DISCIPLINE IN SOFTWARE DEVELOPMENT

The successful construction of medium and large software systems requires the management of the complexity inherent in the problem being programmed. A well-disciplined approach to software development involves the production of a complete specification, a complete problem solution, and program design prior to the inception of actual coding. In practice, this requires the production of some form of program design representation [1] from the original specification, with the action of each module specified with a program design language [2]. Furthermore, data structures are specified and refined, in some cases to physical data structures, but more commonly to logical data structures.

It is from that point that coding begins. The information available to the coder should include, at a minimum, the input and output parameters for each independent program unit and an unambiguous description of the operations to be carried out by each. Analysis of information flow, performance or space requirements, and similar considerations lead to the identification of commonly used routines and data, yielding an initial program structure derived from the design.

A disciplined approach to software development, then, requires that the program design stage precede the program construction stage. The completed software design can be checked against the original specification by “walkthroughs” [3] or similar methods, with the resulting “software blueprints” providing the basis for implementation (or possibly redesign).

An important consideration in the target programming language, then, is the ease with which one can proceed from the design representation, with its modular structure and its degree of abstraction, to the program representation, i.e., executable code. A second key consideration is the ease with which one can determine the conformity between the completed program and the original specification, using testing and/or verification techniques.

PLAIN AND ITS DESIGN CONTEXT

The past few years have witnessed an increased understanding of the relationship between programming languages and problem solving [4,5]. As a result of this work in programming methodology, programming languages are no longer viewed as independent entities, but rather as an integral part of the problem-solving process. Programming languages are now seen as a mechanism for expressing a problem solution in a precise way for computer execution. As such, a given programming language may have a significant effect upon the ease with which the solution may be expressed. If the language does not easily support the abstractions used by the programmer in solving the problem, then the transformation from the problem solution to a correctly executing program will be complex, with the increased likelihood that errors will be introduced during this transformation process.

A number of new programming languages have been designed and/or implemented with a primary or secondary objective of promoting proper programming techniques [6,7,8,9,10,11,12]. In addition, some general criteria for language designs have been advanced [13,14,15,16]. Design of the programming language PLAIN (Programming LAnguage for INTERaction) has proceeded in parallel with these other efforts, commencing in 1975. Unlike the other languages, the intended application area for PLAIN is interactive information systems, typically programs whose end users will be application-knowledgeable and computer-naive. PLAIN is intended to provide the application programmer with a tool that supports the systematic construction of this class of programs. As such, it contains facilities for definition and use of relational data bases, modules for information hiding, string processing with a simple pattern-matching facility, and exception-handling, incorporated into a well-structured, Pascal-based language.

In this paper, however, we shall be concerned primarily with the support provided by PLAIN for concepts of systematic programming. We begin by presenting some goals that encourage a disciplined approach to software construction, commenting briefly on their contribution to the overall goals. Then, following a short survey of other languages, we examine PLAIN with respect to these design goals, partic-

* This work was supported in part by National Science Foundation grant MCS78-26287. Computing support for text preparation was provided by National Institutes of Health Grant RR-1081 to the University of California, San Francisco, Computer Graphics Laboratory, Principal Investigator: Robert Langridge.

731
ularly those of abstraction and modularity, and compare the approach of PLAIN with those of some other modern languages. Information on other aspects of the language and its implementation may be found in [11,17,18].

LANGUAGE DESIGN GOALS FOR SYSTEMATIC PROGRAMMING

Although the intended application areas and the relative priority of the goals vary considerably among the recently designed languages, there are a number of areas of general agreement that can be identified. These common objectives, taken together, provide a sound basis for programming language design. Languages that meet these objectives can be expected to provide an excellent framework for the systematic construction of high quality programs. These objectives are presented briefly and with only the most significant aspects of their rationale, as additional discussion of these issues may be found in the cited references.

1) Support for abstraction

Abstraction has been recognized as a means to develop a representation of concepts that relates closely to the application being programmed, to hide inessential details of the problem solution at various levels of the program development process, and to support the notion of “top-down” design. If a problem solution involves the use of queues or directed graphs, for example, one should be able to make use of those objects in the programming process.

The ability to define these abstract objects, along with appropriate operations on these objects, is extremely valuable. Such objects can be specified formally using algebraic techniques to define their behavior [19]. If the objects and their associated operators are encapsulated so that the representation of the object is isolated and inaccessible from other parts of the program, the facility for data abstraction is analogous to the facility for procedural abstraction provided by functions and procedures in many programming languages.

