 1px solid black; padding: 2px 5px;">e</td> <td style="border: 1px solid black; padding: 2px 5px;">i</td> <td style="border: 1px solid black; padding: 2px 5px;">o</td> <td style="border: 1px solid black; padding: 2px 5px;">u</td> <td style="border: 1px solid black; padding: 2px 5px;">y</td> </tr> </table> : array of 6 characters	a	e	i	o	u	y
a	e	i	o	u	y		

and second the following binding:

Symbol Table	Run-Time Memory	
name binding	storage object : type tag	
Q —————→	<table style="display: inline-table; border-collapse: collapse;"> <tr> <td style="border: 1px solid black; padding: 2px 10px;">37.1</td> </tr> </table> : scalar number	37.1
37.1		

that storage object. Binding results in a different storage object being bound to a name.

The APL input command is: $\leftarrow \square$. Executing \square causes a pure value of some type to be accepted as input from the user's terminal. The input may be a single number or character (called a scalar) or it may be an array (or vector) of any length of either base type. The type of the value is determined, storage is allocated and initialized to this value, and a reference to this storage is returned. The operator \leftarrow binds this new object to the name on its left. Exhibit 7.9 uses the \square operator to illustrate the dynamic nature of binding in APL.

7.1.4 Names and Objects: Not a One-to-One Correspondence

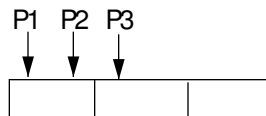
We have seen that although the number of names in a program cannot change dynamically, many applications require that the storage in use expand dynamically. We, therefore, must be able to refer to storage locations that do not have unique names. This need is met by using addresses, or pointers, rather than names to refer to storage, and by binding the same name, recursively, to objects in different stack frames.

Conversely, there are times when it is convenient to bind multiple names to a single storage object or a part of an object. This is often done in order to associate a second data type with that object, or to package logically independent objects together into a group to facilitate their handling. (A full discussion of this topic is in Chapter 15.) The names given to this kind of declaration in some common languages are as follows:

Exhibit 7.10. Sharing through EQUIVALENCE in FORTRAN.

```
DIMENSION P1(3)
EQUIVALENCE (P1, P2), (P1(2), P3)
```

The DIMENSION statement causes an array of length 3 to be allocated and bound to P1. The EQUIVALENCE statement binds a second name to the base address of the array P1, and a third name to a segment of that array, as illustrated below. P1, P2, and P3 now “share” storage.



FORTRAN	EQUIVALENCE declaration
COBOL	REDEFINES clause in a declaration
Pascal	Variant record with no tag field
C	Union data type

When multiple-name binding is used, storage is not allocated for the second name, but it is bound to the same address as the first and serves as a second way to refer to the same storage object [Exhibit 7.10]. If used carelessly, this can cause bizarre misinterpretations of data.

On the other hand, misinterpreting the data on purpose can be an easy way to compute a pseudo-random function. Exhibit 7.11 shows a variant record used as the input to a hashing function. The data is really a character string, but the hashing function is told to construe it as an integer and square it. The result is semantically meaningless but can be used to access a hash table. A diagram of this dual-type object is shown in Exhibit 7.12.

7.2 Binding a Name to a Constant

A constant declaration permits the programmer to attach a symbolic name to a literal value. Although giving a name to a constant does not change the meaning of a program, a judicious choice of names can clarify the programmer’s intended meaning to other programmers. Defining the name once and then using it many times in place of a frequently used value also makes it easier to modify a program: only the definition needs to be changed to change all occurrences of that value.

Many but not all languages permit the programmer to name constants. Some place severe restrictions on the kinds of constants that can be named and/or on the ways initial values can be specified. For example, Pascal requires that the value of a constant be a literal value of a primitive type. Constant expressions may not be used as initial values [Exhibit 7.13] even though they could easily be evaluated by the compiler, as they are in FORTRAN 77 [Exhibit 7.14].

Exhibit 7.11. A variant record used to compute a hash index.

We define a type that provides storage for 4 bytes, which can be interpreted either as a single integer or an array of four characters. This type is useful in writing a hash function that takes a character array as its parameter and uses integer operations to compute a hash address. Diagrams of the variant-record object are shown in Exhibit 7.12.

```

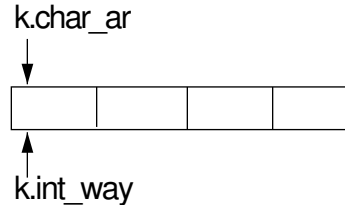
TYPE twokinds = 1..2;
  hashkey = RECORD CASE twokinds OF
    1: (char_ar: ARRAY [1..4] of char);
    2: (int_way: integer)
  END;

FUNCTION hash (k: hashkey): integer;
BEGIN
  hash := (k.int_way * k.int_way) MOD table_size
END;

```

Exhibit 7.12. Diagrams of a Pascal variant record.

The storage object for *k* in Exhibit 7.11 would have the following structure:



This object has dual semantics. If the actual parameter were the string 'ABCD', the two interpretations of the program object would be as follows:

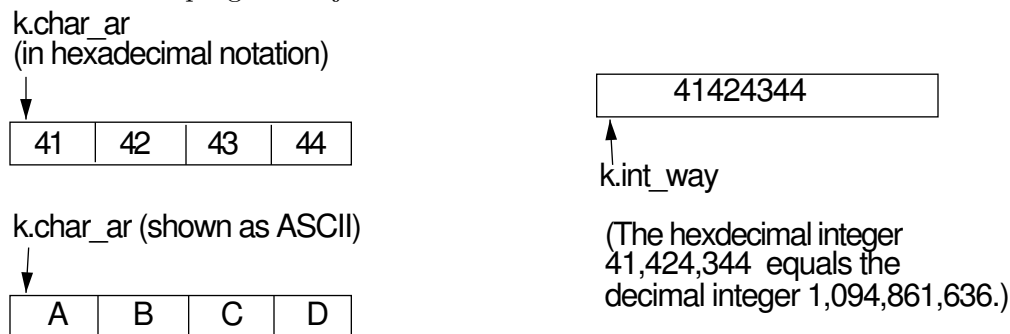


Exhibit 7.13. Constant declarations in Pascal.

```

CONST pi = 3.1416;
      max_length = 500;
      max_index = 499; { Equals max_length - 1 }

```

A real constant and two integer constants are declared. As is often the case, the last one depends on the second, but the dependency cannot be made explicit in Pascal and is indicated only by a comment.

In Ada, constant declarations do not have to precede the rest of the program. A constant declaration may come after variable declarations and depend on the initial values given to those variables [Exhibit 7.15].

