The Structure of Programming Languages

BERTRAM RAPHAEL
Stanford Research Institute, Menlo Park, California

In this paper the major components of every programming language are identified as: (1) the elementary program statement, (2) mechanisms for linking elementary statements together, (3) the means by which a program can obtain data inputs. Several alternative forms of each of these components are also described, compared and evaluated. Many examples, frequently from list processing languages, illustrate the forms described.

The advantages, disadvantages and factors influencing the choice of a form of component for a language are discussed, and the paper concludes with the suggestion that programming languages evolve toward one which will permit all the most convenient ways of structuring programs, organizing systems and referencing data.

1. Introduction

The way in which a programming language is structured can strongly affect the efficiency and modus operandi of a programmer. The three purposes of this paper are (1) to identify the major components of programming languages, (2) to describe alternative forms available for implementing each of these components, and (3) to compare and evaluate the various forms.

The computing community has tried to attach to certain programming languages such ill-defined labels as *imperative*, *declarative* and *implicit*. This paper, while avoiding discussion of the semantics and applicability of those particular terms, attempts to analyze some of the underlying concepts in the structure and use of a language.

Some form of each of the following components is present in every programming language: an elementary program statement, a mechanism for linking one elementary program statement to another or to a group of other statements, and a means by which the program can obtain data inputs. Several factors, such as the nature of the operating system and the kind of problem to be solved, influence the relative desirability of various forms of these components of programming languages. In this paper the alternative forms for these components are discussed.

The examples are drawn largely from various languages for symbol manipulation (sometimes called "list processing languages"). List languages have been chosen because list processing is relatively new; none of the forms for the components of list languages has as yet been established as "standard" even in an informal sense. Instead, a variety of interesting languages having different forms are available. Consequently, list languages are a rich source of illustrative material.

2. Elementary Program Statements

Elementary program statements can usually be classified as either *commands*, *requirements* or *implicit specifications*. A *command* is an imperative statement that commands the action to be taken without saying anything at all about what effect will thereby be achieved. A *requirement* describes the effect to be achieved without saying anything at all about the actions to be taken in achieving the effect, nor requiring that the programmer know how the effect will be achieved. An *implicit specification* is similar to a *requirement*, but the programmer must know something about what actions will be taken to achieve the desired effect.

A. *Commands*. The elementary statements of conventional assembly languages are imperative; they command that certain computations be performed, or that data be moved, or that tests be made. Thus, the terminology "order code" or "instruction set" is used to refer to the repertoire of operations of a computer. For example, the statement CLA X is manifestly a command.

In the list processing language IPL-V [1], the elementary statements are imperatives. They command certain symbols to be manipulated. For example, the statement 30 W1 is the imperative, "Pop-up the stack cell named W1".

The elementary program statements of most conventional programming languages are exclusively commands. Recently, languages have emerged whose elementary statements are statements of requirements or implicit specification statements. Programs having such statements offer outstanding advantages in many situations. However, these programs still require the use of commands. For example, the most natural way to specify when and under what conditions an input/output operation should take place is simply to *command* it to take place at the right time and under the right conditions. Notice the roundabout and inelegant mechanisms necessary to handle input/output in languages not having commands: In LISP1.5 [2], "pseudo-functions" are evaluated for their side effects, such as a "print" operation, and their true values are ignored; in SNOBOL [3], data can be output only by the awkward mechanism of requiring it to be part of the special string named SYSPOT.

B. *Statements of Requirements*. One way of describing a computer[1] is by its transfer characteristics. Data is fed in, certain transformations of the data take place and results are spewed out. One does not have to give a sequence of commands that would carry out the transformation. Instead, one may express the transfer function to be executed by the program by presenting only general descriptions of acceptable data as well as descriptions of the results into which the data are to be transformed. Simon [4] has called these descriptions the input and output *state descriptions* of the computer.

[1] By "computer" is meant the combination of a particular program running on a particular EDP machine.