Such a programming language facility, generically termed abstract data types [20], provides the programmer with the opportunity to define behavioral characteristics of data objects and to refine program and data structures in parallel. It is then possible to create data objects within a program resembling those used in the problem solution, thereby easing the process of transforming the problem solution into a program.

2) Support for modularity

Although there are a number of different definitions of a “module,” for purposes of this paper, one may consider a module to be an object, perhaps a procedure, function, or abstract data type, that carries out a well-defined operation, hides a design decision, or isolates information from other modules. Typically, the actions may be described in a sentence or two of natural language. Furthermore, each module has well-defined interfaces to other modules. Modularity makes an important contribution to the overall comprehensibility of programs, to the practice of programming by levels of abstraction, and to the production of large software systems by allowing various pieces of a software system to be effectively isolated from one another [21,22,23].

The ability to decompose a large problem into a number of smaller ones and to delineate clearly the interactions among the pieces is an important tool in gaining intellectual mastery over complex problems. Software design aids such as HIPO charts [24] and structure charts [25] have been developed to help identify modules and to represent the total structure of the software system so that the decomposed modules can be integrated into a single integrated system. Furthermore, concepts of cohesion (unity of function) and coupling (module connections) [22,25] provide a basis for evaluating module designs.

3) Support for verification and testing

Program correctness, as determined through either formal verification or testing, has been a critical motivation for much of the work in software engineering and programming language design. Verification is a formal mathematically-based proof that a program conforms to its specification. Testing is a collection of activities that provides a practical demonstration of conformity between the program and its specification, based upon systematic selection of test cases and execution of program paths and segments.

Both the characteristics of a given programming language and the practices used to write programs in the language affect verification and testing. The ease of testing and verification is further influenced both by static and dynamic program characteristics [26]. Static factors are those features that may be automatically checked by a compiler at translation time, those that are independent of the execution characteristics of the program. Examples of static aspects include most type checking and some checking for the use of aliasing.

Dynamic factors are those aspects of the program that are dependent upon its execution properties, including control flow and response to exceptional conditions. Issues of programming style, such as the use of uncontrolled branches and pointer structures, clearly affect the complexity of checking required.

Support for verification and testing is closely tied to some of the other issues as well. For example, the desirability of testing or proving program modules individually fits in well with the desirability of system design at the module level. In addition, support for verification and testing implies the prior development of system specifications and hence a systematic approach to software creation. Finally, other issues such as modularity and readability are closely related to issues of program correctness, since the determination of correctness is greatly aided by module simplicity and comprehensibility.
4) Program readability

Program readability has been seen to be a valuable program property contributing to ease of program maintenance and modification [13]. The use of opaque programming "tricks" or the construction of cryptic programs is no longer considered to be an acceptable programming practice, as it has become recognized that programs must be read by humans as well as by machines during their increasingly long lifetimes.

Many properties combine to yield readable programs, including the use of mnemonic variable names, the presence of meaningful keywords, the liberal insertion of comments, and linear flow of program control. Here, too, programming practices are important, since it is possible to write a well-structured, highly understandable program in "poor" languages and a totally incomprehensible program in even the "best" language. Furthermore, program readability appears to be a highly personal and highly subjective quality, significantly influenced by the reader's previous programming experience and programming style.

5) Prevention of self-modifying programs

A number of languages, most notably LISP, treat programs and data interchangeably, in such a way as to permit the code being executed to vary dynamically, i.e., to be determined at execution time. Such an approach is entirely consistent with the concepts of stored programs and Von Neumann machines; unfortunately, though, this approach is in conflict with the goals of program readability and support for verification and testing, since the ability to create new variables and to alter the program dynamically makes verification and testing impossible unless one is able to test or prove all of the programs that can be generated. Furthermore, such programs are often difficult to comprehend, since the actual code is not totally visible. In Pascal and its descendants, procedures and data are separate entities, where data objects may change their values dynamically and procedures are static and immutable. Programs that permit "the execution of data" are forbidden.

6) Control of scope and binding of variables

Block-structured languages provide explicit control over the existence of variables. Space for declared variables is allocated upon entry to a block and deallocated (except for statically allocated variables) upon exit from that block. The set of known variables can be determined from observing the static structure of the program, with no ability to create variables dynamically.

Control of the scope and binding of variables has been identified as a technique that can reduce programming errors caused by side effects, particularly those resulting from indiscriminate use of global variables [27]. Such control is also needed to achieve modularity, since, without it, a programmer may easily circumvent restrictions concerning the proper use of input and output parameters for a module.

The use of pointers should also be noted here, since they may contribute to this problem. Many languages, such as PL/I and Pascal, permit the creation of "dangling references" by having an object in an outer block point to an object in an inner block. When control leaves the inner block, the object pointed to may disappear, but the pointer itself will remain.

