Algorithmic Languages, de Bakker/van Vliet (eds.)
© IFIP, North-Holland Publishing Company, 1981, 29–47

PLAIN: An Algorithmic Language for Interactive Information Systems*

Anthony I. Wasserman

Medical Information Science, University of California, San Francisco, CA 94143, U.S.A.

Reind P. van de Riet and Martin L. Kersten

Wiskundig Seminarium, Vrije Universiteit, Amsterdam, The Netherlands

The programming language PLAIN has been designed to provide an effective tool for the systematic construction of interactive information systems. To achieve this goal, PLAIN started with a PASCAL-like framework and incorporated features for the construction of interactive programs, including string handling, pattern specification and matching, input/output, exception handling, and relational data base definition and management. Additional features have also been incorporated to support a systematic approach to programming, with particular attention given to issues of modularity and data abstraction. This paper describes some of the innovative aspects of PLAIN, shows how they have been synthesized into the language, and illustrates how they are used in the creation of interactive information systems.

1. The Design Context of PLAIN

The User Software Engineering (USE) project [25,27,29] was undertaken in 1975 with the goal of providing application developers with a methodology and programming environment to support the systematic creation of interactive information systems. Interactive information systems may be characterized in the following way:

(1) the user repeatedly types some input, e.g., a command;

* This work was supported in part by National Science Foundation grant MCS78-26287 and by The Netherlands Organization for the Advancement of Pure Research (ZWO) (grant 00-62-139). Computing support for text preparation was provided by U.S. National Institutes of Health grant RR-1081 to the UCSF Computer Graphics Laboratory, Principal Investigator: Prof. Robert Langridge.

(2) this input is decoded and parsed; if it is incorrect, a diagnostic message is presented to the user, who then provides alternative input;

(3) the input is subjected to various semantic checks, which may also produce diagnostic messages;

(4) if the input is validated, then some program action is taken, typically an access to or modification of some item(s) in a database, during which time output messages may be provided to the user.

A study of languages and systems available for the construction of interactive programs [23] led to the conclusion that ''the programming languages designed explicitly for interaction do not [have the structure] for creating modular, well-structured software''. With that in mind, the programming language PLAIN (Programming LAnguage for INteraction) became the first tool to be designed in the USE environment.

The design of PLAIN was carried out in parallel with many other language designs, including CLU [11], ALPHARD [34], GYPSY [3], EUCLID [9], and ADA [8]. These languages all have similar objectives (though with differing emphases) of support for data abstraction, support for system modularity, support for program readability, support for testing and/or verification of programs, and the imposition of greater discipline on the programmer. In addition, each of these languages draws heavily on the ALGOL family of languages, particularly PASCAL [33], and on one another. Of these languages, though, only PLAIN addresses the application area of interactive programs and their need for database facilities.

2. PLAIN Design Goals and Features for Interactive Programs

From the outset, the contribution of PLAIN was seen to be not so much the introduction of new language features, but rather a synthesis of features whose *interrelationships* would lead to a useful tool for such application programs. The approach was to make innovations to support interactive programs and to adhere closely to well-understood approaches for other features.

Essential capabilities for the creation of interactive programs were identified, including:

(1) Data base management. The language must deal with data bases and with operations performed on data bases, as well as with more primitive file concepts.

(2) String handling. Interactive programs involve large amounts of text processing, particularly user-program dialogue.

(3) Exception handling. User errors must be expected, but the user should not be adversely affected.

(4) Pattern specification and matching. Many interactive programs depend on a specific text pattern, e.g., a command, to determine program action.

PLAIN provides these capabilities by synthesizing a PASCAL-like framework with necessary features for interactive programs, including the following:

(1) data of type relation and associated relational algebra-like operators that provide a data base management facility;

(2) data of type char and type string, providing for both fixed and variable length strings;

(3) procedure-oriented exception handling, including a time exception;

(4) pattern specification primitives and pattern matching operations;

(5) sequential and direct access files;

(6) input/output operations, possibly involving patterns and files;

(7) access to external objects, such as data bases.

Space limitations make it impossible to give a complete description of the language or even the above features. A complete language description may be found in the Revised Report [30], and explanations of various other aspects of PLAIN may be found in [26,28,31,32]. In this paper, we wish to summarize the motivations behind the design of features for database management, string handling, pattern specification and matching, and exception handling, and then to show how they work together in the construction of interactive information systems.