The elementary statements of some programming languages consist of statements, in this state description form, of the *requirements* of the program. Some examples may be found in the program generator languages described in Young's [5] paper. The report generator feature of SIMSCRIPT [6] is a similar example.

The pattern match and rearrangement statements of the COMIT [7] and SNOBOL languages are also statements of requirements. They assert that, if a data string is of a certain specified form, then it is to be transformed into a string with a different specified form (without specifying the procedure by which this transformation is carried out). The compiler creates both the tests to determine whether the statement is applicable to particular data strings, and the program that carries out the desired transformation.

Let us define *program* for a moment in a narrow sense as a sequence of commands. Then compilers for languages whose elementary statements are commands are program *translators*; they translate the source language program into the object language program. Compilers for statements of requirements, on the other hand, are program *writers*; they create commands that bring about the required states.

The statement of requirements for solving differential equations could consist of descriptions of how to recognize both a differential equation and its solution. The compiler would then have to write a program (sequence of commands) for transforming the equation into the solution. Unfortunately, we do not now know how to write such general program-writing programs. However, less ambitious, yet useful program-writing programs *are* within our reach.

For instance, in a sample of text a linguist can arrange to separate verb roots from their endings by writing an appropriate SNOBOL or COMIT program. He need merely specify the input state—i.e., tell how to recognize verbs and their roots and specify the required output state—that each root is separated from its ending by a space. He need never know just what procedures are followed in achieving the separation.

C. *Implicit Specification.* Some programming languages have elementary statements that are neither commands nor simple statements or requirements. In order to understand the effects of such statements the programmer must be aware of unstated constraints or interpretation procedures. I call such statements *implicit specifications* of the actions of the program.

Of course the distinction between a *command* and an *implicit specification* is not clear cut. Here any command is considered implicit to the extent that the action it produces is not apparent without further explanation.

Indirectly addressed instructions and "execute" instructions are familiar examples of implicit specification statements; they require unusual evaluation procedures of which the programmer must be aware. (These special evaluation procedures are carried out by special hardware.)

LISP [2] programs are always executed by means of an implicit mechanism called the *eval* operator. The programmer must know that the elements of his program statement will be evaluated in a certain (recursive) order, and he must understand the effect of evaluating each element of an elementary program statement. On the other hand, he does not have to understand the nature of the evaluation mechanism (which, incidentally, may itself conveniently be expressed in LISP).

COGENT [8] is a language in which some program statements implicitly specify production rules for a phrase-structure grammar. Implicit evaluation procedures construct or analyze certain list structures in accordance with those grammatical rules.

Naturally, the more computation done implicitly by the programming system, the less left for the programmer to worry about. Since the implicit procedure generally performs a task that would otherwise be difficult or awkward to define, it is a great boon to the programmer. However, when an implicit evaluation procedure is an intrinsic part of a programming system and cannot be "turned off" when it is not needed, it can cause the programmer troublesome difficulties.

In LISP, the elementary program statements are either definitions of functions or applications of functions to arguments. When a program is executed, the implicit *eval* operator always maintains control as it goes about its task of evaluating expressions by applying functions to arguments. Occasionally a LISP programmer desires to describe an action (such as the modification of the property list of an atomic symbol) that is awkward to couch in the standard terms "arguments," "functions" and "evaluations." The omnipresence of the *eval* operator then actually encumbers the programming task. To get around this problem, most LISP systems offer an alternate means called the PROG feature, for structuring programs. In the PROG mode, LISP becomes a language of simple sequential commands.

3. Subprogram Linkage

The utility of subroutines was recognized early in the development of computer programming. A (closed) subroutine reduces the storage requirements for a program since the single copy of the subroutine may be used in several parts of the higher-level program. In addition, the subroutine provides a convenient building block that can assist a programmer in organizing a complex program.

