5. THE IMPLEMENTATION OF ALGOL W

The two preceding chapters outline the data and control structures which appear with minor variations in most implementations of Algol-like languages, and they suggest how mechanisms for providing debugging facilities can be superimposed upon such structures. By emphasizing the generality of certain ideas, however, those chapters obscure the many crucial and often nontrivial problems of detail which must be solved for a particular language and implementation. This and subsequent chapters describe the internal structure of an implemented debugging system, which is based upon the Algol W language and the IBM System/360, IBM System/370, or ICL System 4 hardware. Various design problems are pinpointed and their solutions are described. Some of these are of general interest; others are specific to Algol W or the System/360 and are mentioned primarily to suggest the kinds of difficulties other designers should anticipate.

This chapter sketches the underlying implementation of Algol W. It illustrates the application of some of the ideas in Chapter 3 and provides information necessary for understanding subsequent chapters. It also attempts to demonstrate that many features of the Algol W language and System/360 hardware naturally lead to design decisions which, as subsequent chapters show, simplify the implementation of useful debugging aids. The treatment of detail is intentionally selective; very cursory attention is given to those features of the compiler and object code which either are quite standard or are unrelated to the provision of debugging tools. A more thorough, although rather obsolete, description of the compiler is available elsewhere [Bauer 68].

The implementation of the Algol W system has been shaped by its history. Despite the fact that Algol W is being used primarily for teaching and research work in university environments, the organization of its compiler resembles that of a "production" compiler (see, for example, [Cocke 70, pp. 13-18, 730-755]) much more than that of a "student" compiler such as WATFOR [Shantz 67]. The original goals of the Algol W implementation project were stated as follows:

> The evolving project should be conducted in a thorough and systematic manner ... making use of the best available methods on compiler construction known. The results should consist of a well-organized system whose structure and principles are sound and precisely understood ... [Wirth 68a].

These goals suggested a rather ambitiously conceived multipass translation scheme. Before the system was completed, it was redesigned for educational use in a university setting. Such use demanded high compilation speed and minimum system overhead. Experiments showed that acceptable performance could be achieved by modifying the operating system interface without changing the basic structure of the compiler. The resulting system has justifiably been criticized for lacking balanced design emphasis [Wirth 68b], but the outcome was a happier one than might have been expected under the circumstances.

Certain consequences of the project's history must be considered in evaluating the design and performance of the debugging system. The emphasis upon the quality of the object code in the original design has contributed to showing that good debugging tools are compatible with acceptably efficient program execution. Although the compiler makes no attempt to apply global transformations which would improve the generated code, it does perform considerable, albeit somewhat uneven, local optimization. With some significant exceptions associated with parameters called by value, the important special cases of most constructs are recognized and appropriately compiled. In addition, the run-time system is carefully organized to minimize administrative overheads, and calls of interpretive system subroutines are avoided wherever possible.

Note

There is no widely accepted standard for measuring compiler performance. In an independent evaluation of Algol compilers, however, the efficiency of the object code produced by the Algol W compiler was judged to equal or exceed that of code produced by all of the nineteen other compilers investigated [Wichmann 70, 72a]. The study included most of the ALGOL 60 compilers supplied by the major computer manufacturers as well as several privately developed ones. The sizes of the object programs produced by a number of these compilers were also investigated; Algol W's object code was found to be slightly more compact than the average [Wichmann 72b]. Unpublished measurements made by the author suggest that Algol W object code compares favorably with that generated by many FORTRAN and PL/I compilers, but the best optimizing FORTRAN compilers [Cocke 70, pp. 242-273] produce code which is two to four times faster.

Acceptable performance with student jobs is obtained by using main storage for all of the compiler's intermediate output. Thus the organization of the Algol W system implies a sacrifice of some space to gain speed, and techniques for the efficient use of auxiliary storage to hold the data structures required by the debugging system have not been investigated. Also bypassed have been problems associated with the merging of independently compiled programs, although it is clear that neither the compiled code nor the tables of debugging information are inherently incompatible with a conventional linking loader.

5.1. System/360 Architecture

The organizations of both the code produced by the Algol W compiler and the run-time environment have been influenced by the architecture of the IBM System/360 [IBM 6x, Blaauw 64]. "System/360" is a generic name for a family of computers based upon a set of processors (i.e., hard-wired or microprogrammed interpreters) offering essentially identical instruction sets but a range of cost and performance levels. The family is well known and widely used; furthermore, it has influenced the design of many other computing systems. Several American and European manufacturers have built compatible hardware, while others have adopted basically similar machine organizations. In addition, the instruction set of the System/370, IBM's more recent range of machines, is identical. Thus the System/360 and System/370 are fair representatives of the current generation of computers.

In its functional characteristics, the System/360 is an evolutionary extension and refinement of earlier designs. The architecture reflects an attempt to provide an integrated instruction set suitable for both scientific and commercial data processing. That instruction set might be considered a word- and accumulator-oriented one, inherited from "scientific" machines, which has been extended to include the character- and storage-oriented operations typical of "commercial" computers. In addition, both the addressing structure and the details of the available instructions facilitate the use of reentrant and easily relocatable code.

The basic unit of information storage and transfer is the byte (or character), a sequence of eight bits. Sequences of four and eight bytes form words and double words respectively. Standard representations and transformations of certain common data types, such as integers, are provided. Main memory cells store a single byte of information, but most instructions reference a contiguous sequence of cells, e.g., a word. The address space of the memory is the set of integers in the range $[n_1, n_2]$, where $0 \le n_1 < n_2 < 2^{24}$. Furthermore, addresses can be computed using standard integer arithmetic. The System/360 provides two sets of central registers directly visible to the programmer. There are sixteen general registers, denoted by R0, R1, ..., R15, and four floating-point registers. Each general register contains a word of information and can be used indiscriminately for accumulating integer or Boolean quantities, for indexing, or for generating an address. The floating-point registers hold double words and serve as floating-point accumulators. Small integers are used to address each set of registers. These addresses are explicit in each instruction; there is no hardware stack. A special register, called the condition code, is set by comparisons and certain other operations; it is interrogated by conditional branch instructions.

With respect to the implementation of Algol W, the most notable feature of the System/360 is the addressing structure. Absolute 24-bit addresses can be formed and manipulated, but machine instructions which reference memory have only 16-bit address fields. Such addresses are really pairs of the form (b, d); the absolute address of the actual operand is given by

$$[\text{contents}(b) + d] \ \mathbf{mod}\ 2^{24} \ .$$

Here b is interpreted as the address of a general register, which is called a *base register* in this context, and d is an absolute displacement.

Note

Some instructions allow specification of a second general register, called an index register, in the absolute address calculation. Many useful instructions do not allow this further level of indexing, however, and it is not used extensively in Algol W object code.

Since all memory address fields of instructions contain such pairs, System/360 programs must arrange to keep appropriate base values in the general registers. The displacement d is specified by 12 bits; thus an instruction reference to memory location m requires that, for some register b ,

$$\text{contents}(b) \leq m < \text{contents}(b) + 2^{12} \ .$$

The code generated by the Algol W compiler makes extensive use of instructions from the "scientific" subset; these have relatively systematic and uniform formats which are easy to assemble and decode. In most arithmetic operations, including address calculation, one operand is contained in an accumulator and the second can come from an accumulator or from storage. Instructions outside the scientific subset are more specialized and less systematic; they are exploited by the administrative code which maintains the run-time environment and are used extensively within the compiler and the supporting run-time system.

5.2. Segmentation

A segment of address space is the set of addresses within some interval [n, n+d) , where d > 0. Such an interval defines a segment of main memory (or simply a *segment*), which is the collection of all cells with addresses in the given interval. The integer n is the origin of the segment; its length is d . In the Algol W run-time system, storage is divided into segments. Whenever a given segment is directly accessible, some base register contains the origin of that segment. Conversely, whenever the value of the corresponding origin is available, the storage within a segment can be made accessible. Much of the administrative code in compiled Algol W programs is concerned with the management of base registers. The problem is considerably simplified by the fact that the structures of Algol W programs and of the necessary run-time environment suggest a natural segmentation.

In Algol W, as implemented for the System/360, the text associated with each of the following entities requires the dynamic allocation of storage for local variables or state information (cf. Section 3.4):

An explicit procedure declaration.

A "proper" block, i.e., the main program itself or any block which contains declarations in its head but is not itself the body of a procedure declaration.

An actual parameter, except one which either is a constant or is an identifier of a simple variable, an array, a procedure, or a formal parameter.

A record class declaration.

Execution of the corresponding text causes the creation of an activation record, which is represented by a dynamically allocated *data segment*.

The program text is also segmented. For each procedure declaration and proper block, the compiled code forms a *program segment* containing machine instructions and constants. Space for such a segment is allocated and initialized before execution begins. The code corresponding to an actual parameter is usually quite short and is embedded within the program segment containing the procedure call (see Section 5.5.2), while a record declaration is simply translated into a table entry for later interpretation (see Section 5.3.3).

In the sequel, mention of proper blocks will often be omitted, for they are treated exactly as nameless, parameterless procedures which are called at their points of declaration.