3. Database Management in PLAIN

A key design goal for PLAIN was to support database management explicitly, rather than working with the lower-level concept of a file as it exists in many languages or relying on traditional approaches to programming languages/data base interfaces. Problems with embedded query languages and with host language interfaces were noted and the need for a *unified* approach to programming languages and data base management was emphasized, so that ''it becomes possible to achieve a level of consistency in syntax and semantics'' and so that ''both type checking and data independence can be achieved'' [24].

Other efforts have been made to extend programming language with database notions [1,2,20,21,22], but these suffer from one or more of the unpleasant problems of language/data management interaction identified in [17], including the difficulty of performing type checking, the tradeoffs between interpretation and compilation, the need to support data abstractions in the database environment, and the unattractive nature of combining nonprocedural data management sublanguages in procedural programming languages.

Two key goals were established for the data definition and management facilities of PLAIN:

(1) Use existing language structures wherever possible. Uniformity of syntax is important so that the data management operations will blend cleanly with other language features. Thus, traditional programming concepts of types and variables should be applicable to database declarations, and the operations on databases should be procedural, in keeping with the procedural nature of the language.

(2) Minimize the number of features added to the language specifically for database management. This objective follows directly from the first objective. Instead of providing a large set of database operations, the decision was made to strive for a compact, yet complete, set of operations. This decision was made with the understanding that the price of the language simplicity would be an increase in the amount of text needed to express complicated data management operations.

These goals pointed clearly toward use of the relational model of data [5] as the basis for database management in PLAIN. From a syntax standpoint, it is possible to exploit the similarity in notation between records and tuples, as was also done by Schmidt [21]. From a language axiomatization standpoint, relations were also the best choice because of their mathematical foundations. Although it was recognized that the relational model is weak in specifying the semantics of the database, it seemed that the potential advantages of the model greatly outweighed the disadvantages for programming language design and implementation.

A data base type definition specifies a structure consisting of an arbitrary number of record occurrences (each called a 'tuple') where each tuple consists of a fixed number of components (called 'attributes'). PLAIN supports two kinds of data base type declarations: **relation** and **marking**. A

relation is a set of tuples and has the property that all tuples are unique; the definition of a relation includes the specification of a non-null set of **key** attributes that uniquely identifies a tuple. A **marking** is a set of referenced tuples from one or more relations. Markings are used to store intermediate results during operations on relations. They play much the same role in database management that temporary variables play in complicated arithmetic calculations. Thus, one may declare variables to be of type relation, using a syntax similar to that for records in PASCAL, or of type marking. Similarly, all attributes must be declared of some type; permissible types are simple types, including scalars, fixed length strings (**type** char[*n*]), and variable length strings (**type** string). Database management operations are provided at the item (attribute) level, the tuple level, and the relation level.

3.1. *Item level operations*

At the lowest level of relation access and manipulation, it is possible to name individual tuples within a relation through a *tuple designator*. If a relation of degree N (N attributes) has M key attributes, where $M \leq N$, specification of values for the M key attributes designates a unique tuple of the relation (or no tuple at all). The syntax for a tuple designator is of the form

```
relation–name [key-value-list]
```

which permits an attribute of a relation to be designated with the notation

```
relation–name [key-value-list] . attribute–name.
```

This mechanism provides two important benefits. First, it is an *associative* addressing mechanism for databases that can be used to obtain single tuples and single attribute values from relations, yielding a clean solution to the problem of converting objects from type **relation** to the underlying attribute type. This makes it possible to perform complete type checking on items in the database, since each attribute must be declared with a type.

Second, it achieves integration at the lowest level between language concepts and database concepts, since the attribute designator may be used in arbitrary expressions throughout a program. Information stored in a relation can be used to declare the dimensions of arrays, to provide a bound on the number of iterations in a loop, or to supply the text for an output message.

3.2. *Tuple operations*

At the tuple level, it is possible to insert tuples and to remove tuples one at a time from a relation. One may simply construct a new tuple by designating a record variable or by specifying values for the attributes of the tuple. The tuple insertion assignment is designated by ':+', while tuple deletion is given by ':–'.

One may also iterate over the tuples of a relation or marking by use of the **foreach** clause in a **loop** statement. The effect of the **foreach** is to permit access to individual tuples in much the same way that iteration is performed over other types of variables.

3.3. *Relation level operations*

High level operations on relations and markings permit the construction of database expressions and the assignment of the expression to a relation or marking variable. The operations supported are selection on a condition (**where**), projection (⇒), natural join on two attributes of the *same type* (**join**), and the set-oriented operations of intersection, union, and difference. The language syntax limits the complexity of database expressions, making it necessary to decompose complicated operations into several steps (perhaps creating markings). The rationalization for this approach is presented in detail in [19].