FORTH, like Ada, permits the declaration of a constant at any point during a program. A constant definition consists of an expression, the word `CONSTANT`, and the name of the constant. The expression will be evaluated first, and its value will be bound to the name. FORTH, unlike

Exhibit 7.14. Constant declarations in FORTRAN 77.

```

PARAMETER (PI = 3.1416, MAXLEN = 500, MAXIX = MAXLEN-1)

```

The same three constants are created as in the Pascal example of Exhibit 7.13. In FORTRAN it *is* possible to use a constant expression to specify the value of a constant. `PARAMETER` declarations must precede the executable code.

Exhibit 7.15. Mixed variable and constant initializations in Ada.

```

pi:  CONSTANT float := 3.1416;
m:   integer;
n:   integer := 5;
max_length:  CONSTANT integer := n * 100;
max_index:  CONSTANT integer := max_length - 1;

```

The real constant `pi` and two integer variables are declared. The second variable is initialized to 5. The integer constants `max_length` and `max_index` are declared and bound, respectively, to 500 and 499. These are legal defining expressions because `n` is previously initialized. The following constant declaration would not be legal because the value of `m` is not defined:

```

max_width:  CONSTANT integer := m - 1; -- Not legal.

```

Exhibit 7.16. Variable and constant declarations in FIG FORTH.

```

100 VARIABLE offset
500 CONSTANT max_length
max_length offset + CONSTANT max_index

```

A variable named `offset` is defined and initialized to 100. A constant named `max_length` is defined and initialized to 500. A second constant named `max_index` is initialized to the sum of `offset` plus `max_length`.

Ada, is interactive. This declaration is translated at run time, not at a prior compile time. Thus there is no need to restrict the initializing expression, which may depend on the current (run-time) value of any name [Exhibit 7.16]. The scope of the constant name is all of the program that follows the declaration.⁴

The original K&R C contained no special defining word for constants, but a semantics that meets the same needs as the constant declaration in Pascal can be achieved by using the compile-time preprocessor that is part of C. Each `#define` statement is a macro definition and is expanded by the preprocessor, replacing the defined names by the defining strings before the beginning of actual compilation. [It is conventional, in C, to use capital letters for macro (constant) names and lowercase for other identifiers.]

There is an important difference between a using a macro name and using a true constant. In a constant declaration, if the constant value is defined by an expression, that expression will be evaluated once, at compile time. In a macro definition, the expression will be evaluated at run time every time the constant name is used. Of course this is inefficient, but it also leaves open the possibility that the wrong answer might be computed because of unintended interactions between the macro expansion and its context.

In Exhibit 7.17, the parentheses around the expression on the third line prevent unintended interaction between the “-” operator in the expanded macro and other operators in the surrounding program. Assume the parentheses were omitted, and consider this call on the constant `MAX_INDEX`:

```
totsize = MAX_INDEX * 10
```

This macro call would be expanded before parsing, yielding:

```
totsize = 500 - 1 * 10
```

Thus the value of `totsize` would be 490, not the intended 4990.

For two reasons, then, a macro-preprocessor does not wholly take the place of a constant declarator in a language. First, it can lead to unnecessary run-time calculations unless the compiler performs optimization on constant expressions. Second, macros can be tricky and misleading to use. True constants are efficient and simple. Thus a `const` declaration, with semantics similar to that in Ada, was added to the language by the ANSI C standard. The last line in Exhibit 7.17

⁴Scope is explained in Section 7.4.

Exhibit 7.17. “Constant” declarations in C.

```
#define PI 3.1416
#define MAX_LENGTH 500
#define MAX_INDEX (MAX_LENGTH - 1)
const int max_index = MAX_LENGTH - 1;    /* ANSI C only */
```

shows the declaration of a constant integer. The initializing expression will be evaluated once, when storage for `max_index` is allocated, and the result will be stored in the allocated object. Thereafter, the value cannot be changed.

7.2.1 Implementations of Constants

In spite of the differences in syntax, the semantics of constants are nearly identical in FORTH, FORTRAN, and Pascal.⁵ In all three languages, a name is entered into the symbol table, a value is provided or calculated and bound to it at compile time, and that binding does not change thereafter.

These semantics can be implemented in two ways. One alternative is to evaluate the defining expression for the constant before compilation, and substitute the result for the constant’s name in the source code. There can be no possibility of accidentally assigning a new value to the constant, because constant names are eliminated from the program before compile time. In this case, it might not even be necessary to allocate any run-time storage for the constant. Instead of compiling the code to fetch from a memory address (like a variable), the compiler might make the constant into a part of the compiled code by generating a “load immediate” instruction.

The second implementation for constants permits their names to exist in symbolic form during translation. This is necessary if the constant name is to have a block structured scope. The translator allocates a run-time storage location for the constant, evaluates the defining expression, and stores the result in the allocated space. This is the same way a variable would be initialized. In this case, some mechanism must be embedded in the translator to prevent the programmer from changing the constant by an assignment or read statement. We can say that this implementation provides an *initialized read only variable* (IROV).

Constants are implemented as IROVs in FORTH. The storage set aside for constants and variables is the same, and the initialization process is the same. Different semantic behavior is achieved at run time by associating different semantic routines with constants and variables. The translator puts a constant’s *value* on the stack when its name is referenced. But when a variable name is referenced, the *storage address* is placed on the stack and must be explicitly dereferenced to obtain the value of the variable.

⁵Except that the scope of definition of the constant name cannot be restricted to an inner block in FORTH or FORTRAN.

7.2.2 How Constant Is a Constant?

The IROV implementation of constants leads to a generalization that is found in *Ada* and *ANSI C*. We see that the constant names in *Pascal* can have local scope. (That is, a constant name declared in an inner block or subroutine is only “known” in that block.) It would be logical to evaluate the initializing expression at block entry time, rather than at compile time.

If we defer evaluation and binding for a constant until block entry time, constants can be created and initialized when local variables are allocated. They are essentially initialized read-only variables. An initializing expression can contain references to parameters and variables in outer blocks as well as to literal constants. With this interpretation, the defining expression is reevaluated each time the block is reentered, leading to the situation that a constant might not be constant! (Such a local constant would remain constant for the lifetime of its block, but the name might be bound to a different constant value the next time the block was entered.)

The primary advantage of such a block entry constant is that a constant can be calculated based on input parameters, and yet the language translator guarantees the integrity of the value: it can not be accidentally changed. The cost is more time spent during block entry.

7.3 Survey of Allocation and Binding

Early High-Level Languages.

The designers of the earliest high-level languages expected the language users to want to represent certain types of external objects. They made it easy for the programmer to represent those types. *COBOL* records are a natural representation of a business transaction, being very much like the paper representation of those transactions. *FORTRAN* makes the representation of numbers and vectors natural. At this historical stage of language development, there was a close relationship among (1) external objects being represented, (2) internal names for those objects, and (3) the storage objects that implemented them. There is no question of what constitutes an object in these languages: it is the combination of all of the above.