7) Language size

Language size has also been seen to be important, since relatively small languages are easier to implement and can make it possible for the programmer to gain complete mastery of the programming language [13,14]. A number of different, albeit "rough," metrics can be used to estimate language size, including the number of keywords, the size of its grammar (in LALR form, for example), the number of statement types, or the size of the compiler or interpreter for a given computer.

There appears to be an optimal size for languages, with some languages being so small as to prohibit an adequate variety of control structures or data types, while other languages are so large as to prevent the average programmer from gaining a clear understanding of the entire language, with all of its syntactic and semantic subtleties.

These seven design objectives are not orthogonal. Indeed, there are numerous intricate connections among them, as well as some inherent conflicts. For example, control of scope and binding of variables is closely related to modularity. On the other hand, restrictions on language size may serve to limit the extent to which a language may support a variety of abstractions. Thus, the language designer seeking to achieve these design objectives must give higher priority to some objectives than to others and must trade off various alternatives judiciously.

LANGUAGES DESIGNED FOR SYSTEMATIC PROGRAMMING

As noted above, a number of different programming languages, including Pascal, CLU, Alphard, Gypsy, Euclid, LIS, PLAIN, Mesa, and Ada, have been designed with most or all of these design objectives in mind. (See [28] for example.) Even though the different languages are intended to serve a diversity of language requirements and applications areas, the languages have more similarities than differences when examined from the standpoint of support for systematic programming.

The most significant differences are those caused by different emphases in the design goals among the various languages. For example, Alphard and Euclid place a heavy stress on the goal of program verification, while the others might be said to recognize the importance of verification without the explicit requirement that programs in those languages will be verified. As another example, LIS and Euclid are seen as system implementation languages, to be used...
primarily for the development of operating systems, compilers, and similar programs, while CLU and PLAIN are application languages. (This is not to imply that the languages in one group cannot be used for other applications, but only the intent of their designers.)

In the remainder of this paper, we will examine the design decisions in PLAIN with respect to these objectives for supporting a systematic approach to program construction, assessing some of the decisions in comparison and contrast with those made for other programming languages. The intent of this discussion is to provide some insight into the design of PLAIN and into some of the tradeoffs that were made in that design; the reader is not expected to agree with all of these decisions—if there were unanimous agreement on these issues, there would not be so many languages! In short, one of the implicit goals of many of these new languages (as can be seen from their defining documents) is to gain additional understanding of programming methodology and the ways in which language features aid or hinder the programming process.

From a software engineering standpoint, each may be regarded as a tool that can be made available to the individual software development group as an instrument for building their product. It is to be expected that some of these tools will receive little use and little acceptance, while the use of others will be strongly encouraged and modified and/or enhanced over time.

Finally, it should be noted that the programming language is part of a complete problem-solving process, which is supported by a software development methodology and a programming environment. The environment and the methodology will vary among organizations and among languages, but it is really the programming language, in combination with the programming environment, that determines the full extent of support for systematic programming that is provided for the programmer.

PLAIN: A LANGUAGE DESIGNED FOR RELIABLE INTERACTIVE SOFTWARE

As noted above, PLAIN (Programming LAnguage for INTERaction) is addressed to the dual goals of support for the construction of interactive programs, i.e., those programs that execute interactively and support for structured programming (in the original sense of that term [4]). PLAIN was designed with features to assist the development of programs involving conversational access to a data base.

These features include:

1) the data type string for variable length strings, along with appropriate operators and functions for string manipulation;
2) an elementary pattern specification facility along with pattern-matching operations, used both for validating user input and for formatting of input and output;
3) the data type relation and a set of operations to provide a facility for relational data base management [17,29];
4) a procedure-oriented exception-handling mechanism for trapping errors and restricting control flow upon the occurrence of an exception, commonly used in the event of user input errors.

This set of features is largely missing from other programming languages that seek to support systematic programming. At the same time, those languages that are most heavily used for the construction of interactive program—BASIC, MUMPS [30], APL, LISP, and FORTRAN—are quite weak in meeting the design objectives stated above. PLAIN, by contrast, addresses both groups of design objectives.

From the outset, the original contribution of PLAIN was seen to be not so much the introduction of new language features, but rather a synthesis of features whose interaction would lead to a useful tool. In particular, the combination of relational data base management and facilities for data abstraction provides a powerful mechanism for structuring operations on data bases. Indeed, the design effort was undertaken with some reluctance, and only after a careful look at a number of other programming languages.