In summary, PLAIN makes a number of advances toward achieving an effective integration between modern notions of programming languages and facilities for database definition and manipulation. In particular, the procedurality of the operations, the ability to perform type checking on database objects, and the associative access feature are the principal unifying ideas.

4. String Handling and Pattern Matching

Features for string handling and pattern matching were also seen as essential for PLAIN. In addition to providing strings as a data type, it is also necessary to provide tools for checking the conformity of strings to predetermined patterns, particularly for user input. User input must be checked for conformity to the syntactic rules and must then be checked to see that it is meaningful in the context of the input. A numeric input might fail not only for reasons of invalid characters, but also for arithmetic overflow, arithmetic underflow, or because the numeric value was not a meaningful value for the corresponding data element.

PLAIN provides for the built-in simple type char (as in PASCAL, ADA, and other similar languages) and for the built-in structured type string. Variables of type char or array of char permit *fixed length* string processing, while variables of type string provide for *variable length* strings. String concatenation is provided with the binary operator '++' returning an array of type char or a string, depending on the operands. String contains is provided with the operator '$'; for strings *a* and *b*, the value of *a*$*b* is true iff the string *b* is contained in *a*. String follows (lexical ordering) is provided with the operator '>>'; for strings *a* and *b*, the value of *a*>>*b* is true iff the lexicographic order of *a* follows *b* in the ASCII collating sequence. The remaining string operations are provided through functions, including length, string searching, substring extraction, insertion, deletion, and replacement.

The key observation for successful handling of user input was to see user inputs as *languages* subject to various kinds of syntactic and semantic rules. In short, one can define a grammar that describes the valid syntax for a given user input.

From that point, it became possible to identify some goals for the inclusion of pattern processing mechanisms in PLAIN, including the following:

(1) simplicity, comparable to that of MUMPS patterns, rather than to the more powerful and general SNOBOL 4 patterns;

(2) the pattern facilities should simplify not only the syntactic checking of user input, but also any subsequent semantic checking;

(3) certain common patterns should be predefined, i.e., 'built into' the language;

(4) the pattern facilities should be usable for control of program output as well, so that it would not be necessary to include a totally separate output management mechanism;

(5) the power of the pattern specification and pattern matching should make it possible to recognize a large class of possible user inputs, such as specified by a context-free (Type II) grammar.

The key idea behind pattern specification and matching in PLAIN was to provide a simple mechanism whereby the programmer could define the grammar for a language, and then use built-in operators to determine the match between a pattern and a string defined by the grammar.

The **pattern** declaration facility permits patterns and pattern sets to be declared. In a pattern, all elements are required for pattern matching, while in a pattern set, only one of an alternative list of patterns is required for matching. In both cases, the declarations are static and, unlike SNOBOL4, it is not possible to create patterns dynamically.

A pattern is composed of a list of pattern elements, which may be string literals, subranges of characters, or the name(s) of other patterns, including pattern sets. Each pattern element may be preceded by a repetition count, which may be definite (a positive integer), or indefinite. The indefinite cases are '*' for zero or more instances, and '.' for one or more instances. In the absence of a repetition count, a default count of one is assumed.

Many common pattern matching cases are covered by predefined patterns in PLAIN. These patterns include A for alphabetic characters, N for numerics, P for punctuation, I for a (signed) integer, X for blank, and S for string, which matches anything.

A simple example of a pattern definition is given by the patterns

```
bookid = (10N);
chkout = ('out', . X, bookid, '/', I)
```

they would match the string 'out 9023633407/12554'. Note that chkout contains the name of another pattern, bookid, as well as string literals and predefined pattern names.

Such pattern names can be combined into other patterns and pattern sets. Thus, the pattern chkout might be an alternative in the pattern set

```
command = [chkout, checkin, reserve, status, quit]
```

where each of the patterns represents the permissible user input for the various commands in the system. (If a string matches more than one pattern in the pattern set, the leftmost matching alternative is selected.)