Allocation is static, and binding is static except for parameter binding. Each set of similar external objects (policies, dimensions, etc.) is represented by a typed identifier. If the external object has parts and subparts, each part in the representation can be named and referenced. When an identifier is defined by the programmer, a storage object is allocated and bound to that identifier for the life of the program.

Additional names can be bound to a storage object through *REDEFINES* or *EQUIVALENCE* statements. Additional names can also be bound to objects temporarily by parameter binding during a call on a subprogram.

Interactive Languages. An interesting and different set of choices was made in *BASIC*. Name binding is totally static: subprograms do not have parameters, so unlike *FORTRAN*, there is no need to deviate from totally static binding. On the other hand, allocation of storage for string

variables is dynamic. This is done by statically binding the string identifier to a pointer object and dynamically binding the pointer to a string object in string storage.

Block Structured Languages. Concern over the difficulty of debugging large programs was beginning to develop in the late 1950s when ALGOL was designed. ALGOL was a remarkably advanced language for those years: it incorporated a clean design, adequate control structures, and recursion.

Block structure, local variables, and parameter binding were devised to ease the problems of name conflicts. With these new semantic mechanisms, the same identifier could be used to correspond to different external objects in different program blocks. This eased the writing of large programs by permitting the programmer to disregard or forget that an identifier had already been used once, so long as the previous use was in an irrelevant context. Block structure is also needed to support recursion.

Dynamic Languages.

With LISP, we see a radically different approach to modeling external objects and processes. LISP objects are implemented by linked structures, called *lists* of storage objects called *cells*. A list is a pointer to a cell or to a simple object called an *atom*. A cell is a pair of lists. Identifiers do not have data types, but can be bound to any object at any time. Storage is allocated dynamically for parameters and also by calling the allocation function CONS. (CONS takes two parameters. It constructs a cell and initializes its two fields to those parameters.)

We see that the relationship between external objects and storage objects exhibited by the block structured languages breaks down completely in LISP. External objects are represented by lists, which can point at lists that represent smaller external objects. Big objects are made out of references to smaller objects, which are true objects, not subfields. This is in contrast to the COBOL concept that big objects may have many small parts, but these parts are not, literally, at the same level.

Variable names exist in LISP, corresponding roughly to FORTRAN object names. However, the number of variable names is not limited to the number of names introduced by the programmer; names can be generated dynamically by a program. Variables may be bound to LISP objects, but the binding is dynamic, not static, so that one variable name may refer to different program objects with different storage amounts and locations at different times. Since storage objects may be dynamically created and pointer assignment is provided, the size of a single program object is bounded only by available storage.

APL, another early interactive language, has dynamic allocation and binding for parameter names and local variables in functions. The basic objects in APL are arrays, either numeric or alphabetic. A tremendous variety of array and matrix operations are predefined. These can produce results of any size and shape, for which storage is allocated dynamically. The resulting references are bound dynamically to untyped identifiers. Global names can be used without declaration.

Local variable names, parameter names, and a local name for the function's return value need to be declared in the function header to distinguish them from global names.

Smalltalk is a relatively new language that extends the LISP concept of object by having all objects belong to classes, either system-defined or programmer-defined. With this extension, objects must now contain a class (or type) indicator as well as private storage. Again, big objects are made out of smaller ones. It is significant that, as in APL, this type information is attached to the storage object rather than to an identifier. It is this property that has earned the label *object-oriented language* for Smalltalk.

Combining Static and Dynamic Objects. In Pascal, ALGOL-like block structured allocation and binding exist side-by-side with LISP-like dynamic creation and pointer assignment, although the two facilities are not closely merged. Pascal programs tend to use either one allocation scheme or the other, with little mixing, because you cannot point at stack-allocated objects.

In C, this restriction on what a pointer may point at is lifted. The address of any storage object and any part of that object is available within the program and can be stored in a pointer. Pointers may, therefore, point at any object.

Identifier binding is block structured in both C and Pascal. An identifier is bound to a new storage object at block entry and remains bound to the same object until it is deallocated. Even though objects can be dynamically allocated from heap storage, these are bound to pointers rather than identifiers.

Data types are associated with identifiers, as in FORTRAN, not with objects, as in Smalltalk and APL. Pointers have a declared base type and should only be bound to objects of the appropriate type. C will give warning errors but still compile if this rule is breached. Pascal will fail to compile.

7.4 The Scope of a Name

7.4.1 Naming Conflicts

When symbolic programming languages were new, each identifier stood for one storage object, and there was no need to distinguish between identifiers and internal names. Very soon though, people began to compile programs that included subroutines written by other people. They began to encounter *naming conflicts* when they would accidentally use some name that was used internally by the subroutine.

The first approach to avoiding such name conflicts was to assign a unique prefix to all of the names used in the subroutine. For example, a SIN subroutine that needed a local variable COUNT might use the name SINCOUNT. The programmer needed only to avoid using names with that prefix in other subroutines. To simplify the programmer's task, some early symbolic languages contained a PREFIX statement that would automatically prefix the name, so the programmer could write COUNT and the translator would convert it to SINCOUNT.

As programs became longer, naming conflicts became a problem even within the same program. A programmer working on one part of the program could not easily remember all of the names

used in other parts of the program, and accidental duplication of names could lead to bugs that were very difficult to locate.

The concepts of *modularity* and *scope* were developed to alleviate this problem. A program is written as many manageable-sized modules, each with a particular well-defined purpose. Modules interact with each other only in well-defined ways. Names defined within a module, unless explicitly “exported”, are not “visible” outside of the module. This introduced the new concept of the *scope* of a name, which is the portion of code in which the name is meaningful. Scoped names could be reused for new objects in different scopes.

The scope of a name is that part of the program in which the name is known and will be recognized by the translator. Scope can be *global*, in which case the name is known throughout the program, or it can be *local*, meaning that it is only known within that program block in which it was defined. In programs with nested scopes, a name, N , may also be *relatively global* by being declared in a block that is neither innermost nor outermost. In programs that are compiled in many modules, names can also have *external scope*, which means that the name is known to all modules.

Along with scoping came the need to distinguish between two concepts that until then had been synonymous—identifiers and complete names. *Identifiers* are the symbols that the programmer writes, and *complete names* are the things that receive bindings. The scope is added to the identifier the programmer writes in order to obtain the *complete name*. Thus if COUNT is defined in two modules SIN and SORT, then the complete names generated by the translator might be written SIN.COUNT and SORT.COUNT. Only one declaration for an identifier may occur per scope, and thus the translator can form a unique name by sequentially numbering the scopes it encounters and concatenating a scope number with the identifier that the programmer used. An attempt to redefine an identifier in the same scope is a translation-time semantic error.