5.2.1. Data Segments

Each data segment represents an activation record and provides storage locations for any variables local to the corresponding procedure instance. Identifiers are bound to storage locations representing the local variables by the scheme presented in Section 3.6. Binding of an identifier I declared in the text for congruence class Q is implemented as follows:

> The fields of an activation record are represented by contiguous sequences of cells within the corresponding data segment; thus a field selector can be characterized by an offset j' and a length l'. The offset is the relative address of the first byte of the field in any data segment representing an activation record in the class Q, and the length is the number of bytes in that field. The value of l' is determined by the declared type of I. The mapping F_Q is computed by the compiler, and $F_Q(I) = (j', l')$.
>
> During execution, data segments are allocated from available storage as they are required. If I is bound by the context to $x.F_Q(I)$, where $x \in Q$, and if a is the origin of the data segment representing the record x, the identifier is bound to the storage cell with absolute address $a + j'$. If base register b contains the value a and if j' is suitably restricted, the address field of a System/360 instruction referencing the storage cell can and usually does have the form (b, j'). The length l' is implicitly encoded in the operation code of the instruction.

A system organization which guarantees that any required segment origins are immediately available in the general registers is the subject of the next several sections. This organization provides particularly convenient and efficient access to variables.

Note

If j is the usual "ordinal number" determined by the position of the declaration of the identifier I within its unit of textual scope (see, e.g., [Randell 64, Section 2.2]), the mapping from j to j' is not linear for the following reasons:

Sizes of Algol W storage cells are type-dependent.

Certain storage management routines require allocation of contiguous storage for all reference variables within a segment (see Section 5.3.3).

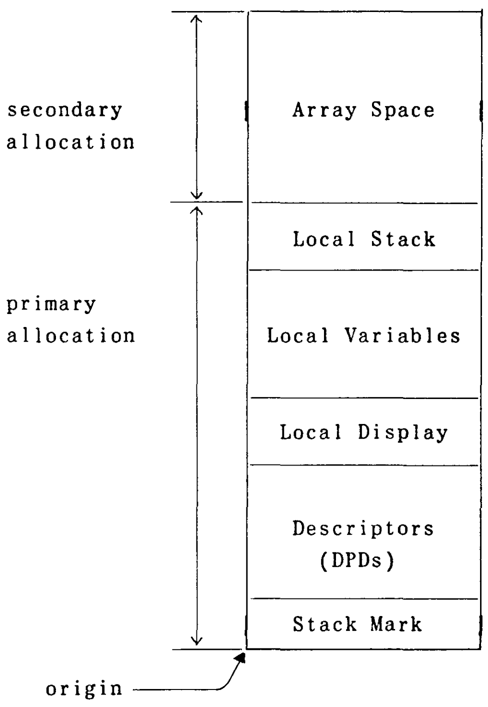
Some distinct syntactic (i.e., lexical) scopes are combined for purposes of storage allocation. A single data segment usually provides the storage associated with both the formal parameters of a procedure and the local variables of the outermost block of its body; also, a control identifier is bound to a storage cell which is allocated as part of the storage for the enclosing procedure or block.

The Algol W language maintains a clear distinction between those activation records which necessarily have nested lifetimes and those which do not (cf. Section 3.7.3). The latter are exactly the members of congruence classes defined by Algol W record class declarations. The program texts associated with such classes are always empty, and it is convenient to imagine that the corresponding activation records are created, deleted from S , and made elements of $U - S$ as parts of single state transitions. In each well-defined state, therefore, the only elements of $S - \{P\}$ are the activation records in the congruence classes defined by Algol W's procedure declarations, and the only elements of $U - S$ are the retained activation records in the classes defined by record declarations.

The lifetimes of all elements of $S - \{P\}$ are properly nested. Space for the data segments representing these activation records is managed as a LIFO stack. The implementation uses the convention that allocated space in the stack grows "upward", i.e., if a_1 and a_2 are the origins of segments associated with procedure instances x_1 and x_2 respectively,

$$a_1 > a_2 \text{ iff } x_1 \rightarrow^+ x_2 .$$

The format of a typical data segment in the stack is illustrated by the following diagram:



Stack Data Segment

The stack mark and the local display consist of control variables that store some state information for the procedure instance. Their use is discussed in Section 5.3.2.

The dynamic parameter descriptors (DPD's) define the actual parameters of the activation (see Section 5.5.1); the procedure's formal parameter identifiers are bound to these cells. These descriptors and the local variables are the environment variables of the procedure instance.

The compiler is able to precompute the amount of storage required for intermediate results that cannot remain in the general registers. Such temporary storage is reserved in a "local stack" allocated within each data segment. All manipulation of indices in this "stack" is done by the compiler, and the temporaries behave as anonymous local variables.

Array elements are stored in an extension of the data segment and are accessed indirectly.

Note that the length of the primary allocation can be computed by the compiler, but the lengths of the secondary allocation and thus of the data segment generally depend upon dynamically evaluated array bounds.

Array storage presents some further complications. The number of elements of an array cannot necessarily be precomputed; moreover, a mechanism for mapping a multidimensional array into linearly addressed storage is necessary. Algol W uses a well-known solution based upon sequential storage allocation and a mapping function which is linear with respect to each subscript (see, for example, [Sattley 61]). In the linearization of multidimensional arrays, the subarrays defined by fixing the value of the final subscript occupy contiguous storage locations, e.g., matrices are stored in column major order. Thus the address of an array element with indices $x_1, \ldots, x_n$ is given by

$$a_0 + \sum_{i=1}^{n} d_i\,(x_i - l_i)$$

where

a_0 is the origin of the storage block containing the values of the array elements.

For each i , l_i and u_i are the declared array bounds and

$$l_i \leq x_i \leq u_i \ .$$

d_1 is the size of each array element.

For $1 \leq i < n$,

$$d_{i+1} = (u_i - l_i + 1)\,d_i \ .$$

This expression can be rewritten as

$$a'_0 + \sum_{i=1}^{n} d_i\,x_i \ , \quad \text{where} \quad a'_0 = a_0 - \sum_{i=1}^{n} d_i\,l_i \ .$$

If the sequences of the coefficients $\underline{d}$ and of the indices $\underline{x}$ are viewed as vectors, address calculation involves the evaluation of an ordinary inner product $\underline{d} \cdot \underline{x}$.

In general, allocation of array storage and computation of the coefficients must be deferred until actual block entry. Thus the storage cell to which an array identifier is bound contains an array descriptor (sometimes called a "dope vector") for that array. Space for the array elements is obtained from the stack also and is contiguous with the other storage for that data segment containing the array's descriptor. Unlike the other declarations of Algol W, array declarations generate executable code. The effect of executing that code is to extend the secondary allocation of the data segment and to fill the descriptor with values of a'_0 and of l_i , u_i , and d_i $(1 \le i \le n)$. Subsequent reference to an array element proceeds as follows: as the indices are evaluated, the value of the inner product is accumulated in a general register b . The virtual origin of the array a'_0 is then added; thus the appropriate address field for a System/360 instruction is $(b, 0)$. Subscript bounds are recorded in the descriptor to allow the option of index validity checking.

If J is some subset of the set of subscript positions I , the address of an element can be expressed as

$$[a'_0 + \sum_{i \in I-J} d_i x_i] + \sum_{j \in J} d_j x_j \ .$$

If the values of x_i are fixed for $i \in I - J$, this expression defines an accessing function for a projection of the original array. Such a projection is used to construct subarray descriptors. The sequence $\underline{d}'$ of coefficients for the subarray is some subsequence of the original $\underline{d}$; if $d'_j = d_i$, then for some $k \ge 1$,

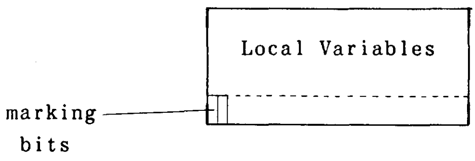
$$d'_{j+1} = d_{i+k} = f_j (u_i - l_i + 1) d_i \ ,$$

where

$$f_j = \prod_{m=i+1}^{i+k-1} (u_m - l_m + 1) \ .$$

Data segments representing elements of $U - S$ are allocated from a pool of storage which is managed using garbage collection techniques (see Section 5.3.3). Segments within the pool can be allocated and freed in an arbitrary order. Those segments occupy fixed locations within the pool and are never moved. The pool itself behaves as a stack of large data areas, called "pages", which grows "downward" toward the stack used for allocation in $S - \{P\}$ and shrinks "upward" as entire pages become free.

Data segments within this storage pool have a simpler format than those in the stack. Their typical structure is illustrated by the following diagram:



Pool Data Segment

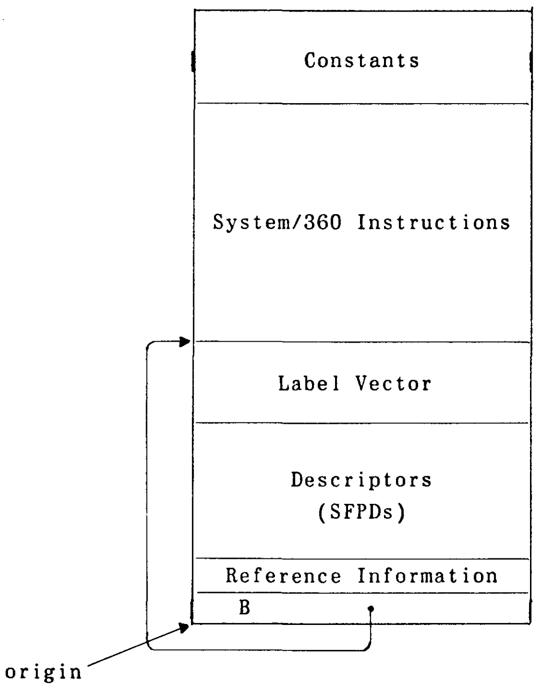
Because the corresponding program texts are null, the control information normally recorded in an activation record is not required. Algol W imposes the further restriction that only simple variables can be declared within a record class; thus there is no secondary allocation. Two Boolean control variables are, however, reserved at the beginning of each data segment in the pool. These "marking bits" are used by the garbage collector.