A more complex example can be given by combining patterns and pattern sets to define a class of strings representing permissible ways to input a date, showing that patterns and pattern sets may be nested.

```
date = [form1, form2, form3];
form1 = (one-or-two, sep, one-or-two, sep, two-or-four);
form2 = (one-or-two, 1X, month, 1X, two-or-four);
form3 = (month, X, one-or-two, ',', X, 4N);
```

```
one-or-two = [1N, 2N];
two-or-four = [2N, 4N];
sep = ['/', '-', '.'];

month = [longenglish, shortenglish];

{intermediate months omitted in the next two pattern sets}
longenglish = ['January', 'February', ..., 'December'];
shortenglish = ['Jan', 'Feb', ..., 'Dec'];
```

Note that, from a syntactic standpoint, this pattern specification handles most of the forms of giving the date in the English language. Among the strings accepted by date are '2/2/1972' and '27.08.80', corresponding to form1, '4 July 1778' and '22 Nov 63', corresponding to form2, and 'June 6, 1944', corresponding to form3.

Two more observations may be made about this scheme:

(1) the availability of the built-in patterns and the ability to include string literals eliminates the need for a separate lexical analysis tool; primitive text units, i.e., tokens, can be placed within the patterns and pattern sets;

(2) the pattern declaration mechanism permits one to specify an arbitrary context free language, since patterns may contain arbitrarily many patterns and pattern sets with a completely recursive capability;

PLAIN contains two pattern matching operators: one for determining the *exact* match between a string and the pattern specification, and one for determining whether the pattern can be found anywhere in the string. Accordingly, two binary pattern matching operators, pattern match (?=) and pattern contains (?) were defined. The left-hand operand for each is a string; the right-hand operand is the name of a pattern. The pattern match operator '?=' returns true iff the pattern matches the entire string. The pattern contains operator '?' returns true iff the pattern matches a substring.

For example, if one uses the patterns form1 and form2 declared in conjunction with the date example above with the variables sa, sb, and sc as follows:

```
var sa, sb: string; sc: char[16];
```

with the following assignments

```
sa := '04/02/77';
sb := '27 Aug 72';
sc := 'Received 6-11-66';
```

then sa? = form1 is true, sc? = form1 is false, sb?form2 is true, sa? = form2 is false, and sc?form1 is true.

The binary operators **match** and **contains** are used with the **case** statement to allow branching based on pattern matching. These operations return a pattern name if the case expression, which must be of type string or array of char, is successfully found in the designated pattern set. The pattern name is then used as the case selector, as follows:

```
case input match month of {assume input declared of type string}
  when longenglish, shortenglish: english-date (display)
  when others: unknown-date (display)
end case
```

The remaining necessary capabilities are to be able to split a given string into its components and to combine two or more shorter strings into a longer string, based on patterns. The **split** and **combine** operations, respectively, provide these capabilities in PLAIN. The **split** operation apportions a string value to one or more variables, possibly discarding part of the string. The **combine** operation assembles two or more expressions into a single string value according to a specific pattern. The assembled string value is then assigned to a variable. A given string may be split or combined according to different patterns as necessary at any level of the pattern matching. Such a facility is particularly useful for processing of command languages.

With this set of pattern matching capabilities, it is possible to make effective use of the pattern facility in conjunction with the string handling features and to carry out the input/output and string processing that is essential to the effective construction of interactive programs. These string-handling and pattern matching features are described at greater length in [31].

5. Exception Handling

The ability to anticipate and to handle non-standard situations is essential to the construction of reliable systems. Thus, the specification for a system may provide not only for 'normal' conditions, such as proper operation of the hardware and meaningful user input, but also for abnormal conditions, such as hardware errors and arithmetic overflow, describing the action to be taken if these conditions arise during system operation.

Accordingly, exception-handling mechanisms have been designed and implemented in many programming languages, including PL/I [14], MESA [16], CLU [12], and ADA [13]. Also, there have been proposals made for the inclusion of exception-handling mechanisms in languages and systems, and for the specification and implementation of exceptions [4, 7, 10, 15, 18].

The goals established for the exception-handling features of PLAIN are the following:

(1) Association of exceptions – it should be possible to associate exception handlers with specific exceptions and to bind this association at the statement level in the source program; it should also be possible to attach this association to a group of statements, such as a procedure body.

(2) Fielding of exceptions – it should be possible to pass an exception from the environment in which it was signalled to any previous level of invocation for handling.

(3) Orderliness – it should be possible to carry out normal shutdown procedures in the event of a fatal error, permitting, insofar as possible, the closure of open files, and the generation of messages.

(4) Grouping of exceptions – it should be possible to define a group of exceptions that are to be treated similarly under certain conditions.