Psychologists have shown that a human being can contemplate at most about seven discrete objects at any one time. In designing a system involving more than seven program segments, the programmer can either use complex flowcharts and focus his attention on a small part of a highly connected network at one time, or he can build a hierarchical structure containing independent units each of which has only a small number of subunits, and which can thus be "debugged" independently of the rest of the structure. The feasibility of building such a hierarchical structure depends largely upon the manner in which subroutines must be constructed and executed.

Some languages have provisions for identifying a group

of elementary program statements as a logical block—e.g., with begin and end statement brackets as in ALGOL. In this section we are concerned with relations between logical blocks rather than with the internal structure of a block.

A. *Explicit Call.* Most well-known programming languages require subroutines to be identified as special kinds of entities, both when constructed and when used. TSX SUBR,4 and CALL SUBR(ARG) are typical examples of explicit subroutine calls. In each case the subroutine SUBR must itself know how and where to find its arguments, where to put its results, and how to get back to the calling program. Subroutine calls in COMIT [7], a language of requirement statements for string manipulations, are handled by simple transfers of control, but only after a return location has been put in some standard place. Usually, a single pushdown list is used for all subroutine returns. As in the FAP and FORTRAN cases, the subprograms know how to get back to the calling program.

Minor differences in immediate convenience have major effects on overall programming habits. Subroutines of the conventional type described above are usually used to eliminate duplicate copies of programming code. They are infrequently used as a mental aid in organizing systems, because to use them in this way would require extra linkages and more complicated coding and loading procedures. We have all seen flowcharts of programs that are virtually unreadable and unmodifiable because they do not have a hierarchical structure.

B. *Execute Calls.* By an "execute call" is meant a subroutine call which is syntactically indistinguishable from the basic instructions of a programming language. For example, an assembly language programmer uses macro instructions in exactly the same way as he does the basic instructions of the language. Macro instructions generate "open" subroutines; that is, a new copy of the instructions in the macro definition is inserted at each use. Thus macros do not save space the way closed subroutines do. Their purpose is strictly to reduce the burden on the programmer of keeping track of "lower level" details.

IPL-V instructions are of four kinds: data transmission, storage cell pushdown and popup, branch, and execute. There is no special subroutine entry or exit mechanism; none is needed. The argument of an execute instruction may be either (1) the name of any of approximately 100 builtin programs (to perform list processing, arithmetic, input/output, and other operations), or (2) the name of any IPL program (including the one in which the execute statement occurs). Arguments and results are transmitted according to conventions which are independent of how the program unit being executed fits into the larger program system. Any program may be used as either a main program or a subroutine. Each program simply runs to termination, at which time the system executive knows where to go next (by using a pushdown stack as a conventional program counter).

IPL-V programmers rarely write routines (program units) having more than about 20 lines of code. Newell's recent version of GPS [9] has over 30,000 lines arranged as many short routines. The problems of organizing systems like GPS would be tremendously difficult were it not for the ease with which short program segments may be written independently of each other and then plugged into an appropriate part of a hierarchical structure.

C. *Function Composition.* The mathematical idea of "function" has been carried over into programming to mean a subroutine that calculates a single number, the "value" of the function. Function calls may be nested to achieve the effect of mathematical function "composition." In FORTRAN, for example,

```
X = SIN(MAX(X1,ABS(X2)))
```

generates a hierarchical set of subroutine calls. The result of each subroutine execution is transmitted to its calling program in a standard way for all "function" subprograms.

The basic element of a LISP1.5 program is the *function*; no other kind of main or subprogram is used. A LISP function call is similar to an IPL subprogram execute call in that a LISP function may call any function, including itself, simply by writing the name (or definition) of the called function at the appropriate place in the definition of the calling function. Functions are thus convenient organizational units of large LISP programs.