Note

Because of System/360 addressing and alignment restrictions, these two Boolean variables effectively occupy at least 8 and often 32 bits.

5.2.2. Program Segments

Each program segment corresponds to the source text of a single procedure declaration (and not a record declaration). It contains the machine instructions and constants required to express that text in System/360 object code as well as some additional descriptive information. The format of a typical program segment is shown by the following diagram, where B designates an unconditional branch instruction:



Program Segment

The reference information consists of a pair of constants used by storage management routines (see Section 5.3.3).

The static formal parameter descriptors (SFPDs) describe the fixed attributes of any parameters required by the procedure. They are used to detect mismatched parameter lists in the (relatively rare) cases in which this check cannot be completed by the compiler.

There is a branch instruction in the label vector for every explicit label appearing in the body of the procedure; all transfers to such labels are made indirectly through the branch instructions. This vector simplifies the compilation of intersegment transfers, since an address, in the form of a segment name and offset, can be assigned to each label before the details of the final code are known.

None of the information in a program segment is modified during execution, and recursion requires no special treatment. The code in a single program segment is shared, via pointers, by all procedure instances in a congruence class (cf. Section

3.3); indeed, the sharing of program segments provides a convenient test for congruence.

5.3. The Structure of Activation Records

An execution environment P is provided for compiled Algol W programs by the System/360 hardware and some supporting software. As discussed in the preceding section, every activation record in U - {P} is represented by a data segment. Sequence control (SC) variables are not simple pointers as suggested in Section 3.5 but are pairs similar to the pairs representing label values (cf. Section 3.7.3). The pair consists of the origin of the data segment corresponding to the dynamically enclosing procedure instance and a "return address", which points to the next System/360 instruction to be executed by that procedure instance. Thus the value of x.SC is a pair having the same value as

[D(x), D(x).IA] ,

but activation records do not contain the IA fields introduced in Section 3.3. Those fields are replaced by the second subfield of SC . Note, however, that the instruction address for procedure instance x is recorded in the activation record of $D^{-1}(x)$, not of x . In Algol W, D^{-1} is a well defined function (cf. Section 3.5.2), and thus the same information is encoded in the total set of control variables.

Notes

This scheme was adopted because the System/360 instruction set slightly favors storing the "return address" (using a "store multiple" instruction) locally to a called procedure. In addition, the value of P.SC is [D(P), D(P).IA] , which reflects the fact that D(P).IA is kept in a register of the System/360 hardware, not the activation record of D(P) .

Some investigators of Algol-like languages introduce labels as primitives in their models of control structures (cf. [Johnston 71]). The alternative formulation of SC variables introduced in this section is actually favored by these investigators.

Each access control (AC) variable consists of a single component, which is the origin of the data segment representing the textually enclosing procedure instance. Since a display scheme is used, most operations that are performed to maintain the correct accessing environment involve vectors of AC variables.

The close correspondence between procedure instances and activation records has already been noted. In the Algol W implementation, every such record (except the one for P) is represented throughout its lifetime by a unique data segment which, in turn, can be identified by, and accessed through, its origin. Sometimes it is awkward or unnecessary to distinguish among these classes of objects. When the intended meaning is clear from context, the same name will frequently be used to designate either a procedure instance, its activation record, the corresponding data segment, or the origin of that segment.

5.3.1. The Processor P

The processor's central registers represent part of the activation record of P . In any well-defined state of the computation, P.SC designates the procedure instance which is being executed. One component, the origin of the corresponding data segment, is contained in a reserved memory cell MP , the "mark pointer". The other component is the address in the matching program segment of the next compiled instruction to be executed. It is usually contained in the hardware's instruction address register IA ; during the execution of operations which are simulated by the supporting software, however, that address is stored in a fixed general register or in a reserved memory cell.

A display is used in the implementation of Algol W for the System/360 (cf. Section 3.6.1). Thus P has an access control vector, and that vector is maintained in the general registers. The display element d[i] , where i is the height of the designated activation record, is held in register 13 - i . Every Algol W program is considered to be declared and executed within a standard environment x_0 with height 0 . Register R13 contains a fixed value that points to a data segment corresponding to x_0 and containing the predeclared variables as well as some constants and code sequences that are made available to all compiled programs. If T(P) = P.AC = x and if the procedure instance x has (adjusted) height n , then general registers 13-n through 13 serve as the display, and $T^j(P)$ is contained in register (12-n) + j . Note that register 13-n itself corresponds to P.AC and contains the same value as MP . The display registers are used as base registers in the compiled code and allow direct access to all variables in $T^*(x)$. This accessing scheme is convenient (cf. Section 5.1), relatively simple, and reasonably efficient as implemented for the System/360.

General registers 2 through 12-n are available for expression evaluation; in particular, any computed addresses of array elements or substrings appear in these registers, as do the values of reference expressions. R14 and, when required, R15 contain the base addresses for the program segment containing the code being executed, i.e., the code for procedure instances in the congruence class of x . Such base values are required for branch instructions and for those instructions which access constants embedded within the program segment. R0 and R1 are reserved for system linkage and "scratch" use.

5.3.2. Records in S - {P}

The data segments representing activation records in S - {P} include fields which correspond to control variables and hold state information. Those fields have a standard format. The SC component is represented by the contents of the following two fields, which are parts of the stack mark:

DL, the "dynamic link", contains the origin of the data segment corresponding to the dynamically enclosing procedure instance.

RETA, the "return address", holds the address of the next instruction to be executed by P for the procedure instance designated by DL .

If $x \rightarrow y$, then x.DL = y and x.RETA is the address of an instruction within the code for y .

The AC component of a data segment is expanded into a local display vector DSP . If the (adjusted) height of a procedure instance y is n , with $n > 1$, and if

$$y = x_n \Rightarrow x_{n-1} \Rightarrow \dots \Rightarrow x_0 \quad \text{where} \quad x_i = T^{n-i}(y) ,$$

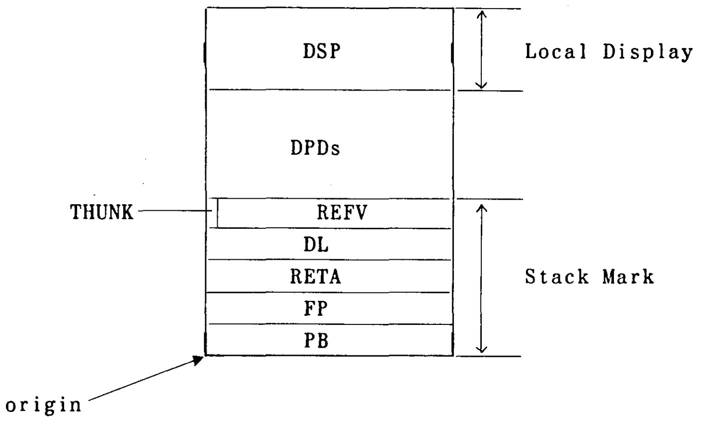
then the local display for y contains n-1 elements, and these are the origins of the data segments representing x_2 through x_n . Since there are System/360 instructions for loading and storing multiple general registers, copying a local display vector to or from P's display vector is quite inexpensive.

Notes

The field DSP is arranged so that x_n.DSP[i] contains x_{n-i+1} . For $n > 2$, x_n.DSP[2] contains x_{n-1} , the origin of the data segment corresponding to the textually enclosing procedure instance. Thus the subfield DSP[2] corresponds to AC , the "static link", and it occupies a standard position within each data segment.

In the execution of any particular program, the data segments at heights 0 and 1, representing the standard environment and the main program respectively, are constant; the origins of these segments are therefore omitted from the local displays.

There are a few other fields of interest that contain control information; they are included in the following diagram, which also illustrates the format of such fields.



State Information

The field PB , the "program base", stores the origin of the program segment containing the compiled code for the procedure instance. In the case of an actual parameter, the code is properly embedded in the program segment.

The field FP , the "free storage pointer", records the next stack location not allocated to the data segment. A data segment with origin x occupies the interval of address space [x, x.FP) . Requests for stack space always reference MP.FP . Thus truncation of the dynamic link chain by resetting MP automatically releases the storage allocated to all deleted data segments.

THUNK is a Boolean field which is set if, and only if, the data segment corresponds to a procedure instance associated with an actual parameter. The storage management routines require this information as well as the information stored in REFV about reference fields within the data segment (see Section 5.3.3).

Note that the control variables described in this section use pointers to impose a substantial amount of structure upon the set of program and data segments. This structure is heavily exploited by the debugging system (see Chapter 6).