In most modern languages, however, the programmer does not give names to all scopes. Nevertheless, the same rules apply. The identifier COUNT, defined in two different scopes s_1 and s_2 , denotes two different names. When programs are translated, the complete name is formed automatically by the translator. There is no need for the programmer to write the complete names, so complete names are not a part of the syntax for programming languages. However, we still need a way to write a complete name in this chapter, so we will write the name as a pair of a scope number and an identifier, for example, (s_1 .COUNT) and (s_2 .COUNT).

The programmer refers to (s_1 .COUNT) when writing the identifier COUNT within scope s_1 and *has no way to refer to this variable elsewhere*. This limitation is intentional and is what gives scope its power. By providing no way to write the complete name ($s_1.y$) outside of the scope s_1 , we are making accidental references to ($s_1.y$) impossible. The information stored in a named variable is only “visible” or accessible within the scope of the name. It is “hidden” from the rest of the program.

In most languages, each subroutine definition comprises a scope, and thus parameters can be given “dummy names”. It is immaterial whether the identifiers used for dummy names are the same as or different from names in the calling program, for the translator will generate different complete names for them.

Language design came full-circle with the invention of object-oriented languages, where programmers, again, have reason to write complete names. Each object class has a name and defines a scope. Within the definition of a class, the programmer uses simple names to refer to the class members (objects and functions). Some class members are private, and are accessible only within the class definition. However, a class may also have public members, which can be used by other parts of the program.⁶ When this is done, a double colon symbol, called the *scope-resolution operator* is written between the class name and the member name to form a complete name.

7.4.2 Block Structure

The idea of *block structure*, introduced in ALGOL-60, permitted the programmer to define nested modules called *blocks*. In a block structured language, the programmer can introduce additional blocks at quite arbitrary places, wherever they are convenient. Identifiers declared within a block are translated to complete names whose scope is that block *and all nested blocks*. A well-written program uses many short blocks, and names are declared in the smallest possible block. Each block corresponds to a scope. Thus several sets of declarations might be relevant to a particular use of a name in the program.

This situation can arise in C because a new block may be opened anywhere, using “{”, and names may be defined at the beginning of any block.

In Pascal, blocks cannot be opened and closed at arbitrary places. But each function or procedure body is a block, and it may be nested inside other function or procedure definitions.⁷ Thus we arrive at nested scopes by a different path. The simple program in Exhibit 7.18 has three scopes which are outlined by boxes. One scope is created by the main program, and two by functions within it. Let us call the scopes A, B, and C. Scope A is the lexical parent of scopes B and C.

Nested blocks reintroduce ambiguity into the naming rules that scopes were invented to avoid. We said previously that although an identifier might refer to different names in different scopes, the identifier together with the scope in which it was contained was enough to disambiguate it. Now, however, we see that scopes can be nested, so an instance of an identifier can appear simultaneously in the several scopes of its enclosing blocks. Once again we are faced with an ambiguity. We must determine which scope identifier to use in forming the complete name.

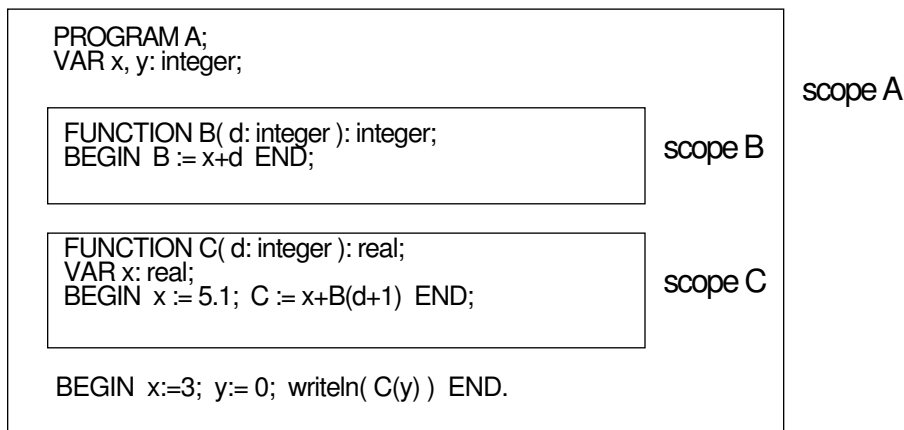
To resolve this ambiguity, we introduce yet another concept—that of a *defining occurrence* of an identifier. Each identifier is given meaning by a declaration.⁸ At that point, the identifier, with its type, is entered into the symbol table. All other occurrences of that identifier are *nondefining occurrences* and are called *uses*. Uses are found in executable statements and initial value clauses.

An identifier can be redefined by a new declaration in a nested block, and this introduces a new complete name. The scope of each complete name is the smallest scope that contains its declaration. In order to translate a use of an identifier, the compiler must decide which complete

⁶Object oriented scope and referencing rules are explained in Chapter 16, Section 16.4 and in Chapter 18, Section 18.2.

⁷All function definitions in C are at the top level. They may not be nested within each other.

⁸In some languages a name is defined by default when the translator sees the identifier for the first time.

Exhibit 7.18. Nested scopes in Pascal.

name corresponds to it. The rule used is called *lexical scoping*:⁹ The complete name to which a use, U , refers is the one produced by the nearest declaration for U . The nearest declaration is the one in the smallest enclosing block.

Exhibit 7.18 has a short program with a nest of three scopes, defined by the main program, (scope A) and the two subroutines (scopes B and C). The object identifiers in use are x , y and d . Each parameter or local variable declaration defines a complete name, thus the complete names formed from these identifiers are: $(A.x)$, $(A.y)$, $(B.d)$, $(C.x)$, and $(C.d)$. Subroutine names are visible in the scope of the enclosing block (so that they may be called from that block). Thus the complete names of the subroutines are $A.B$ and $A.C$. This program is rewritten using complete names in Exhibit 7.19. Exhibit 7.20 shows the contents of the stack during execution of function B.

A straightforward implementation of lexical scoping works as follows. When a block or subprogram is entered, storage for its parameters and local variables is allocated in a stack frame on the run-time stack. The stack frame also includes the return address for the subprogram and a pointer to the stack frames of the lexical and dynamic parents of the block. The pointer to the lexical parent is called the *static link*, and the pointer to the dynamic parent is called the *dynamic link*.

The static links are the means of implementing complete names, and they provide an easy way to describe lexical scoping. To find the correct complete name for a use, U , the compiler must start at the stack frame for the block that contains U . If there is no declaration for U in that stack frame, start following the chain of static links backward through the stack. The first declaration for U that you find is the relevant one. Following the static link can bypass a variable number of stack frames that were allocated for blocks called by the lexical parent of the current block.

Once the compiler determines which scope defines U , the chain of stack frames can be short-

⁹The word “lexical” is used because the scope of a name is a static property determined by how the program is laid out on the listing.