Given the planned number of innovations for supporting interactive programs, it was decided to be fairly conservative with respect to the inclusion of new features for systematic programming. The original intent was to remain fairly close to Pascal for these features; however, parallel developments in other language design efforts, including all of those mentioned above, were highly influential and the resulting language resembles Pascal somewhat less than was originally planned.

These new features are not only intended to support the creation of well-structured programs, but to go beyond that point so as to make a well-disciplined approach to program development a necessity for proper use of the language. In particular, it was considered extremely important to include features that aided modular decomposition of systems, with emphasis on intermodule communication [31], and to support joint refinement of procedures and data.

We now outline some features and design concepts of PLAIN that provide good problem-solving support and that impose various programming restrictions. The primary objective is not so much to present the PLAIN language in detail as to show the motivations of the design from the standpoint of programming discipline, with reference to the set of design objectives discussed above. Because of the interactions among these objectives, though, the subsequent discussion is structured along slightly different lines.

Abstraction and modularity in PLAIN

Abstraction and modular decomposition are two critical intellectual tools used by humans to solve problems. They are intricately related to one another, as each is intended to exhibit a view of a process or an object. For example, merely describing (at some level of abstraction) a process for sorting numbers into ascending order is inadequate for incorporating that process into a computer program; it is also essential to include a description of the interfaces between that operation and the host program.
To look at it another way, a module is a "black box" that provides an abstract view of a process or object to its invoker. Even though support for abstraction and support for modularity are presented as two separate design objectives, the extent to which one is achieved strongly affects the extent to which the other can be achieved. This is apparent if one considers the effect of being able to examine the internal structure of one module from another module; if one makes use of that internal information, then the abstraction is violated.

Many of the differences between Pascal and PLAIN are caused by the desire to provide better support for abstraction and modularity in PLAIN. Pascal has four key discernable weaknesses in this regard:

1) Unrestricted access to global variables—program units may freely access and/or modify variables declared in a containing lexical scope (unless the inner scope has a newly declared variable with the same name); thus, the use of specific variables is hidden, and a considerable amount of code inspection is required to determine the data flow. Access to dynamic structures via globally-declared pointers also makes it possible to create "dangling references," since the object being pointed to may be deallocated.

2) Absence of input/output parameters for modules—parameters in Pascal are passed by value and by reference (var). However, passing a variable by reference is not a guarantee that it is an output parameter, since it is considered a good programming practice (and an efficient one) to pass structured variables by reference, thereby eliminating the space and time required to make a copy of the parameter. Nonetheless, neither the procedure heading nor the procedure call gives an indication as to input or output parameters. Indeed, the concept of passing parameters by value and by reference is an implementation concept rather than a programming concept.

3) Lack of support for data abstraction modules—Pascal supports procedural abstractions (procedures and functions), but has no facility for defining encapsulated data types, similar to those present in CLU (a cluster), Alphard (a form), Euclid (a module), or others.

4) Side effects in functions—it is possible for a Pascal function to accept parameters by reference and to modify them within the body of the function; similarly, it is permissible for a function to make an assignment to a global variable. Such a capability goes against the mathematical concept of a function, as well as breaking down the abstraction embodied in the function and (effectively) creating additional output parameters from the function module.

PLAIN attempts to overcome each of these weaknesses, thereby providing stronger support for abstraction and modularity. First, all use of global variables must be declared in the heading of the individual program unit (procedure, function, data abstraction module). The PLAIN imports list is similar to that of Euclid and the glocon/glovar declarations used by Dijkstra [32]. Some of these names are local declarations, some are parameters, but the rest are global variables or other program units. These nonlocal names must appear in the import list, along with a classification of their use, as modified, readonly, or invoked. This requirement does not apply to constants or to type declarations, which may be used freely. The effect of the imports list, though, is to increase the visibility of the use of variables throughout a program and to permit the reader of a module to determine the interrelationships between modules, both invocations and data connections.

In conjunction with use of the imports list to specify access to variables and program units, PLAIN contains the ability to restrict the use of a given variable to a designated set of program units. This feature, called the restricted to clause, controls the extent to which globally-declared variables may be used. With the imports clause alone, any global variable may be freely imported. However, there are many instances when it is desired to share a variable among a set of program units and to prevent it from being accessed by other units. (Labeled COMMON in FORTRAN can serve this same purpose.)

Consider, for example, a program in which routine main may call procedures p1, p2, and p3. Further, assume that p2 and p3 will both need the variable k, but that neither of them calls the other. Hence, communication of the value of k must occur through main. It is desired to prevent p1 from obtaining (and possibly modifying) k. Thus, one can declare

```
var k: integer restricted to p2, p3;
```

as a way of achieving the desired protection.