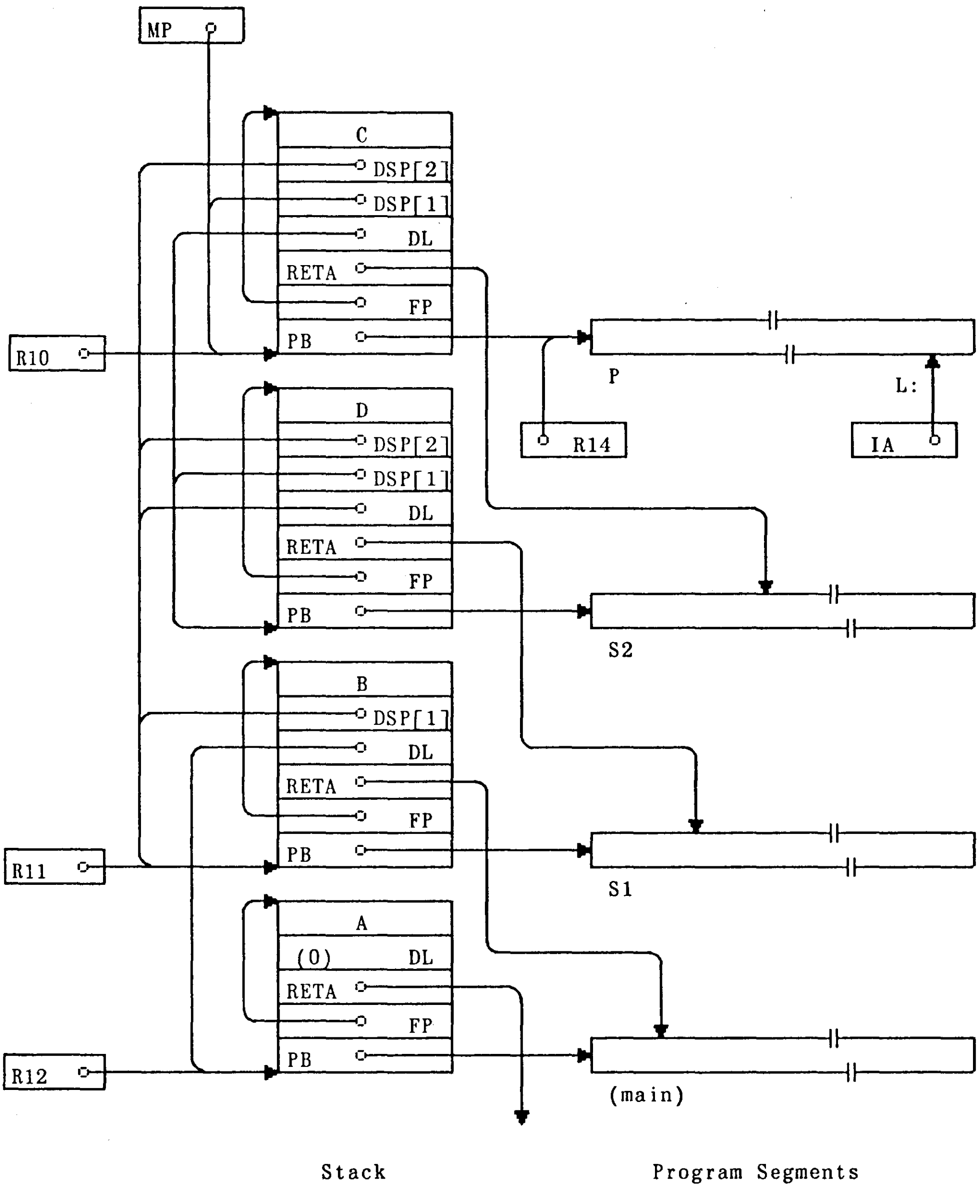


Figure 5-1

Snapshot of Algol W Segments

Example

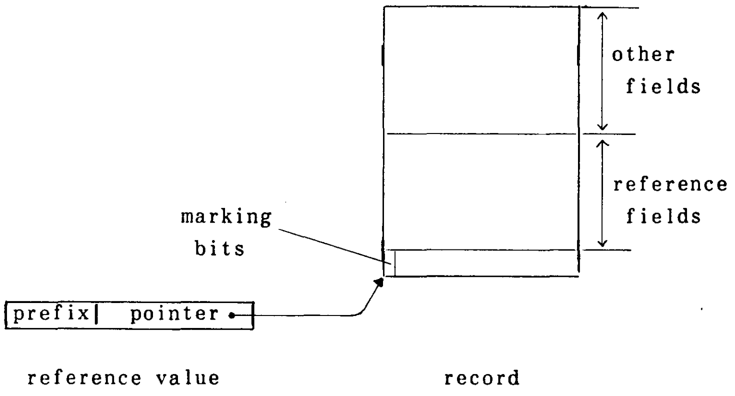
Figure 5-1 is a snapshot of the state of the computation described by the following Algol W program when the label L is encountered:

```
     begin  real A;
S1:    begin
       procedure P;
          begin  real C;
          A := B := C := 3.14159;
L:        end P;
       real B;
S2:       begin  real D;
          P
          end
       end
     end .
```

This figure illustrates most of the data structures described in this section and the relations among them.

5.3.3. Records in U - S

An Algol W record class declaration is analogous to a declaration of a parameterless procedure. A record creator "activates" the "procedure", which returns the origin of (or a "reference" to) a data segment representing a record of the congruence class described by the declaration. The identifiers introduced within that declaration denote field selectors. They are known outside the body of the declaration and can be used with any appropriate reference value to access a variable within a data segment of the correct class. The offsets within such segments are assigned by the compiler so that the variables containing reference values occupy contiguous storage locations at the beginning of the segment. The compiler also assigns to each congruence class a nonzero identifying number which is unique over the set of record classes. Each page in the storage pool contains records of a single class (cf. Section 5.2.1). The value of a reference quantity consists of a pointer to a data segment and a prefix containing the number that identifies the class of that data segment. The null reference has a standard value and a unique prefix.



Record and Reference Formats

The execution environment P includes a global data structure called the *record table.* For any particular Algol W program, the i-th entry in that table contains information about record class i . Each entry consists of three subfields . One contains the length of any record in the congruence class, and a second contains the number of reference variables within each such record. These two fields are constant and are initialized to values supplied by the compiler. The final field is the header of a list of pointers to available records in the class. A routine within P is called whenever a new record is required; it removes a record from the head of the free list, initializes its reference fields to the null value, and returns a properly prefixed pointer.

Whenever the free list for the desired record class is empty, a garbage collection occurs. There are two phases. An initial marking phase systematically inspects all elements of U and marks those records in $U - S$ that must be retained. An element of U is potentially required for subsequent use in the computation only if it can be accessed by a chain of reference variables beginning with a variable local to some $x \in S$. Two cases are distinguished.

If $x = P$, the reference chain begins with an intermediate result in a general register or a local stack cell. The implementation guarantees that the first record in such a chain is "protected". This protection is provided by one of the marking bits. That bit is set in the record designated by each temporary prior to any operation, such as a procedure call, potentially causing a garbage collection, and the bit is cleared before the temporary is discarded or assigned to a variable.

Example

If R is a record class identifier, the first record allocated in the evaluation of the (nonsensical but legal) form

if R = R **then** ...

must be protected prior to allocation of the second.

If x ∈ S - {P} , the root of the reference chain will be encountered in a systematic scan of S - {P} . Scanning begins with the record designated by MP and proceeds by using the DL field to follow the dynamic link chain (cf. Section 3.5.2). The data segments corresponding to records on that chain are arranged so that all local reference variables (including value and result parameters) occupy contiguous fields, as do the descriptors of all local reference arrays. If x is such a segment, there are no local variables if x.THUNK is set; otherwise, x.REFV and the reference information in the program segment designated by x.PB contain respectively the number and origin of the reference variables and arrays.

Note

The counts are maintained locally to each procedure instance so that uninitialized data structures will not be inspected. For example, the call of the procedure in a declaration of the form

reference (R) **array** Q(1::N, 1::F(N))

might cause a garbage collection with Q's descriptor partially uninitialized.

All access paths rooted in the reference variables local to elements of S - {P} or to protected elements of U - S are followed to locate and mark the records to be retained. References can be used to link records in arbitrary ways, and an adaptation of the Schorr/Waite/Deutsch marking algorithm [Schorr 67] is used to explore the access paths systematically. Note that each reference value identifies the class of the record that it designates, and the record table entry for that class contains the information necessary to locate any embedded references.

In the second phase of the garbage collection, the entire storage pool is scanned; unmarked records are returned to the appropriate free list, and the bits used to mark accessible records are reset. If the free list for the desired class remains empty after the garbage collection, a new page is initialized and added to the pool, and a new free list is constructed.

5.4. Control Transfers

Control transfers in the System/360 implementation of Algol W occur approximately as outlined in Section 3.7. The compiler does, however, perform an analysis of the entire program so that the overhead associated with such transfers can be somewhat reduced. It attempts to avoid allocating and initializing variables that store closures and label values. In many cases, such data structures are synthesized from display elements and from constants embedded within program segments just when they are needed for control transfers. Some effort is also made to minimize the bookkeeping instructions required to maintain the display. The special cases that arise in this analysis do not substantially affect the design of the debugging system, but some knowledge of them is necessary for understanding a few of the implementation details discussed in Chapter 7. Note that all transfers of interest in this section occur between elements of $S - \{P\}$, i.e., between procedure instances represented by data segments in the stack.