Exhibit 7.19. Complete names in Pascal.

```

PROGRAM A;
VAR (A.x), (A.y): integer;

  FUNCTION B( (B.d): integer ): integer;
  BEGIN B := (A.x)+(B.d) END;

  FUNCTION C( C.d: integer ): real;
  VAR C.x: real;
  BEGIN (C.x):=5.1; A.C:= (C.x)+A.B((C.d)+1) END;

BEGIN (A.x):=3; (A.y):= 0; writeln(A.C(A.y)) END.

```

Since *x* is not redefined in scope *B*, the *x* referred to in that scope is the one in the lexically enclosing scope, which is scope *A*.

circuited so that run-time references will be more efficient. If the definition is found at a global level, the address to which it is bound is static and may simply be compiled into the code. If the definition of *U* is neither local nor global, but occurs in some scope between local and global, extra storage can be allocated in the local stack frame for a pointer to the defining reference. At block entry time the actual address of the proper binding can be determined by tracing through the frames by the above method, then copied into this pointer area. A use of a relatively global variable then becomes like a use of a VAR parameter, causing an automatic dereferencing of the pointer. (VAR parameters are explained in Chapter 9, Section 9.2.)

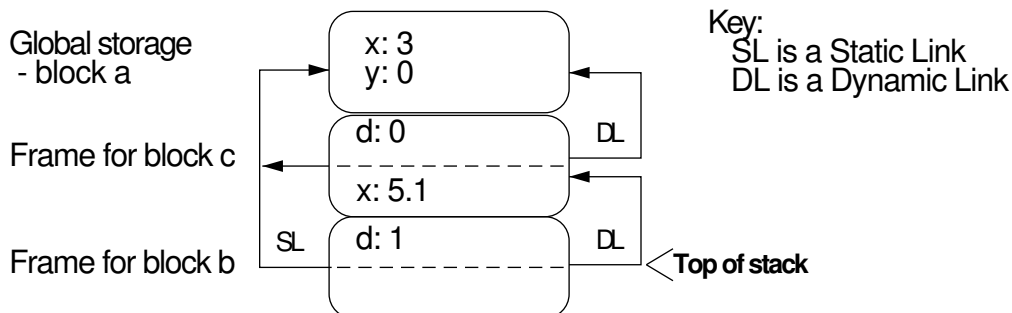
A final, confusing aspect of nested scopes must be mastered: the phenomenon of masking. Re-declaration of an identifier locally will *mask* any definitions of the same identifier that are relatively global to it. By this we mean that the object created by the declaration in the enclosing block cannot be accessed within the enclosed block because every use of that identifier in the enclosed block will be associated with the object created in the smaller scope. In the Pascal example [Exhibits 7.18 and 7.19], the object (A.y) is accessible within function *C* because it was declared globally. But the object (A.x) cannot be accessed within *C* because the local declaration for *x* will “swallow up” all references to *x*.

A Block Structured Symbol Table.

Let us say that a complete name is *born* when the compiler begins translating the block in which it is defined. It *dies* at the end of that block. The symbol table must include only names that are alive, since dead names must not be used to interpret any references.

Exhibit 7.20. A stack diagram for nested scopes.

This diagram shows the contents of the stack during execution of the program in Exhibit 7.18, at the point that the main program has called function `c` and `c` has called function `b`, but `b` has not yet returned.



In a block structured language, a name becomes *visible* when it is born and *invisible* again when it dies. However, a living name also becomes temporarily invisible when it is masked by a declaration in an inner block, and it becomes visible again when the masking variable dies.

Block structured binding cannot be implemented with a “flat” symbol table (that is, a one-dimensional list) because bindings for invisible names must be retained, and bindings for dead names must be discarded. Thus the symbol table grows and shrinks, and we can describe it as “stack-structured”. For each name declared in a block, a type and binding are pushed onto the stack for that name when the compiler begins translating the block and are popped off when the translator reaches the lexical end of the block. There are two ways to organize the stack—either the bindings for the block can be inserted as a group, or each variable name can have an associated stack of bindings. Exhibit 7.21 shows a program in which several variables are declared in more than one block. Note that a function name is in the scope of its enclosing block, not the block created by the function definition. Exhibit 7.22 shows this stack-structured symbol table with a stack of bindings for each multiply defined name.

7.4.3 Recursive Bindings

The number of names in a particular program is determined by the programmer and does not change after the program is translated. However, any language with recursion permits creation of an unlimited number of stack-allocated objects at run time. On each invocation of a recursive function, new storage is allocated, in a new stack frame, for parameters and local variables. This new storage is bound to the parameter and local names and remains bound to them until control exits from the subprogram. We say that each stack frame corresponds to one *dynamic scope*.

Thus each name declared in a recursive function must be bound simultaneously to several objects in different dynamic scopes. If a subprogram has called itself five times and not yet returned once,

Exhibit 7.21. A Pascal program demonstrating block structure.

This program contains a function within a function. During translation of the innermost function block, block III, three sets of bindings are recorded in the symbol table. This is diagrammed in Exhibit 7.22. However, only the bindings for the innermost block are active, or *visible*, at this time.

```

program demo_bind;                                     I
const a=10;
var b, c: real;
  function d (a:real): real;                           II
    function c (b:integer): real;                       III
      begin c:=b/2 end;
    begin d := a+c(b) end;
begin writeln(a+trunc(d(c))) end.

```

Exhibit 7.22. A symbol table with block structured bindings.

We depict a block structured symbol table using columns for the object's type, block identifier, name, and binding. The symbol table entry for each name is a stack of types and bindings. These are pushed onto and popped off of the symbol stacks at the beginning and end of translation of the relevant program block.

Symbol Table			Run-Time Memory
type	block	name binding	storage object
real variable	II	a	→ []
integer constant	I	a	→ 10
integer variable	III	b	→ []
real variable	I	b	→ []
function(integer): real	II	c	→ [A piece of executable code]
real variable	I	c	→ []
function(real): real	I	d	→ [A piece of executable code]

there are six sets of storage objects simultaneously bound to the local names (one for the original call and one for each of the five recursive calls).

Recursively bound names are ambiguous, and this ambiguity is not like the static ambiguity introduced by block structure. In a block structured program, a name may be simultaneously within the scope of several definitions [Exhibit 7.18]. But we are able to identify and resolve the resulting ambiguity at *compile time* by identifying the lexical scope of each name and forming a unique complete name from the block identifier and object identifier.

This method for disambiguation will not work for the dynamic, run-time ambiguity caused by recursion because the recursive invocations all come from the same lexical block. A single symbol declaration, such as the declaration for `jj` or `kk` in Exhibit 6.18, produces more than one allocation and binding. The disambiguation rule that applies here is the rule for *dynamic scoping*: the *most recent* active binding is used. To find that binding, start searching in the current stack frame. If the name is not defined there, follow the *dynamic link* back to the stack frame of the calling function, and so on. All frames on the stack will be examined in order, until the definition of the identifier is found.