Furthermore, PLAIN, like Ada, overcomes the dangling reference problem by forbidding deallocation of dynamically allocated variables. While this is not an entirely satisfactory solution from the standpoint of storage utilization, it is the only solution that permits the use of pointers without resorting to garbage collection and without permitting dangling references. The use of objects of pointer type is restricted in PLAIN in order to limit the number of program units that are aware of the representation of dynamically allocated objects.

Next, PLAIN has different rules from Pascal concerning parameters. PLAIN parameters may be either readonly or modified. A readonly parameter is an input parameter to the procedure or function whose value is not changed by the procedure or function. A modified parameter is a parameter that may have a value assigned to it during the execution of a procedure (possibly as a result of a call to a procedure invoked from within that procedure); as such the actual parameter for a formal modified parameter must be a variable. It may or may not have an input value. (An alternative strategy would have been to follow LIS and Ada, which have in, out, and inout parameters. The readonly parameters and the modified parameters are separated, in both the procedure declaration and the procedure invocation by the symbol "→".)
For example, one might declare a procedure for the greatest common denominator with the following heading:

\[
\text{procedure gcd (m,n: integer; x,y,z: integer);}
\]

with a valid call appearing as

\[
gcd(59,93;x,y,z)
\]

where \(x\), \(y\), and \(z\) have been declared as integers in the invoking routine.

This decision has several implications for implementation. First, conformity to the declaration must be checked to make sure that no assignment is made to readonly parameters. This involves making sure that the formal parameter does not appear on the left hand side of an assignment statement, in the modified part of an actual parameter list for a procedure called from within the given program unit, or as a modified variable imported into a lexically nested program unit. Although all of these checks can be made prior to execution time, they can involve a considerable amount of overhead.

An implementation advantage, however, is that it then becomes unnecessary to pass any of the parameters by value, thereby eliminating the overhead associated with copying of parameters. Because the use of the parameter can be checked from the program text, it is possible to pass all parameters by reference, regardless of whether they are read-only or modified. Thus, the programmer may accurately characterize all parameters as readonly or modified, depending upon their actual use. The overhead occurs at translation time and not during program execution.

The features described to this point have a significant impact upon the ease of transformation between the design phase and the program. Suppose that a system had been designed using the practices of Structured Design [19]. Part of the design representation is a structure chart showing the hierarchical structure of the system and the calls between modules. Each path between modules is numbered and an accompanying parameter table shows the input and output parameters for each module. For example, in Figure 1, the call to \(A2\) from \(A\) (path 5) provides \(Y\) as an input parameter and obtains \(Z\) as an output parameter; it can be seen that \(Z\) is then passed to \(MAIN\) as an output parameter; it can be seen that \(Z\) is then passed to \(MAIN\) as an output of \(A\) (path 1).

Third, PLAIN contains a facility for encapsulation, bearing some resemblance to similar features in CLU, Euclid, and Ada. In addition to defining new types, one can also encapsulate a set of related procedures and functions, providing a feature similar to that of the Ada package. Each encapsulated type declaration consists of a rep clause, in which the representation of the type is declared, an ops clause, in which the operators upon the type are declared, an exports clause, in which the names of externally visible operators are given, and an optional exception clause, in which one can name exceptions associated with the operations upon the type.

The procedures \text{read} and \text{write} may be defined in the type to extend the built-in \text{read} and \text{write} operations. The Boolean function \text{equal} may be defined to extend the built-in equal function for structured variables. The procedure \text{init} may be defined to specify actions to be carried out when a variable of that type is declared. The abstract type facility, along with several of the features previously described, can be illustrated by the familiar example of an integer stack.

The operations upon the stack may be specified as follows:

\[
\begin{align*}
\text{create:} & \quad \rightarrow \text{stack} \\
\text{push:} & \quad \text{stack} \times \text{integer} \rightarrow \text{stack} \cup \text{stackfull} \\
\text{pop:} & \quad \text{stack} \rightarrow \text{stack} \cup \text{stackempty} \\
\text{top:} & \quad \text{stack} \rightarrow \text{integer} \cup \text{stackempty} \\
\text{empty:} & \quad \text{stack} \rightarrow \text{Boolean} \\
\text{equal:} & \quad \text{stack} \times \text{stack} \rightarrow \text{Boolean} \\
\text{size:} & \quad \text{stack} \rightarrow \text{integer}
\end{align*}
\]