5.4.1. Classification of Procedures

For purposes of code generation, the Algol W compiler classifies all procedures as *open* or *closed*. An open procedure can be thought of as one in which the declarations in the formal parameter list are "open" for inspection by every caller of the procedure; if those declarations can possibly be "hidden" from any caller, the procedure is closed. More precisely, a call operation using the closure c is open if it occurs within the scope of the declaration creating c and if c is named by the identifier introduced in that declaration. Any other call is closed. A congruence class of closures or, by extension, procedure instances is open if no call operation using a closure in that class can possibly be closed; otherwise, the class is closed. The distinction between open and closed procedures is useful because closed procedures are rare and significantly better code can be generated for open procedures, especially with respect to parameters (see Section 5.5).

Note

"Open" is sometimes used to mean that the call is actually eliminated by using the body of the procedure as a template for macro expansion. Only a call that is open by the above definition can possibly be so expanded, but such substitution is not used in the implementation of Algol W.

The only variables to which closures can be assigned in Algol W are those introduced by the declarations of certain formal parameter identifiers. If a closure is not assigned to such a variable, it can be used only in open calls.

Conversely, the closed calls are exactly those in which the closure is specified by a formal parameter identifier. Thus the classification algorithm is trivial. The compiler marks a congruence class as closed if the corresponding procedure identifier ever stands alone as an actual parameter. All other classes are open. Classification is completed before any code is generated.

Example

Consider the following Algol W program:

```
        begin  integer M;  real Z;
        procedure P (real procedure F);
           begin real X;
    L1:    X := F(M)
           end P;
        real procedure R (real value X); 1/X;
        M := 10;
    L2: Z := R(10);
    L3: P(R)
        end .
```

The procedure call labeled L1 is closed, the call at L2 is an open call of the closed procedure R , and the call at L3 is an open call of the open procedure P .

Afterthought

The classification of a procedure as open or closed affects the code generated for both its body and any open calls of it. Although very few Algol W programs use many closed procedures, it probably would have been worthwhile to provide two entry points for each procedure text, one used by open calls and the other used by closed calls. Of course, no code need be generated for an entry point known to be unused.

5.4.2. Procedure Call and Entry

The instruction set of the System/360 requires that a call operation be implemented as a sequence of simpler operations, much as suggested in Section 3.7.1. The initial part of that sequence is specified by inline instructions generated for the call; the remainder, by instructions appearing in a prologue of each procedure text, called the *instantiation code* of the procedure.

Suppose that procedure instance x executes a call operation in which the argument is the closure c . It is always possible to identify a pair of general registers r_1 and r_2 having the property that [c.tp, c.ac] = [r_1, r_2] immediately prior to the call. The value of c.tp is an address within a program segment, and c.ac is represented by the origin of a data segment. In the case of a closed call, an existing closure which is the value of a formal parameter is simply copied into a standard pair of registers so that c = [R3, R4]. To allow checking of the correspondence between formal and actual parameters, the actual transfer in this case is done indirectly through a support routine provided by software in the execution environment P . Details of such indirect transfers are provided in Section 5.5.4.

In an open call, the closure is constructed dynamically. The value of c.tp is the origin of a program segment, the identity of which is known. The absolute address of that segment is treated as a literal (i.e., as a System/360 address constant) within the code for x . In preparation for the transfer, it is loaded into the program base register R14 . Since c.ac = $T^m(x)$ for some $m \geq 0$ in an open call, its value is available in a display register r . If c is an element of a closed congruence class, the value of r is copied into a standard register R4 so that c = [R14, R4]. If c is open, however, c.ac need not be moved to a standard location and c is represented rather implicitly by [R14, r] . The actual transfer is initiated in either case by a "branch and link" instruction specifying c.tp (R14) as the destination address.

The instantiation code entered by the branch instruction allocates a data segment to represent a new procedure instance y and initializes the fields in that segment's stack mark. The origin of the segment is copied into MP and into display register 13 - h(y) , where h(y) is the height of y . If y belongs to a closed congruence class, the display is completed by copying the h(y)-2 elements of R4.DSP , the local display of T(y) , into display registers 13 - (h(y)-1) through 11 (cf. Section 5.3). This is unnecessary for an open procedure, since it is guaranteed that $T(y) = T^m(x)$ for some $m \geq 0$ and thus that the two procedure instances share display elements (see Section 3.7.1). In both cases, MP.DSP is initialized with the values of registers 13-h(y) through 11 . Prior to that initialization, the control variables of y are not necessarily consistent; thus the instantiation code cannot strictly be considered a part of y .

5.4.3. Procedure Exit and Return

Return operations must also be implemented as sequences of simpler operations; again, these sequences follow the general pattern suggested in Section 3.7.1. If $y \rightarrow x$, part of the code for the transfer forms an epilogue to the text of y . That code copies y.DL , the origin of the data segment for x , into MP and into a standard register R2 . It then transfers by branching to the location in the program text for x given by y.RETA . Code at that location uses R2.PB and R2.DSP to reload the program base register and any necessary display registers. If the call operation that created y was open, then $T(y) = T^m(x)$ and only the m display registers corresponding to x , T(x) , ..., $T^{m-1}(x)$ are restored. Note that resetting MP releases the storage used by the data segment for y .

The code following a return from a function procedure must retrieve the value computed by the procedure. The location of that value is determined by the declared type of the procedure and the number of parameters. Function procedures with at least one parameter return the value in a central register or set of registers appropriate for the declared type, and the location depends only upon that type. String procedures are exceptions; since there are no hardware accumulators for strings, a pointer to the string is returned in a general register (R3) . All values of parameterless function procedures are also returned in a main storage location addressed by R3 , since permissable uses of such procedures overlap those of name parameters, and the latter must return addresses (see Section 5.5). Storage for the value designated by R3 can be part of the data segment for y , which is released by the transfer operation; therefore, code in x uses the value or copies it into a local variable prior to any operation potentially requiring storage allocation.

5.4.4. Transfers to Labels

Algol W does not provide label variables, switches, or label parameters (although the effect of the latter can be obtained by using goto-statements as parameters). Thus transfer to a label from procedure instance x is possible only if the label is declared in $T^m(x)$, $m \geq 0$. If $m > 0$, the code generated for the transfer resets MP to the value of the display register containing the origin of $T^m(x)$, loads the program base register R14 from MP.PB , and branches to a label vector entry in the program segment for $T^m(x)$. Note that the display need not be reset and that data segments removed from *S* by the transfer are automatically deallocated by resetting MP ; no elaborate "go to interpreter" is required (cf. Section 3.7.2).

5.5. Parameters

Algol W has inherited from ALGOL 60 most of the subtleties of name parameters that are demonstrated by the examples in Section 3.8. The problem of type conversion is exacerbated by the relatively large number of interconvertible types and the variety of parameter transmission mechanisms. In an open call of an open procedure, all relevant attributes of the formal and actual parameters can be deduced by the compiler; it can verify compatibility and generate any code necessary for type conversion. In a closed call, however, the compiler cannot always deduce the precise attributes of the formal parameter to which a particular actual parameter corresponds. The allowable assignments involving value parameters (upon procedure entry) and result parameters (upon procedure exit) potentially require type conversions in which the types must be determined dynamically. This situation, which arises nowhere else in Algol W, requires that the representations of the actual parameters in a call of a closed procedure include an encoding of their types.

Example

Consider the (closed) procedure call P(V) . If the type of P's formal parameter is, e.g., T **value** , then the actual parameter V can have any type T' such that

domain(T') $\subseteq$ domain(T) .

Alternatively, the type of the variable V can be fixed; let it be some type T . Then a valid (although unlikely) Algol W program, similar to the second ALGOL 60 example in Section 3.8, can be written in which the code compiled for the call must work correctly for actual procedures having formal parameters with any of the following attributes:

T' **value** , for all T' with domain(T') $\supseteq$ domain(T)

T

T **procedure**

T' **result** , for all T' with domain(T') $\subseteq$ domain(T)

T' **value result** , for all T' with domain(T') = domain(T) .

These problems are resolved approximately as described in Section 3.8. Consider an actual parameter appearing in a call performed by procedure instance x to create procedure instance y . In the general case, the value assigned to y's formal parameter is a composite data structure. One component is a type code; the other, a closure c constructed as the procedure value of the actual parameter (see Section 3.4.2). The value of c.tp is a pointer to program text constituting a thunk, and c.ac = x . The thunk evaluates the parameter in the context of x to obtain a pointer value. The type code indicates the type of the value designated by the pointer. Each access to the parameter within y thus requires activation of a procedure. A pointer value is always returned; access to the operand value of the parameter is through the pointer. Transmission of a parameter by value or by result is completed by the entry or exit sequence of the code for y ; in effect, an assignment involving the actual parameter and an implicitly declared local variable of y is inserted by the compiler at the beginning or end of the procedure body (cf. [Wirth 66b, Section 5.3.2.2]). The assignment is done semi-interpretively by supporting software which additionally examines the type code of the actual parameter and performs any required conversion. A similar mechanism is used for array parameters, which are actually transmitted by copying the value of an array descriptor into local storage. Thus a formal array is accessed by "descriptor value" rather than by name.