Lexical versus Dynamic Scoping. The rule for dynamic scoping is used in place of the lexical scoping rule in non-block-structured languages such as APL and the older forms of LISP. In any language where both recursion and nested scopes are supported (such as Pascal or C), both lexical and dynamic scoping rules must be used. The rule for lexical scoping determines the mapping from multiply defined identifiers onto complete names. Then the rule for dynamic scoping defines the meaning of names in recursive scopes.

For simple situations, lexical and dynamic scoping produce the same result. However, in general, they do not. The difference is illustrated by the interpretation of the global variable `x` in Exhibits 7.23 (LISP) and 7.25 (Pascal). LISP uses dynamic scoping, as diagrammed in Exhibit 7.24, and Pascal uses lexical scoping, as diagrammed in Exhibit 7.26. The functions defined in the two languages are identical except for the scoping rules used to interpret them, but they compute and print different answers.

7.4.4 Visibility versus Lifetime.

The scope of a name is not necessarily the same as the lifetime of the object to which it is bound. This mismatch of scope and lifetime can happen in several ways. The most familiar is by using a reference (VAR) parameter in a subroutine call. During the procedure call the dummy parameter name is bound to an actual parameter. Storage for a calling block has a longer lifetime than the called block, since the called block will exit first. Thus, within the procedure, the parameter name refers to an object with permanence greater than the scope of the name.

In languages with dynamic allocation, such as Pascal and C, list and tree structures can be built out of heap-allocated objects. As each new cell of the structure is allocated, a reference to it is stored in its predecessor. A reference to the first cell of the structure must be stored in a named, stack-allocated, pointer variable. Frequently the scope of these pointers is less than global, while

Exhibit 7.23. Dynamic scoping in LISP.

	Program Notes
(defun A (x y) (printc (C y)))	Define a function named A, with two parameters, x and y. The function prints the result of calling a function named C on the parameter y.
(defun B (d) (+ d x))	Define a function named B, with one parameter, d. The function returns the result of adding d and x. Since x is not defined locally, it is a global reference .
(defun C (d) (let (x '5.1)) (+ x (B (+ d 1))))	Define a function named C, with one parameter, d. Set local variable x to value 5.1. Return the result of adding x to the result of executing func- tion B on the parameter d incremented by 1.
(A 3 0)	Call function A with parameters 3 and 0. It will print the number 11.2.

Exhibit 7.24. Diagram of bindings in dynamic scoping.

Stack frames (with dynamic links) are shown for the stage of execution at which all the functions in Exhibit 7.23 have been called and none has returned. The name *x* is used but not defined in function B, so its meaning is determined by following the dynamic links backward until the definition of *x* in function C is encountered.

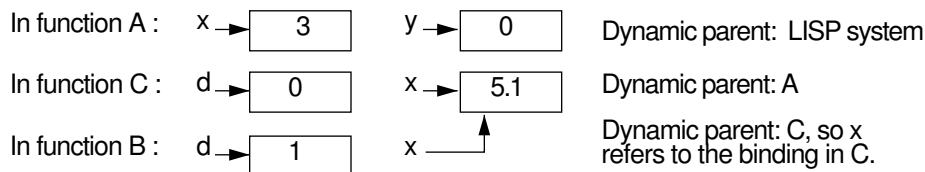


Exhibit 7.25. Lexically scoped functions in Pascal.

```

PROGRAM a;
VAR x, y: integer;
FUNCTION b(d: integer):integer;
BEGIN b:= x+d END;
FUNCTION c(d: integer):real;
VAR x: real;
BEGIN x:= 5.1; c:= x + b(d+1) END;
BEGIN x:= 3; y:= 0; writeln(c(y)) END.

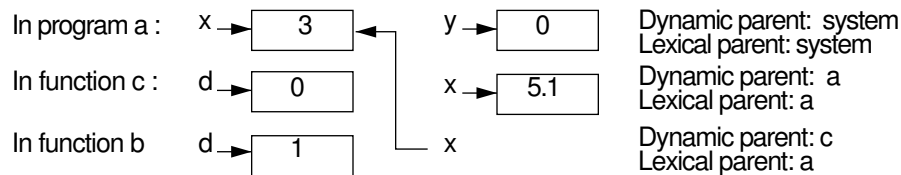
```

We define a main program, named **a**, and two functions, **b** and **c**. The program is the lexical parent of both functions. When this is executed, **a** calls **c** which calls **b**. The number 9.1 will be printed.

Exhibit 7.26. Lexical scoping in Pascal.

At one point during execution, the main program has called function **c** which has called function **b**, so stack frames for all three coexist. Program **a** is the dynamic parent of **c**, which is the dynamic parent of **b**.

Pascal is a lexically scoped language, so one follows the static link from the current stack frame to find the meaning of nonlocal names. The **x** referred to in function **b** is not the object in **c**'s frame, but the one in **a**'s frame, because **a** is the lexical parent.



heap storage is permanent. Thus the head-pointer points at an object that could live longer than itself and could be passed outward by copying its reference into a pointer with a longer lifetime.

Nonhierarchical Sharing. A serious shortcoming of lexical scoping is its strict hierarchical nature: names defined within a block cannot be exported and made known outside the block. In a language that is limited to hierarchical scoping, there are useful protection and sharing mechanisms that cannot be defined or emulated. Examples of such data facilities are “named common” storage in FORTRAN and “packages” in Ada. Such facilities can be superimposed on lexical scoping only by expanding a language’s semantic basis to include more than one kind of scoping.

An Ada package has two classes of names, some which are known outside the scope of the package, some known only within the package.¹⁰

In FORTRAN, access to a storage object may be shared by several subroutines. To do this you place the object’s name in a COMMON statement and include identical COMMON declarations in all subroutines that need access to the object. The user can create and name several independent COMMON areas. These named COMMON areas can be used to set up a nonhierarchical sharing structure [Exhibit 7.27]. This sharing structure cannot be built in a strictly block structured language, since block structure and lexical scoping either permits no sharing of data at all, or indiscriminate sharing among all subroutines at the same level.

A COMMON area does not “belong” to any single subroutine, but is allocated in the program environment area where it can be accessed by any subroutine containing the right declaration. The declaration for a COMMON statement supplies names (with associated types) which will be bound to successive locations in the common area.

It is the programmer’s job to ensure that all subroutines that share a common area contain compatible declarations. The compiler will not check that. If the order or types of the names declared in two subroutines differ, everything will compile and link, but compute nonsense. On the other hand, the particular names declared in different subroutines are completely arbitrary and can be different.

An example of the use of COMMON to achieve nonhierarchical sharing is shown in Exhibit 7.27. There are three common areas, and matching common declarations are included in the pair of subroutines that share each area. Subroutines POINT and BLANK share the storage area named POINTBLANK which contains the variables X and Y. Likewise, subroutines POINT and CHECK share area CHECKPOINT and BLANK and CHECK share BLANKCHECK. The common areas and bindings created by these definitions are diagrammed in Exhibit 7.28. Note that CHECK and POINT call the variables in area CHECKPOINT by different names, but the data types of the components do match.