Most conventional programmers flowchart before they code. However, the author knows of no experienced LISP or IPL-V programmer who uses flowcharts. They may first code either the highest- or the lowest-level routines, and they may code extremely short or moderately short routines. But these programmers use the routines themselves as flowcharts, thereby eliminating a time-consuming step in the programming process. For example, the highest-level function in a program for symbolic differentiation of a function y with respect to a variable x would be defined in LISP1.5 approximately as follows:

```
diff[y; x] = [if constant [y] then 0;
              else if variable [y] then vardiff [y; x];
              else if algebraic [y] then algdiff [y; x];
              else if trigonometric [y] then trigdiff [y; x];
              else otherdiff [y; x]]
```

where *constant*, *variable*, *vardiff*, etc., are other LISP functions. Note that the program serves as its own flowchart because of the ease with which subfunctions are used.

4. Inputs to Routines

The peculiar ways and means of referencing data influence the modus operandi and efficiency of the programmer. The three principal ways of getting inputs for routines are (1) by referring to the data itself, (2) by referring to the data by a "name," usually associated with its address, and (3) by referring to it implicitly by means of values of variables or values of functions of other data.

A. *Direct Data.* The assembly language pseudo-ops, OCT and BCD, and the FORTRAN-type H format version are familiar examples of the inclusion of data itself in programs. The value of the LISP expression (QUOTE, (A, B, C)) is the list (A, B, C). Similarly, one may define

the same string of symbols in SNOBOL, and give it the name LIST, by the statement

LIST = "(A, B, C)"

B. *Reference by Name.* In the FAP "CLA X" or FORTRAN "Z = X + C", the letter X is the *name* of the desired data item and is identified in a symbol table with the storage location of the data. In IPL-V and in SLIP [10] most of the basic list-processing routines or functions have symbols naming list structures as their inputs. Here a name is identified in a symbol table as a pointer to the head of the list.

C. *Implicit Reference.* The "dummy variables" in a FORTRAN subprogram are implicit data items; neither the values nor the locations of the data are known when the program to manipulate the data is written.

In LISP programs *all* data are defined implicitly by dummy variables. A program is a set of defined subfunctions. At execution time direct data is provided to the system's top-level functions.

D. *Discussion.* Data names in programs are analogous to proper names in ordinary discourse: Sometimes they are essential, at other times they are superfluous, and occasionally they are confusing.

In ordinary discourse, one can use a pronoun such as "he" to refer to an antecedent. Also one can identify a person by his description rather than by his name. When one says, "The Prime Minister of England" one refers to the current office-holder whatever his name may be. Programs in FORTRAN or FAP, or in IPL, SLIP or COMIT, can frequently refer to data items only by their proper names. Thus, the programmer must make up names for each immediate computational result in order to be able to refer to it later. In LISP, on the other hand, intermediate data structures do not have names. All data transmission is handled automatically as specified implicitly in intermediate function definitions. Since the need to make up explicit names takes some of the programmer's attention and provides an opportunity for errors, one might expect LISP's completely implicit data transmission philosophy to be an unmitigated blessing.

Unfortunately, the naming problem is a two-edged sword. All large LISP programs with which the author is familiar—including Slagle's calculus integration [11], Evans' geometry analogy solver [12], and Raphael's question answerer [13]—make extensive use of inconvenient pseudo-functions and features of the LISP system for the sole purpose of *introducing* the ability to access intermediate data by their names. For complex symbolic data processing, treating each program segment as an independent mathematical function is not sufficient. One must also have names for entry points into "permanent" data structures that various subprograms manipulate

The ability to use different levels of data reference can enhance the flexibility of using named data. In IPL-V, an elementary program statement can contain the data itself, the name of a cell containing the data, or, for an additional level of indirectness, the name of a cell containing the name of the cell containing the data. These various ways of getting at inputs to routines have proven to be quite useful as well as confusing.