A scheme with this much generality is quite expensive. The Algol W compiler therefore analyzes each call and each actual parameter to discover special cases for which a more efficient mechanism can be used. The analysis is moderately complex. Again, few of the details significantly affect the design of the debugging system, but knowledge of some of them is required for understanding parts of Section 7.5. The compiler attempts to exploit the empirical observations that most calls are open and that most actual parameters are simple in structure. The two primary optimizations are the following:

For any actual parameter with a certain syntactic form, the pointer value of the parameter is *a priori* constant. Simple variables and constants are examples of such forms. For such a parameter, call by name and call by address are equivalent. The closure is therefore replaced by the pointer value that it would return upon each invocation. The pointer value itself is constructed by the calling sequence. A tag field is also added to the value of each actual parameter so that closures and pointers can be distinguished.

In an open procedure call, code to convert each actual parameter passed by value to the type of the corresponding formal parameter is generated by the compiler as part of the calling sequence. All assignments to the implicitly declared locals in an open procedure are therefore done by inline code which performs no validity checking or conversion. Note, however, that assignments involving parameters with the attribute **result** or **value result** must still be done interpretively, since the conversion in these cases is part of the text of the called procedure, not the calling sequence.

Afterthought

Even with these improvements, the transmission of a value parameter to an open procedure can require a call of a thunk to obtain the initial value. Since value parameters are known to be very common, it would be worthwhile to perform the evaluation of the actual parameter as part of the calling sequence whenever the call is open. Such preevaluation would be particularly attractive in conjunction with separate entry points for open and closed calls as suggested in Section 5.4. A call of y using the open entry point could transmit any value parameters by directly assigning the operand values of the actual parameters to the corresponding local variables of y .

5.5.1. Dynamic Parameter Descriptors

In the Algol W implementation, the value of a formal parameter is a data structure called a *dynamic parameter descriptor* or *DPD*. Such a descriptor has the following format:

D4	D5
D0	D1

Dynamic Parameter Descriptor

Field D0 is a tag field and contains three attribute bits, which are called the P-, Q-, and X-bits. The P-bit governs the interpretation of the remaining fields. If it is set, the DPD contains a closure which is the procedure value of the actual parameter; otherwise, the DPD contains the address which is the pointer value of that parameter. The other bits are used only for dynamic validity checks. The Q-bit indicates whether the actual parameter is an updatable variable. The X-bit is inspected only if the formal parameter is declared to be a procedure, and it is set if the actual procedure itself requires parameters.

When the actual parameter is an explicit constant or is the identifier of a simple variable (not requiring type conversion), an array, or a control variable, the P-bit is not set. The actual parameter can be accessed by simple indirection; its address is contained in field D1 , and field D5 is unused. For all other actual parameters, the P-bit is set and the DPD contains a closure c , with c.tp in field D1 and c.ac in D5 . Field D4 is used when necessary to encode the type or (for a string) the length of the actual parameter so that any necessary interpretive conversion can be performed (see Section 5.5.3).

5.5.2. Thunks

The code for a thunk (or "implicit subroutine") generated for an actual parameter is embedded within the program segment containing the text for the call. Each call is preceded in the object code by the sequence of thunks required to describe its actual parameters; when the sequence is nonempty, a branch is inserted to jump around its elements.

There are three classes of thunks, and they correspond to actual parameters that are expressions, statements, and procedures. To simplify the treatment of certain puns, a parameterless procedure standing as an actual parameter is always classified as an expression or statement. Execution of a thunk in one of the first two classes creates an activation record z in $S - \{P\}$. The code therefore includes a prologue specifying operations comparable to those performed by the instantiation code of an explicitly declared procedure (see Section 5.4.2). Both the stack mark and local display are initialized. In particular, the field z.THUNK is set. Note that z.PB contains the origin of the program segment within which the thunk is embedded, not of the thunk itself. The thunk is considered to be a closed procedure with respect to display updating, but its initialization code can otherwise be simplified somewhat because there are no parameters and no local variables.

The body of a thunk generated for a statement consists of the code for that statement. A thunk generated for an expression returns a pointer value in a standard register, R3 . There are two cases.

> If the expression is a variable, the pointer value of the variable is computed by the body of the thunk, and the thunk returns that pointer value. The variable is necessarily local to $T^m(z)$ for some $m > 0$ and has a lifetime at least equal to that of D(z) , the procedure instance which requires access to that variable.
>
> For any other expression form, code in the thunk assigns the value of the expression to a location in the local stack of z . The value returned in R3 is then a pointer to a "variable" which is an artifact of the implementation and has shorter lifetime than D(z) . The Q-bit in the DPD for such a thunk is set to indicate that the pointer can only be used to obtain an operand value.

A thunk for either an expression or a statement terminates with a standard epilogue; indeed, such thunks are designed to be used interchangably with parameterless procedures (cf. Section 5.4.3).

A thunk generated for a procedure identifier does not create an element of S - {P} ; instead, it uses the closure c created for itself and used to enter the thunk to construct a closure c' for the actual procedure. It then transfers to a system routine that checks parameter correspondence and invokes the actual procedure using c' . That procedure returns directly to the caller of the thunk. Since c.ac points to the activation record for the procedure instance in which the actual procedure identifier stands as a parameter, the closure c' can be constructed using a constant embedded in the program text at c.ac.PB and an entry from the local display c.ac.DSP (cf. Section 5.4.2).

5.5.3. Parameter Transmission

In preparation for a call operation, procedure instance x must construct a sequence of DPDs describing the actual parameters. The facts that data segments are contiguous and are allocated from a single stack are used to simplify transmission of the DPDs to the new procedure instance. Since the stack mark consists of a fixed number of components, the offset of the field in a data segment corresponding to formal parameter i depends only on the value of i , not upon the congruence class of the data segment (cf. Section 5.3.2) or the declared types of the formal parameters. Since the origin of the newly created data segment will be x.FP , the DPDs are constructed in sequence and assigned to storage cells with the correct offsets relative to x.FP . The instantiation code for the new procedure leaves these cells unaltered; they become the fields corresponding to the formal parameters in the new data segment and are correctly initialized.

Note

This strategy is safe because construction of the sequence of DPDs never requires allocation of additional storage on the stack. In general, the code must include a test for stack overflow. Storage is arranged so that space for a stack mark and at least 8 DPDs always remains at the top of the stack, and the overflow test can almost always be deferred until entry into the procedure's instantiation code.

If an actual parameter is itself a formal parameter identifier, an existing DPD is simply copied. Otherwise, a DPD for each actual parameter is synthesized in preparation for a call. The code for doing the synthesis initializes the D1 field of the DPD with the absolute address of a variable or constant if call by address is used, and to the entry address of a thunk if call by procedure is indicated. In the latter case, the display register for x , 13 - h(x) , is assigned to the D5 field. The assignment to D1 automatically sets D0 to the most common combination of PQX bits; other combinations are set explicitly. The initialization of field D4 depends upon the attributes of both the call and the called procedure. The compiler must guarantee that this field is set correctly if there is any possibility that it will be used. A type code for the actual parameter is assigned to D4 in the following circumstances (see Section 5.5.5):

In an open call of an open procedure where the type of the formal parameter (e.g., **real result**) forces interpretive conversion.

In an open call of a closed procedure where the type of the formal parameter (e.g., **real value**) potentially requires interpretive conversion.

In a closed call for every actual parameter with a type (e.g., **real**) that is assignment compatible with at least one other type.

The value of D4 cannot possibly be relevant in any other case.

Example

The data strucures built in preparation for executing the procedure call

Q(Y, A(I)+Z, 3)

are summarized in Figure 5-2. Note that an actual parameter is updatable iff the P- and Q-bits agree. The snapshot depicts an open call of an open procedure. If the call were otherwise, each of the D4 fields would contain a type code.

5.5.4. Access to Parameters

The code to access a parameter that is an expression copies the DPD into a standard pair of registers, [R3, R4] . It then interrogates the P-bit. If that bit is not set, R3 contains a pointer to the actual parameter. Otherwise, the low-order bits in the register pair [R3, R4] are a closure; that closure is activated, and its execution terminates with a pointer value loaded into R3 . In either case, the absolute address of a storage location is loaded into the standard register R3 , and the appropriate System/360 instruction field for referencing the value of the actual parameter is (3, 0). A thunk for a statement is accessed using the DPD as a closure in a similar way, but no value is returned.