Axioms:

\[
\begin{align*}
\text{top}(&\text{push}(s,i)) = i \\
\text{top}(&\text{create}) = \text{stackempty} \\
\text{pop}(&\text{push}(s,i)) = \text{if size}(s)<\text{MAX} \text{ then s else stackfull} \\
\text{pop}(&\text{create}) = \text{stackempty} \\
\text{equal}(s1,s2) = \text{if empty}(s1) \& \text{empty}(s2) \text{ then TRUE else if empty}(s1) \mid \text{empty}(s2) \text{ then FALSE else (top}(s1) = \text{top}(s2)) \& \text{equal}(\text{pop}(s1),\text{pop}(s2)) \\
\text{size}(&\text{create}) = 0 \\
\text{size}(&\text{push}(s,i)) = \text{size}(s)+1 \\
\text{size}(&\text{pop}(s)) = \text{size}(s)-1
\end{align*}
\]

Before presenting the PLAIN module, it is important to make some observations about the specification. First, the \text{create} operation is carried out by the declaration of a variable of the type \text{integerstack} in the program using the data ab-
straction. Thus, there is no explicit create operation in the integer stack module. Next, the stack specification given here is somewhat different from the specification given elsewhere in the literature [19,33], primarily to accommodate the stack-full result caused by the finiteness of machine resources.

The code for the module is shown in Figure 2. It should be noted that the implementation is not a direct encoding of the specification (hinting at some problems that verifiers might have). The primary difference is that the specification of equal uses a recursive definition, while the implementation examines individual elements of the stack. There are three reasons for this change: 1) recursion is usually more expensive in terms of machine resources; 2) pop is a procedure, not a function, and so cannot be used in the language in the way that it is used in the specification, and; 3) naming rules complicate the means of referring to individual objects in each of two different stacks being compared. In addition, one would have to make copies of the stacks to use a recursive equal operation without destroying the stacks; that, too, is more expensive than a simple element-by-element comparison.

Limited parameterization of the type definition is permitted, as shown by the stack size parameter MAX. The formal parameters must be of a simple type. Thus, one can use a single data abstraction to define integer stacks of different sizes, but not to define a stack of integers and a stack of strings. The reason for this restriction is that relation is a data type and it was desired to prevent abstract type definitions from accepting relation as a parameter; the cleanest solution was a complete prohibition of type parameters. The resulting facility is less powerful (but easier to implement) than the generic package facility of Ada. One can now declare, for example,

\[
\text{var s1: integerstack [50]; s2: integerstack [100].}
\]

As noted above, it is the intention of PLAIN to disallow side effects in functions. At the simplest level, it is possible to make certain that no globals are imported and modified, and that no randomly globals or parameters are used as modified parameters in procedures called from within the function. Also, the syntax of the language forbids the presence of modified parameters; in their absence, it is impossible to use aliasing to cause side effects.

In order to be strict about the side effects requirement, though, more checking is required. First, certain data base operations must be prohibited; specifically, those modifying the current tuple indicator or the data base itself, caused by iterating through a relation, can be considered a side effect. Second, input/output operations must be restricted, since alterations to a file may be considered a side effect, especially if the file can be read after termination of the function. Such a restriction can cause complications for the software developer desiring to place debugging messages within functions, for example. Third, since functions may call procedures, all of the procedures called during execution of a function (to an arbitrary number of levels of invocation) would have to be checked to make certain that they, too, do not violate these restrictions on side effects.

In short, even though it is highly desirable to prevent all side effects, the costs of doing so, both in execution overhead and programmer inconvenience, must be considered. The prevention of input/output operations is particularly problematic in this regard, and PLAIN relaxes the side effect restriction to permit input/output within the body of functions. Otherwise, PLAIN requires sufficient declarations by the programmer in the heading of each program unit that it is possible to check procedures to see if assignments to global variables are made.

From an implementation standpoint, it is straightforward to check the restrictions on the use of globals. A flag can be set to indicate whether or not the stack of activations includes a function call. If there is an active function call, i.e., the calling sequence of program units includes a function, then the procedure to be executed must be checked for modified globals. Otherwise, the call is disallowed and an exceptional condition is raised. Note, though, that this is only a partial solution to the problem, since the declaration (in an import statement) that a procedure can modify a global variable does not necessarily mean that the global is modified on a particular call to the procedure, since control flow may bypass any statements causing a disallowed assignment. Without this compromise, however, it would be necessary to check every assignment within such proce-