An online, text manipulation system being developed at SRI [14] provides another example of the importance of naming. The ability to identify a segment of text by its label in a standard hierarchical outline arrangement is a key feature of this system. Here, not only does the label name the text, but also the characters in the label specify the relative location of the text. An implicit naming scheme could not have this desirable property. The next version of this text manipulation system may be written in SNOBOL 3 [15], which is well suited for analyzing the characters in a string name as well as for utilizing a hierarchical program structure.

5. Factors Influencing the Structure of a Language

Thus far, different kinds of elementary program statements and various ways of linking subroutines and of referencing data have been discussed. Let us now mention some factors influencing the choice of these components in a programming language.

A. *Environment.* The mode of use of a computer influences the choice of language components. Batch, job-shop computer operation requires the minimizing of computer running time. Online "conversational" operation, whether for a single user on a small computer or for a time-sharing system, requires the optimization of other parameters. The job mix also influences language choice. The users of the MIT time-sharing system [16] are research-oriented. Their system uses MAD as its principal compiler language, rather than FORTRAN, because for these users it is advantageous to have a fast compiler and a more sophisticated language at the expense of an inefficient object code. The SRI CDC-3100 will have a special fast assembler that compiles nonrelocatable code because of its special advantages for use in an online system with symbolic debugging capability.

B. *Storage Allocation.* How to control free storage is a major problem for designers of symbol manipulation systems. One must decide (1) the precise form of the storage control mechanism, and (2) how much control should be exercized over the mechanism by the program.

A "free storage list" of linked cells is the basis of most dynamic storage allocation schemes. Cells are generally removed from this list as they are needed by the processes of the language. Several methods have been used for returning unneeded cells to the free storage list. Cells may be returned under program control, automatically as they are abandoned, or periodically by "garbage collection." The reference-count scheme used by SLIP is a clever compromise between continuous and periodic reclamation of free cells. A "garbage collector" might simply construct a linked list, or it might compress the list into a contiguous block of storage (by any of several available schemes). Choice of the storage allocation mechanisms depends upon the programming language to be used and, to some extent, upon the application to which the program will be put.

C. *Escape Mechanisms.* No matter how general or powerful a programming language seems to be, some user is bound to come up with a task that is awkward to perform with any of the available components. Therefore it is important to provide "escape mechanisms" with which the user may enhance the language.

The easiest escape mechanism provides the ability to write special-purpose machine-language subroutines that can be called from within the language. Of course, some sort of subroutine calling mechanism must be inherent in the language before externally written subroutines may be used. Thus, for example, machine-language subroutines are convenient to use in SNOBOL 3 as a by-product of the new facility to define and use SNOBOL "function" subprograms.

A more elegant—and more difficult to implement—escape mechanism is the ability for the programmer to actually extend the *syntax* of a language. This is possible if the behavior of a compiler is influenced by a syntactic description of the language being compiled. The author is of the understanding that the proposed LISP II [17] language will have a feature of this kind.

6. Conclusions

We have seen that several apparently mutually exclusive features of programming languages all have their advantages. Names are sometimes useful handles by which to reference data items, and sometimes the compulsory use of names is inconvenient; subroutines should be usable implicitly through function composition or execute calls, yet it should be possible to define special-purpose subroutines explicitly; and the elementary statements of programs are most conveniently specified, in various situations, as commands, requirements or implicit specification statements. How are we to resolve these issues?

One might think that a suitable subset of desirable components could be selected for any one "problem oriented" language. But if this were done, programmers who wished to widen their scopes would have to learn a variety of "foreign" languages. Furthermore, as more difficult and more interesting problems are attacked, larger varieties of components are generally needed than would be available in any standard set.

Embedding one language within another [18] offers one solution, although frequently this has the drawback of compounding the minor disadvantages of the embedded and the embedding languages. One observes a more direct approach to more powerful languages in the evolution of existing systems: FORTRAN I to ALGOL, or LISP 1 to LISP 1.55 [19]. Each sequence represents the progression towards a language containing more of the desirable components and alternatives discussed above. A quite ambitious language currently being implemented is LISP II [17] which resembles a marriage of ALGOL, LISP 1.5 and COMIT, and includes provisions for automatic storage allocation and escape mechanisms.