Static Local Storage. ALGOL OWN variables and C variables with the “static” storage class are examples of another kind of mismatch between lifetime and scope. These variables are statically allocated and are immortal like global variables, but the scope of the names of these objects is block structured. Each time a block is entered in which a static variable is declared, the variable

¹⁰See Chapter 16 where this topic is fully developed.

Exhibit 7.27. Named common in FORTRAN.

Just the COMMON declarations are given here for three subroutines that share storage in a non-hierarchical fashion.

```

SUBROUTINE POINT
COMMON /POINTBLANK/ X, Y(100)
COMMON /CHECKPOINT/ JJ, KK, LL

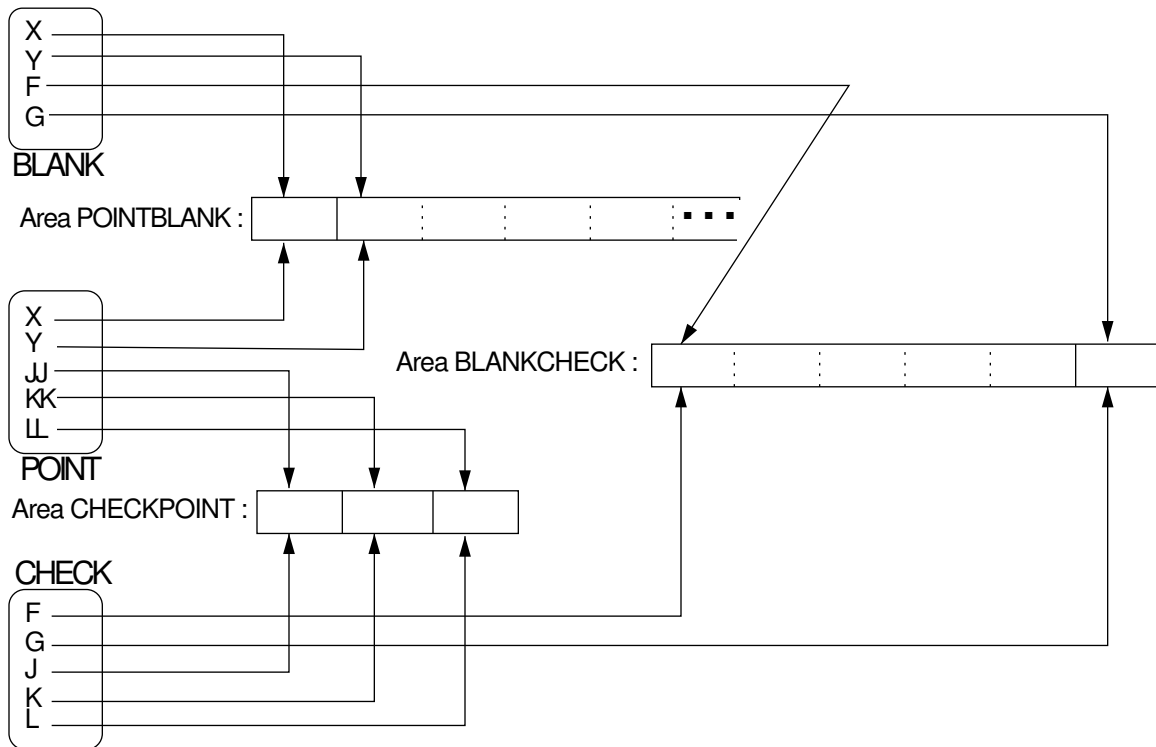
SUBROUTINE BLANK
COMMON /POINTBLANK/ X, Y(100)
COMMON /BLANKCHECK/ F(5), G

SUBROUTINE CHECK
COMMON /BLANKCHECK/ F(5), G
COMMON /CHECKPOINT/ J, K, L

```

Exhibit 7.28. A diagram of common storage in FORTRAN.

This is a storage diagram of the common areas created in Exhibit 7.27. An arrow represents a binding.



name becomes known again and refers to the static object, which becomes accessible. The run-time system ensures that the name always refers to the same storage address. At block exit time the name becomes undefined, and outside the block, no name refers to the object. The object is then invisible and remains inaccessible and unchangeable until its block is reentered.¹¹

This is an important facility. It permits the value of a variable to be retained between executions of a subprogram and yet protected from external tampering. The need for this is apparent if you consider an output buffering procedure which must remember how full it left the buffer in order to know where to store the next value. The buffer pointer must be retained between calls on the output function, yet it should be “hidden” from all other parts of the program to ensure unhindered operation. The ability to “hide” information is now recognized as a cornerstone of good programming practice.

In a language such as Pascal, which does not support static local objects, the buffer pointer would have to be declared globally (all globals are static). This works, but it increases the scope of the pointer to the entire program and creates the possibility that the value of the pointer can be accidentally changed by some remote and irrelevant section of code.

7.5 Implications for the Compiler / Interpreter

A language translator must determine the proper meaning of an ambiguous name according to the semantics defined for that language. We need to make a distinction here between the way interpreters and compilers handle the task.

One of the essential differences between interpreters and compilers stems from the fact that more information is often available at run time than at compile time. An interpreter can query the user about the required amount of data on each run and allocate arrays exactly as long as are needed. A compiler must allocate storage before actual data values are known, and therefore, perhaps, before actual storage requirements are known. The programmer using a compiler must, therefore, establish array lengths adequate for the maximum size data set the program will ever process. If a program is to process data sets whose size is highly variable, its array lengths will be excessive on most runs, and considerable storage will be allocated that is never used or needed.

Typed languages can be translated by a compiler. Type declarations permit a compiler to anticipate how much storage will be needed in the future to hold program objects. Commands can be compiled to increment and decrement the stack allocation pointer by the appropriate amounts when control enters and leaves program blocks at run time. Typeless languages cannot be compiled in the same sense; the source code can be parsed and symbol tables can be set up, but storage management must be done dynamically because the size of the storage objects that will be needed is not predictable.

In an interpreter, translation and execution proceed at the same time. Dynamic allocation with *dynamic binding* is the natural and easy method to implement. The symbol table exists during execution, and binding of allocation address to symbol is done when the allocation happens.

¹¹An extended example of the use of static storage in C is given in Chapter 16.

Each symbolic reference is interpreted just before it is executed, in the context of the results of prior computations, and thus the most recent binding of a symbol is used. The path of execution of any program may depend on inputs and conditionals and, therefore, be impossible to predict. Consequently, dynamic binding may bind a name differently from run to run, causing inconsistent interpretations of global references and errors that are hard to track down or identify.