---

**Figure 2—Encapsulated type definition for integer stacks in PLAIN**

type integerstack [MAX: integer] is
module
import push, pop, top, empty, equal; exception stackfull, stackempty;
pop
record
stacktop: 0..MAX;
elements: array [1..MAX] of integer
end record;

eps
function size(integerstack): integer; //computes size of stack
begin
end stacktop := size;
end size;

functions
empty: integerstack; //return true if stack is empty
begin
end empty := size = 0;
end empty;

procedure push [x: integer]; //pushes integer x onto stack s
begin
end push;

procedure pop [x: integerstack]; //pops top element of stack s
begin
end pop;

function top (s: integerstack): integer; //return value on top of stack; no pop
begin
end top := s.stacktop;
end top;

function size(s:integerstack): Integer; //computes size of stack s
begin
end size := s.stacktop;
end size;

procedure pop (s: integerstack); //pops top element of stack s
begin
end pop := s.stacktop;
end pop;

function equal (x: integerstack); //returns true if stack x = stack y
begin
end equal := true;
end equal;

end module;
Support for verification

The design of PLAIN was motivated primarily by application needs; in the application areas addressed by PLAIN, there is a strong need for software and data reliability, particularly in areas such as medicine, where proper operation of a system may have life-critical importance. At the same time, though, the need for operational systems is so great that most developers of such systems tend to begin by writing code rather than by following any kind of coherent system design methodology. At present, there is almost no likelihood that anyone would attempt to prove the correctness of such a system, even had they produced a sufficiently rigorous specification.

Thus, support for program verification was not a major objective in the design of PLAIN, in the sense that it is in Alphard or Euclid. The assistance that PLAIN provides for program verification comes primarily through its resemblance to Pascal and to other modern languages. For example, PLAIN contains an assert statement that can be checked at execution time, but the statement only permits a Boolean expression, with no provision for such essential features as expressions involving universal or existential quantification. (Such quantification could be checked in a Boolean function that is part of the assertion.)

Along the same line, PLAIN is like Pascal with respect to aliasing, rather than including the features of Euclid that prevent aliasing. However, PLAIN improves upon Pascal with respect to the use of procedures and functions as parameters by requiring type information to be provided for the parameters of the procedure and function parameters. In this respect, it follows the proposal of the British Standards Institute for Pascal [35]. In this way, it is possible to perform a greater degree of type checking while still permitting function and procedure parameters.

This is not to say that the design of PLAIN ignores the possibility of verification, though, only that it was not a principal goal. A significant problem is that effective verification techniques have not yet been developed for the class of programs addressed by PLAIN. For example, very little has been done concerning verification of data base operations. Furthermore, even though the data base operations may be mechanically correct, it is impossible to guarantee with the present collection of facilities that the results are semantically meaningful.

PLAIN takes one small step in this regard, however, through its rules concerning type compatibility. In PLAIN, any two types having different names are different types. (The designers of Ada subsequently made the same definition.) Among the data base operations, the join operation of the relational algebra can only be performed on two objects of the same type. Thus, one can make judicious use of the data type facilities to assure that only meaningful joins can be performed.

As an example, consider two relations A and B, where A contains the attribute "age" and B contains the attribute "quantity"on"hand." If these attributes are both declared to be of type integer, then the relations A and B may be joined on these compatible attributes, however meaningless the result may be. If data types "agetype" and "amounttype" are defined in advance, though, with "age" declared to be of type "agetype" and "quantity"on"hand" declared to be of type "amounttype," then it becomes impossible to perform the join. In this manner, one may specify exactly which joins may occur and may verify their correctness from a logical standpoint.

Another verification problem is presented by the exception-handling mechanism. Once again, there are no practical methods for verifying programs in the presence of exceptional conditions; one might say that the occurrence of such a condition means that a program has failed to satisfy some input assertion and that the program therefore cannot be proved correct. Yet exception-handling is fundamental to PLAIN, since it is necessary to provide the programmer with facilities to prevent exceptional conditions from causing a program to terminate abnormally. The anticipated end users of PLAIN programs, being largely computer-naive, can be expected to make numerous errors, particularly in input, that must be properly trapped and handled; one simply cannot say that the program has failed to meet some input assertion and must therefore be terminated. Accordingly, the application programmers writing programs in PLAIN must be given the ability to trap and handle exceptions.

The PLAIN exception-handling mechanism, described at length in another paper [36], seeks to provide a well-structured flow of control following the occurrence of an exceptional condition. The programmer may create a handler procedure that can be associated with the occurrence of a specific exception at a specific program location. When an exception is raised, either through the signal statement, or through an automatic mechanism in the language processor, the handler procedure can carry out any required actions, potentially clear the offending exception, and then return control to normal program flow, to the beginning of the statement in which the exception occurred (retry) or to the invocation point of the procedure in which the exception occurred. In this way, exceptions can be passed through succeeding levels of invocation with any necessary actions being taken at each level. Since exception-handling is done with procedures, it is possible to pass parameters from the environment of the exception to the handler procedure, following the normal rules for scoping of declarations. At any point, the active exception may be cleared by the handler for that level so that normal program operation can continue.
The intent of this approach is to facilitate both the programming of exception-handling actions and the verification of programs in the presence of exceptions, since this method avoids the unrestrained flows of control and unrestricted access to variables that characterize some of the other exception-handling schemes. Although a more detailed approach to this verification is sketched out in [36], there has not yet been any practical experience with the application of verification techniques to such programs.