Assembly languages have already evolved to the point where they are fairly standardized (because useful components have been generally identified and provided, rather than because any committee has established "standards"). A machine language programmer can now switch from one manufacturer's digital computer to another's and, although the instruction sets will differ, the structure of the assembly language, the pseudo-ops, and the macro-programming facilities will be pretty much the same. Hopefully, before too many more years elapse, problem-oriented languages for symbol manipulation as well as for scientific and business application will reach a similar state. The most convenient ways of structuring programs, organizing systems and referencing data will become generally recognized and available. Perhaps then, those of us primarily interested in *using* computers can stop worrying about the terminology and the nature of computer language design.

RECEIVED SEPTEMBER, 1965

REFERENCES

1. NEWELL, A. (Ed.) *Information Processing Language V Manual.* Prentiss-Hall, Englewood Cliffs, N.J., 1961.
2. MCCARTHY J., ET AL. *LISP1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1963.
3. FARBER, D. J., ET AL. SNOBOL, A string manipulation language. *J. ACM 11,* 2 (Jan. 1964), 21–30.
4. SIMON, H. A. Experiments with a heuristic compiler. *J. ACM 10,* 4 (Oct. 1963), 493–506.
5. YOUNG, J. W., JR. Non-procedural languages—a tutorial. Presented at ACM So. Calif. Chapters 7th Ann. Tech. Symp., Mar. 1965.
6. MARKOWITZ, H. M., ET AL. *SIMSCRIPT—A Simulation Programming Language.* Prentiss-Hall, Englewood Cliffs, N.J., 1963.
7. MIT RESEARCH LABORATORY OF ELECTRONICS AND COMPUTATION CENTER. *COMIT Programmer's Reference Manual.* MIT Press, Cambridge, Mass., 1961.
8. REYNOLDS, J. C. COGENT programming manual. ANL-7022, Argonne Nat. Lab., 1965.
9. NEWELL, A., ET AL. Report on a general problem-solving program. Proc. Int. Conf. on Information Processing, Paris, UNESCO House, 1959.
10. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM 6,* 9 (Sept. 1963), 524–544.
11. SLAGLE, J. R. A heuristic program that solves symbolic integration problems in freshman calculus. *J. ACM 10,* 4 (Oct. 1963), 507–520.
12. EVANS, T. G. A heuristic program to solve geometric-analogy problems. Proc. 1964 Spring Joint Comput. Conf., Vol. *25,* Spartan Books, Washington, D.C., May 1964, pp. 327–328.
13. RAPHAEL, B. A computer program which "understands". Proc. 1964 Fall Joint Comput. Conf., Vol. *26,* Oct. 1964.
14. ENGELBART, D. C. Augmenting human intellect: experiments, concepts, and possibilities. Sum. Rep., AF49(638)-638)-1064, SRI Proj. 3578, Stanford Res. Inst., Menlo Park, Calif., Mar. 1965.
15. FARBER, D. J., ET AL. SNOBOL 3. Bell Telephone Labs, Holmdel, N.J. (unpublished)
16. CORBATÓ, F. J., ET AL. *The Compatible Time-Sharing System —A Programmer's Guide,* MIT Press, Cambridge, Mass., 1963.
17. LEVIN, M., ET AL. LISP II project. Tech. Mem. Ser. 2260, Systems Development Corp., Santa Monica, Calif., 1965.
18. BOBROW, D. G., AND WEIZENBAUM, J. List processing and extension of language facility by embedding. *IEEE Trans. EC-13,* 4 (Aug. 1964), 395–400.
19. WOOLDRIDGE, D., JR. The new LISP system (LISP1.55). Artif. Intel. Proj. Mem. *13,* Stanford U., Feb. 1964.