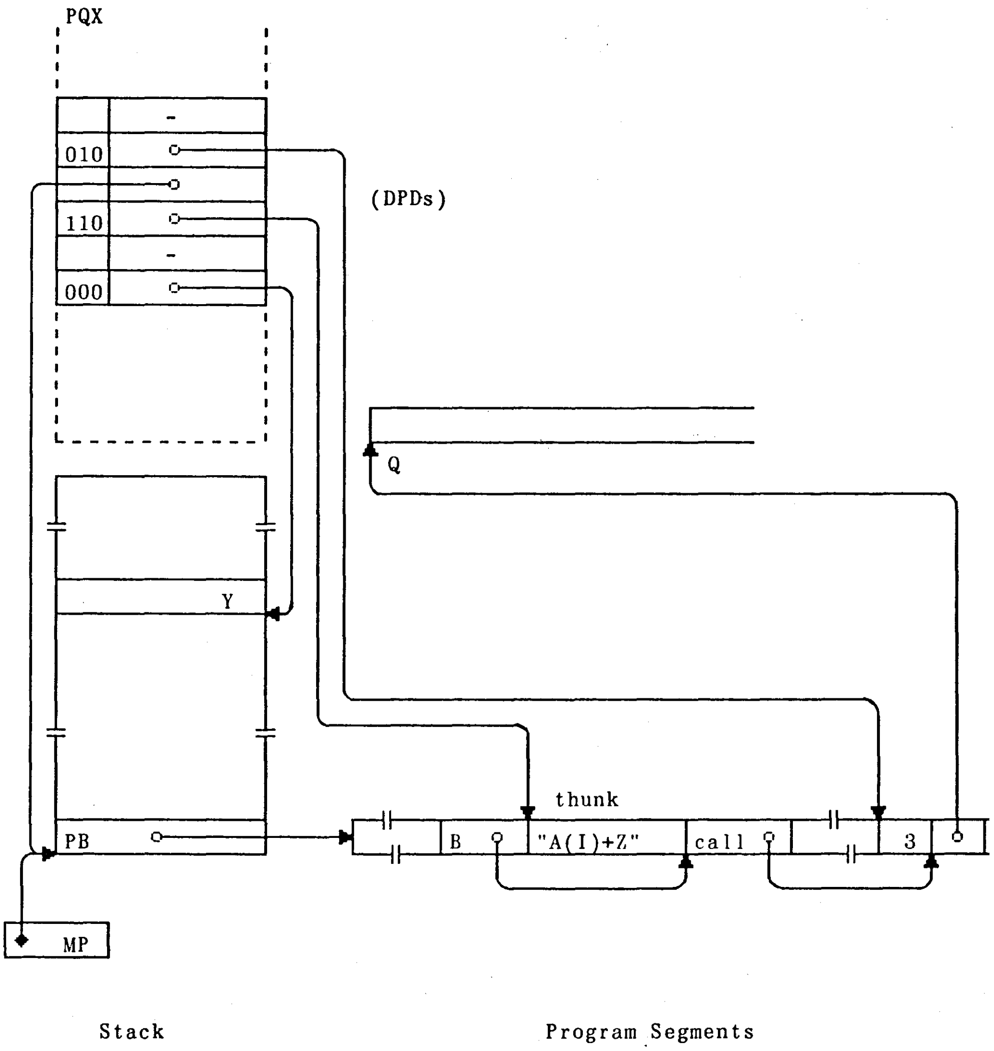


Figure 5-2

Snapshot of Algol W Call Operation

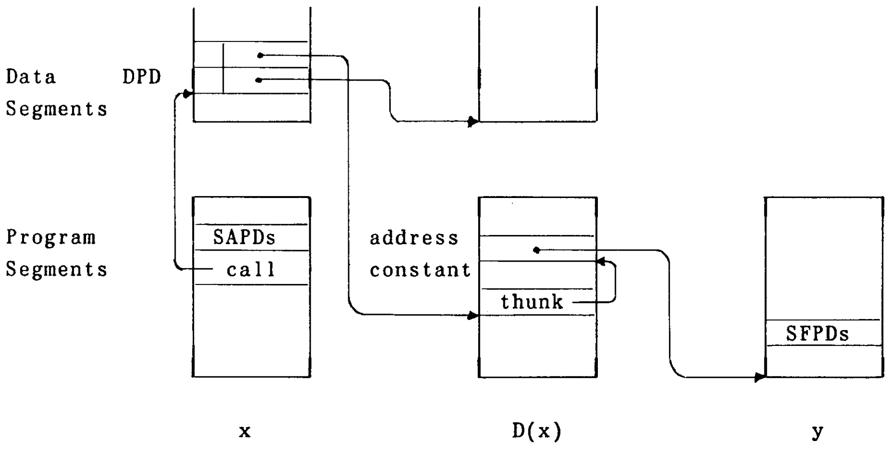
Note

When relevant, the test of the P-bit is combined with a test of the Q-bit using the System/360 "test under mask" instruction. The latter test detects any attempt to assign to an expression that is not a variable.

If an actual parameter is accessed to obtain an operand value, that value can be in storage which is released when the thunk terminates, and code following the access must fetch the value immediately. Such immediate use is not necessary if the pointer value is the address of a legitimate variable (see Section 5.5.2).

A thunk for a procedure is accessed after the parameter list for the actual procedure has been constructed in the usual way. A copy of the DPD is loaded into [R3, R4] just as it would be for an expression. The "branch and link" instruction that initiates the transfer through the thunk is followed by a sequence of "static actual parameter descriptors" (SAPDs) describing the attributes of the actual parameters. The system routine entered indirectly through the thunk matches the SAPDs with the static formal parameter descriptors (SFPDs) of the actual procedure (see Section 5.2.4). These descriptors are located using the return address and the entry address in the closure constructed by the thunk respectively. If the parameters' attributes are compatible, the system routine completes the transfer.

The following diagram illustrates the data structures involved in a closed call of y from procedure instance x :



5.5.5. Value and Result Parameters

If a formal parameter of some procedure is declared with either of the attributes **value** or **result** , the compiler introduces a local variable of that procedure with the type of the formal parameter. Within the body of the procedure, all references to the formal are to that local variable, not to the DPD for the actual parameter. The attribute **value** causes the compiler to insert at the beginning of the procedure's text an assignment of the actual parameter's operand value to the local variable. The attribute **result** causes the compiler to insert at the end of the text an assignment of the local variable's value to the variable designated by the pointer value of the actual parameter. In both cases, the actual parameter is accessed in the normal way through a DPD.

In any open procedure, the operand value can be assigned directly to the local variable; no conversion is required. In a closed procedure, the assignment is done by a semi-interpretive support routine. Code within the body of the procedure obtains the pointer value of the actual parameter in the normal way. The pointer is left in R3 . Subsequent code inserts the type of that parameter, obtained from the D4 field of the DPD, and the type of the formal parameter into a standard pair of registers and calls the support routine. That routine places the address of the converted value into R3 . Finally, the assignment is completed by inline code. Note that the value of R3 is often unchanged by the support routine; this is always true when the instance of the closed procedure was created by an open call.

Assignment of a result parameter is treated similarly, but interpretive conversion is sometimes used even for an open procedure. In this case, the pointer value of the actual parameter is saved, and the address of the local variable is loaded into R3 prior to the call of the conversion routine. That routine also supplies the length of the converted value so that an assignment moving the correct number of bytes can be fabricated for inline execution (using a System/360 "execute" instruction).

Afterthought

All interpretive conversions by open procedures could be avoided if each call of such a procedure specified the location of a temporary for receiving any result requiring conversion and if code following the call performed the final conversion and assignment.

5.6. Other Features of the Object Code

Most of the details of the object code are conventional and not relevant to the design or understanding of the debugging system. Outside of expressions, most code sequences generated by the compiler are derived from standard code templates. A few such templates are associated with each syntactic construct of the language; the compiler selects the most appropriate template on the basis of any special cases it can detect. This section mentions some miscellaneous details of interest.

5.6.1. Expressions and Assignments

The order of evaluation of operands in arithmetic expressions and in assignments is chosen by the compiler to minimize accumulator requirements. The basic selection algorithm is well known (see, e.g., [Nakata 67]), but modifications have been introduced to reflect asymmetries and anomalies in the System/360 instruction set. The result is that the order in which variables are mentioned and procedures are called in the object code is not necessarily the order of appearance in the source code.

Logical (i.e., Boolean) expressions are evaluated from left to right, and the evaluation of any operand not required to determine the value of the expression is bypassed. These expressions are most frequently used in clauses which determine the flow of control. When they are so used, actual generation of logical values is not required and is not implemented in the object code; the result of evaluating a logical expression is to select between branch targets.

Evaluation of a string expression does not force the string value into a hardware accumulator; such values are, however, formed in an anonymous area of the local stack unless source and destination strings are guaranteed not to overlap and no length conversion is required.

Record creation with initialized fields is implemented by generating code to create an uninitialized record and then to perform a series of assignments to the record fields.

5.6.2. Computed Addresses

In most cases, the operands of machine instructions are specified by (b,d) addresses, where b contains the origin of a data segment and d is a displacement. Absolute addresses of cells within data segments are generated for certain constructs, however. In addition to array elements, certain function procedures, and actual parameters as described in preceding sections, they are computed in the following circumstances:

In the code for accessing substring variables, the substring offset is loaded into an index register and then the absolute address of the first character of the substring is generated using a System/360 "load address" instruction.

If the value of a reference expression is computed in preparation for a record field access, there are certain unusual circumstances in which the absolute address of that field is computed before the field is accessed.

In fetching operands, computed addresses are used immediately after they are formed in the register. If the computed address is that of a variable on the left of an assignment operator, however, it occasionally happens that the address must be stored in the local stack and later reloaded.

5.6.3. Implicit Branching

Conditional and iterative statements are implemented by standard techniques. The implied branch instructions have destinations within the compiled object code; they do not use label vector entries. Case clauses are compiled into indexed branch instructions; the target of the indexed branch is a second branch instruction which locates the appropriate case element.

5.7. The Algol W Compiler

The Algol W compiler accepts Algol W source text and converts it directly to System/360 machine code. The structure of that code and the environment in which it is executed have been described in the preceding sections. This section is an abbreviated outline of the organization of the compiler. Once again, details not relevant to the design of the debugging system are omitted.