**Support for program readability**

Although, as previously noted, program readability is difficult to quantify and can be strongly affected by individual programming styles, it is possible to provide language features that enhance program comprehensibility. Many of these features provide support for other systematic programming goals as well. In general, the design of PLAIN attempts to follow Hoare’s dictum that “the readability of programs is immeasurably more important than their write-ability” [13].

As with many other language aspects, much of the readability of PLAIN programs results from its resemblance to Pascal. Among the common features supporting readability are:

- provision of appropriate keywords
- format free program structure permitting indentation on lines
- control structures supporting linear flow of program control within program units
- prevention of self-modifying programs
- straightforward provision for comments
- limited language size.

Similarly, the Pascal-like program structure retains the disadvantage of placing the main program at the end of the program text.

PLAIN incorporates some additional features intended to enhance program readability (as well as to help in achieving other goals). These features are the following:

- fully bracketed control structures
- explicit importing of global names into a module
- input/output parameter lists in both declaration and call of procedures.

The use of fully bracketed control structures permits a more consistent language definition and can reduce the use of begin-end pairs as separators. The reduction in begin-end pairs not only eliminates unnecessary program “clutter,” but also removes a major source of programming errors, making the begin-end now serve only the single purpose of enclosing an entire executable program unit (main program, function, or procedure).

In Pascal, for example, the structure of the if statement is

```
if Booleanexpression then statement [else statement].
```

In PLAIN, as well as in Ada and other newer languages, it is

```
if Booleanexpression then statementlist [else statementlist] end if.
```

Similar gains are achieved with the case statement. The statement is terminated with an end case and individual cases are separated with the reserved word when. Again there can be a considerable reduction in the number of begin-end pairs, producing a situation in which both readability and writeability are improved.

The imports list, discussed above, in addition to helping enforce rules concerning modularity, is an aid to program readability. Because declarations and imported names are all visible in the heading of a program unit, it is easier to comprehend, modify, and/or validate units independently. The designers of Ada have taken the opposite view, claiming that importation of a large number of objects will detract from program readability and cause additional clutter. This author believes that the proper use of structured objects, combined with efforts to minimize coupling between modules, will prevent the imports list from becoming excessively long, and that its presence provides a good mechanism for specifying the interface between the PLAIN program and its execution environment. Further experience in the use of these languages may help to resolve this difference.

Another improvement to readability comes about from the restrictions on the use of pointer variables in PLAIN. Because pointer variable may only be used within modules, most program units are free of expressions involving complicated data access methods, such as multilevel pointer structures. While PLAIN does not achieve a uniform reference mechanism, the number of reference methods is quite small. Furthermore, function and procedure calls must be used to access the operations on the complex data structures defined in data abstractions. This restriction has several benefits:

- access to the physical representation of a data object is sharply restricted so that the reader of the program only needs to understand the logical operations on the object once the isolated representational information is understood
- the reader, typically performing a maintenance activity, needs to study much less of the program text in order to make changes to the data structures
- meaningful names can be chosen for the functions and procedures, thereby aiding reader understanding of the program.

It must also be recognized that some of these gains in readability come at the expense of some overhead in space...
or execution time as a result of the additional procedure and function calls needed to accomplish the encapsulation of data.

CONCLUSION

This paper has examined the design of the programming language PLAIN from the standpoint of the support that it provides for the notions of systematic programming, focusing on both its strengths and weaknesses. It can be seen that the design of PLAIN places major emphasis on the goals of abstraction, modularity, and readability, and that it makes advances over Pascal and features of some other modern languages with respect to supporting a well-disciplined approach to software construction.

At the same time, support for program verification and testing was consciously left at a lower level than is possible given the current technology of programming language design. The language size is moderate, containing more features and more syntax than Pascal, and being comparable to Ada in that respect. The goal of small language size was not achieved as fully as had been hoped, due to the apparent needs of the application area.

The implementation of PLAIN is presently under way on the PDP-11 computer under the UNIX operating system, and it is expected that an initial implementation will be operational in the summer of 1980. It is anticipated that implementation experience and increased use of the language will eventually lead to revisions in the language to provide improved support for the dual objectives of aiding the construction of interactive information systems and encouraging the use of systematic programming methodology.

REFERENCES