In a compiler, on the other hand, all symbolic names are discarded at the end of compilation, before execution begins (except when a symbolic debugger is in use). All binding of symbolic name to storage location is done within the compiler, even for local variables that become allocated in the middle of execution. For locals, the compiler determines the address, relative to the current stack frame, that they will occupy in the future when they become allocated. Thus, for an ambiguous name, a compiler must maintain a stack of bindings that is parallel to the stack of allocation areas that will exist in the future when the program is executed. A binding is pushed onto the stack at the time the compiler begins to translate a subprogram and is popped off it when the end of the subprogram is compiled. The binding used to interpret a symbolic reference in the program is the one on the top of the stack. This translation scheme implements lexical scoping: the binding used to interpret a name is the one in the smallest enclosing block in which the symbol is defined. Lexical scoping is the *only* method that can be implemented by a compiler. A compiler can determine how definitions are nested, but it cannot guess the order in which they will be executed.

Lexical scoping is considered to be superior for two reasons. First, the binding that will be used is always predictable, and, therefore, programs with lexical scoping are easier to debug. Second, there are languages, for example LISP, for which it is common to write both interpreters and compilers. The interpreter is used while a program is being developed and debugged, then the compiler is used to produce code that executes faster. Since dynamic and lexical scoping produce different semantics, it has often been the case that a program, fully debugged under an interpreter using dynamic scoping, will not work when it is compiled and will have to be debugged again. The scoping discipline affects the meaning of a program and, therefore, *should* be part of a formal definition of any programming language.

Interpreting Block Structure. Block structure is implemented using stack allocation and block structured binding. The following is a description of the operation of block structure in an interpreted language:

1. Storage objects are allocated on the run-time stack whenever a block is entered. One object is allocated for each locally declared name or parameter. Local variables are usually not initialized. Parameter storage is initialized to the actual parameter values or references to variable parameters.
2. The new storage object is bound to the local name, giving the name an *additional binding*. If the block is a recursive procedure, the name could have bindings created in the enclosing blocks, in which case more than one storage object is bound to the same name, and the meaning of the name is the most recent (dynamic) binding. If control leaves the block and then reenters it, the name will be bound to a different storage object.

- Local bindings remain static until control leaves the block at *block exit* time. A given name refers to the same storage object from block entry until block exit. At block exit, all local storage objects are deallocated and local identifiers revert to their prior bindings (possibly “undefined”). The freed storage is made available for reuse by “popping” the run-time stack.

The number of storage objects on the stack changes only when blocks are entered or exited, and then only changes by the number of names (locals + parameters) declared in that block.

Compiling Block Structure. The implementation of block structure in a compiled language is not quite the same as the implementation in an interpreted language. The difference is that the compiler deals with storage that *will be* allocated rather than storage that *is* allocated. The address at which a stack frame will start is completely unpredictable at compile time. However, the number, size, and order of the items in each future stack frame is known to the compiler.

A common frame-management scheme is to have a central “core” in each stack frame that contains everything except parameters and local variables. A calling program puts parameters on the stack and increments the stack pointer (which is kept in a machine register which we will call the *SP*). Then the calling program fills in the static link, dynamic link, and return address in the core area of the stack frame. It puts a pointer to the dynamic link field of the new frame into a machine register. (Let us call this register the *FP*, for frame pointer.) The contents of the new *dynamic link* is the current value of *FP*. Finally, the program branches to the subroutine.

The subroutine can now find its stack frame because the address is in the *FP*. It can use the *SP* to find the top of the stack. The first action of the subroutine is to increment the *SP* to allocate space for local variables. The subprogram code is compiled to refer to its parameters and local variables *using addresses that are relative to the FP*. The last parameter might be $FP - 2$, and the first might be $FP - 12$. The local variables will be in locations such as $FP + 4$ and $FP + 20$. Stack locations past the *SP* are available for use as a scratch pad and will be used for temporary storage during calculations.

At block exit, several things happen:

- The return value, if any, of the subprogram is loaded into a machine register. (It is often called the return register.)
- The dynamic link field is copied back into the *FP*.
- The *SP* is decremented by the size of the stack frame.
- A branch is taken to the return address.

This erases all trace of the subprogram.

Exercises

1. What is a name? What is the meaning of a name? Do languages follow a one object-one name rule? Explain.

2. Do compilers need names to execute a program? Explain. What is the semantic aspect of a name? Why is this important for human interpretation of a program?
3. What is the role of the symbol table? How are new names added to it?
4. In APL, why are names typeless?
5. What are primitive symbols? Why are they necessary?
6. How do names relate to objects in a compiled language? In an interpreted one?
7. What is binding? How is it invoked?
8. What do we call storage that is allocated once at load time and remains until program exit, but whose usage is restricted to one function?
9. Define and contrast static binding, dynamic binding, and block structured binding.
10. In modern functional languages, assignment is not supported, and in APL, it is rarely used. Dynamic binding takes the place of assignment. Explain how this is implemented.
11. In block structured languages, when is a name visible? Invisible?
12. This question and the next seven involve the following skeletal program written in C. How many name scopes are defined in this program skeleton? Draw a box around each block and label these blocks with the letters A, B, C, etc.

```

int x, y, z;
fun1()
{
    int j, k, l;
    {
        int m, n, x;
        ...
    }
}
fun3()
{
    int y, z, k;
    ...
}
main()
{ ... }

```

13. Name a block that is nested within another block.
14. In which scopes are the global variables `x`, `y`, and `z` all accessible?
15. What variables are accessible to the `main` function?

16. Are `k` in `fun1` and `k` in `fun3` the same variable? Why or why not?
17. Are the global variables `y` and `z` accessible in `fun3`? Why or why not?
18. In which blocks does `j` refer to a global variable? In which blocks is `j` a local variable?
19. In which block or blocks can we use both `m` and `k`?
20. Name a language in which storage can be shared by two functions, neither of which is enclosed within the other, but kept private from all other functions and from the main program. Explain how to accomplish this goal in your language.
21. How are FORTRAN's "named common storage" and Ada's "packages" examples of nonhierarchical data facilities?
22. What is the function of a constant declaration? Why is it useful to the programmer?
23. What is the difference between defining a true constant whose meaning is a constant expression and using a macro to define a symbolic name for that expression?
24. How are constants implemented in these languages: Pascal, Ada, FORTRAN, FORTH?
25. What is a naming conflict?
26. How did the concept of scope and modularity resolve the problem of naming conflicts? What is the difference between an identifier and a complete name?
27. What is a block structure? What are nested blocks? What is the scope of the complete name of identifiers declared within a block?
28. Consider a language in which all variables must be declared with fully specific types. Is this language more likely to be interpreted or compiled? Why?
29. Why is the scope of a name not equal to its lifetime? Explain.
30. Explain how the compiler and interpreter each determine the semantics of an ambiguous name.