The compilation algorithm uses three passes. The first pass is a scanner which constructs a compact linear representation of the source program. Declarations are also recognized, and information extracted from them is used to build a set of symbol tables. The second pass parses the program string and produces a tree-structured representation of the "abstract syntax" of the program. The final pass generates reentrant System/360 machine code. The second and third passes perform a number of local optimizations, but no global optimization is attempted.

Communication among the passes is based upon shared tables, which are accessed randomly, and upon intermediate encodings of the program, which are written and read sequentially. The overall flow of information is shown by the following diagram:

$$\begin{tikzcd}
 & {} \arrow[d, "\text{source text}"'] \\
 & \text{Pass 1} \arrow[dl, dashed] \arrow[d, "\text{program string}"'] \\
\text{symbol tables} \arrow[r, dashed, leftrightarrow] \arrow[dr, dashed] & \text{Pass 2} \arrow[d, "\text{tree code}"'] \\
 & \text{Pass 3} \arrow[d, "\text{machine code}"'] \\
 & {}
\end{tikzcd}$$

In the student-oriented version of the compiler, each of the shared data structures remains in main storage from the time it is created until the time it is discarded. All of the major structures are designed to be relocatable, and the compiler includes a set of routines for reallocating and repacking working storage when necessary.

5.7.1. Pass One

Syntactic analysis requires information about the attributes of identifiers to make context-sensitive parsing decisions and also uses such information to perform related semantic actions, such as type computation. Since the use of an identifier can precede its declaration, a preliminary scan is required. The main function of the first pass is to perform this scan. It recognizes declarations and extracts information from which a skeletal symbol table is constructed. This pass must perform a detailed scan of the input text; therefore, it has also been assigned the task of lexical analysis, the result of which is a compact and convenient internal encoding of the source program.

The first pass is basically a scanner implemented as a finite state automaton and augmented with a stack for recording information about the nested block structure. The lexical scanner performs the usual functions of removing blanks and comments, recognizing multicharacter representations of basic symbols, and the like. It delimits and syntactically checks constants but does not convert them to internal format. The scanner also recognizes members of the syntactic class ⟨identifier⟩ . Each textually distinct identifier encountered is assigned a unique internal identifier number. The correspondence between identifier numbers and character strings is recorded in a pair of tables, the identifier directory and identifier list. The former is a vector indexed by the identifier number; each entry is a pair consisting of the length of the associated character string and its origin in the identifier list. The latter table is just the concatenation of all distinct character strings used as identifiers. Note that the identifier number, identifier directory, and identifier list (as well as the hashing functions used by the scanner) are independent of the block structure.

The lexical scanner also builds a linear program string representing the original source program. Entries in this string have one of the following formats:

Identifiers are represented by a distinguishing code followed by the identifier number.

Constants are represented by a type code followed by the constant itself, delimited by internal quotation conventions.

All other basic symbols of the Algol W language are replaced by single-character (8 bit) internal codes.

The source coordinates can be computed as the program string is scanned; they are not explicitly recorded.

Note

Various studies have indicated that single-character identifiers and constants are especially common. The author's own measurements indicate that such identifiers account for 30% to 75% of all identifier usage; other measurements have given estimates of approximately 50% [Wichmann 70]. To save storage space, single-character identifiers and single-digit numbers represent themselves in the program string.

The declaration processor is activated by the recognition of certain basic symbols. A new scope is introduced for each occurrence of the following symbols:

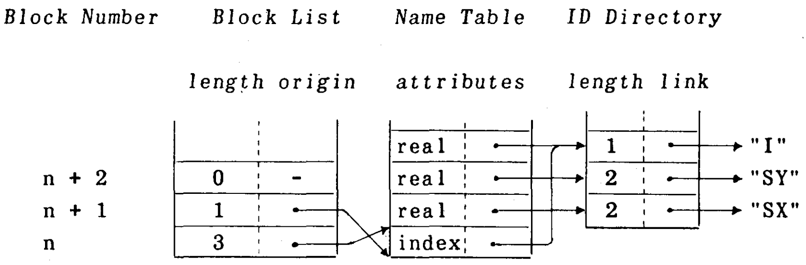
begin , **procedure** , **for** .

Each scope is identified internally by a sequentially assigned block number. When the scanner is in declaration mode, the number and declared type attributes of each encountered identifier are entered onto a stack. When the scope is closed by a matching delimiter, all stack entries for that scope are moved to a name table, and the origin and length of the corresponding name table segment are recorded in the block list, which is indexed by the block number. The scanner does not attempt to record the nesting of scopes. The following program fragment provides a simple example of the data structures built by pass one.

Example

```
...
begin
real SX, SY, I;     SX := SY := 0;
for I := 1 until N do
   begin
   SX := SX + X(I);   SY := SY + Y(I)
   end
end
...
```

The corresponding tables are shown below.



Note that name table entries for any given scope are contiguous and retain the order imposed by the source program.

5.7.2. Pass Two

The second pass parses the program string, computes and checks type information, and builds a tree-based representation of the source program which explicitly displays the syntactic structure. With the exception of some context-sensitive restrictions upon the use of identifiers, valid sentences of Algol W can be described by phrase structure grammars satisfying various sets of restrictions that guarantee unambiguous and efficient parsing. Initially, a simple precedence grammar and parser [Wirth 66a] were used. An SLR(1) grammar and the parsing algorithm of Eve and Anderson [Anderson 73] were subsequently adopted. The important features of both parsing algorithms are the following:

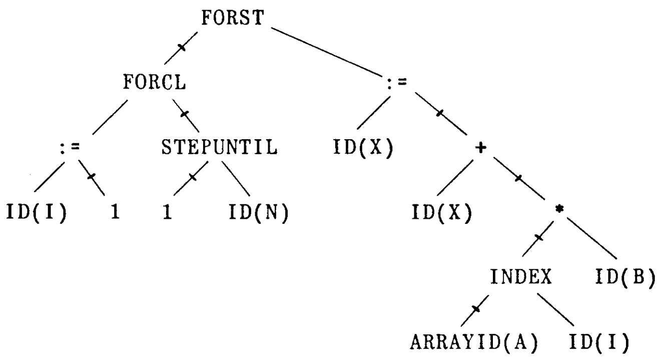
Parsing is "bottom-up"; reducible phrases are delimited by decision functions based upon the state of the parser and the local context, and such phrases are replaced by the corresponding nonterminal symbols as they are encountered.

The parse is completely unambiguous; local decisions are always correct and backup is never required. Thus any necessary "semantic actions" can be performed as soon as a phrase is recognized.

The tree is actually represented as a forest of binary trees. A separate tree is generated for each procedure declaration, and the trees can be processed independently. In general, each nonterminal node is labeled by the name of an operator, and its subtrees represent the operands. Thus the tree associated with the statement

for I := 1 **step** 1 **until** N **do**
X := X + A(I)*B

is



Terminal nodes of the tree normally contain pointers to table entries. Identifier nodes contain the indices of name table entries. Construction of these nodes, as well as type determination and checking, require that identifier numbers in the program string be mapped into name table indices. Thus the second pass must reconstruct the nesting of scopes, and it does this by maintaining an analog of the run-time display. When a new scope, corresponding to a block, a parameter list, or a for clause, is opened, the height of the compile time display is increased by one, and the block list entry for the scope is added as a new display entry. Closing a scope results in decrementing the height and deleting a display entry. The display is used for selecting the name table segments to be searched when an identifier is encountered. If the identifier is located in the segment indicated by the i-th display entry, that occurrence of the identifier is said to have a compile-time height of i .

The second pass computes the adjusted height (or run-time height) of each congruence class, and it allocates a field within the activation records of the appropriate congruence class for every state variable and every explicitly declared identifier. The latter assignments, which are entered into the name table, establish the (b,d) addresses which will later be used in machine instructions. Because some scopes are merged or eliminated in the run-time structure (see Section 5.2.2), the compile-time height of an identifier is not necessarily identical to the run-time height of the corresponding activation record.

An explicit link to the first son is retained in each nonterminal node in the tree structure; the link to the second son is implicit. The advantage of the explicit link is that either subtree can be traversed first in subsequent processing. The second pass, in fact, places computed switch values in each nonterminal node; these direct later traversal so that, for example, the number of accumulators required for expression evaluation is minimized. In the example above, switch settings are indicated by ticking the edges leading to the first subtrees to be processed. The main consequence of these switches is that left-to-right evaluation within a flow unit cannot necessarily be assumed.

5.7.3. Pass Three

The third pass traverses the tree structure constructed by the preceding pass and generates appropriate System/360 instructions. The algorithm for traversing a (sub)tree is the following:

Exit if the root is a terminal node; otherwise, inspect the tree switch to select the first subtree.

Traverse that subtree.

Visit the root to generate object code associated with the first operand (e.g., to load it into an accumulator) and to select the second subtree.

Traverse the second subtree.

Visit the root to generate object code associated with the second operand and operator.

Information about any intermediate results is maintained in a separate operand stack; there are also several auxiliary stacks and deques used for address fix-up and register allocation information. Note that most of the work is done in this pass as an operator inspects its operands, not as the operands themselves are encountered in the tree traversal.

Note

The intention was to simplify the treatment of the special cases which often arise in connection with terminal nodes, but any benefits of such a scheme have proved rather elusive.