(5) Programmer-defined vs. built-in exceptions – the exception-handling scheme should support both the handling of built-in exceptions and the definition, signalling, and handling of programmer-defined exceptions.

We designed a procedure-oriented approach to exception-handling for several reasons:

(1) the use of a call provides a constraint upon control flow, since control can return from the handler to its invocation point;

(2) the same handler can be invoked for several different exceptions or for several different instances of the same exception;

(3) the use of procedures serves to separate the exception-handling code from the remainder of the code;

(4) data coupling is made more visible through the parameter passing mechanism of procedure calls.

PLAIN provides built-in exceptions for commonly occurring exceptional program conditions, and permits the declaration of user-defined exceptions. Built-in exceptions are raised automatically by the runtime system, while user-defined exceptions must be explicitly raised. The **signal** statement is used to signal a condition or event that needs special handling. The execution of a signal statement causes the program unit being executed to be immediately terminated at the point of the signal, with control returned to the invoker of the unit with the named exception as an active exception in the invoking context.

Program statements may optionally contain an exception part, which contains a list of exceptions and the names of associated exception-handling routines, called handlers. A **handler** is like a **procedure** in that it may be invoked from numerous places within a program and that standard parameter passing rules apply. Handlers are also like procedures in that there are no restrictions upon declarations or statement types; in other words, any type of computation may be performed within a handler.

The handler attempts to perform whatever actions are necessary to take care of the exception that caused it to be invoked and then returns to the point of invocation. There are four possible ways in which the computation may then proceed:

(1) the exception has been *cleared* and normal program execution may continue;

(2) the exception has not been handled completely and is then passed to the invoker of the routine in which the exception occurred, causing the termination of the routine;

(3) the exception has been *cleared* and the program segment (statement or compound statement) associated with the exception is *retried;*

(4) a different exception is returned to the location where the first exception occurred, which must be handled before handling of the first exception can be completed.

This mechanism permits exceptions to be passed up the activation chain and permits them to be handled at each level until they are cleared or until the absence of a programmer-defined handler causes the system-defined default handler to be invoked, thereby causing program termination.

The **clear** statement clears the exception that caused the invocation of the handler. The **retry** statement clears the active exception and then returns control to the beginning of the statement from which the handler was invoked, attempting to restore the environment which then existed. (Note that not all these effects, e.g., input/output and database updates, can be feasibly undone.) The **clear** and **retry** statements may only be used within a **handler**.

There are three built-in user-callable handlers that facilitate the use of this mechanism:

(1) abort, which signals the unclearable fail exception to the invoker of the currently executing routine;

(2) continue, which clears the active exception and results in continued execution of the currently executing routine;

(3) pass, which passes the active exception to the invoker of the currently executing routine.

Although this mechanism is more complex than some of those provided by other languages, it also provides some facilities that are not present in other exception-handling schemes, but that are important for interactive programs, including:

(1) exception handling is preemptive so that executions may be interrupted and stacked, making it possible to react to an exception while handling another;

(2) the pass handler makes it possible to pass exceptions through successive function/procedure invocation levels to a point at which the exception is meaningful in terms of the intended function; a low-level exception may or may not signify an error condition;

(3) the **retry** statement (see above) makes it very easy to program the common situation of asking the user to repeat input that does not conform to expected patterns.

These features may be illustrated by considering an example of user/program dialogue, such as asking the user to type in a valid bookid as defined above. In this example, the program reads a variable input according to the bookid pattern. An exception part is associated with the **read** statement to handle the various exceptional conditions that might arise. If the user transmits the break or the escape character, the handler break-message will be invoked. An exception can then arise while executing the **read** statement in break-message.[1]

```
var input: char[10];
  limit: integer;
  .
{limit is set to the number of tries we are willing to make}
  .
read[bookid]: input![ioerr: abort; patform: ask-again;
       break, escape: break-message];
```

[1] The exception parts shown in this example are intentionally thorough. In practice, the thoroughness of the exception parts would depend on the desired robustness of the program.

```
   {ask-again and break-message are user-defined handlers}
   .
   .
handler yes-or-no;
imports limit: modified;
begin
   if limit > 0 then
      write 'Please answer yes or no';
      limit := limit-1;
      retry {causes read in break-message to be repeated}
   end if;
   write 'The program is being terminated';
   signal fail;
end yes-or-no;
   .
   .
handler break-message;
var answer: string;
pattern yes-no = ['yes', 'no'];
begin
   write 'Do you wish to terminate the program? ...';
   read [yes-no]: answer![patform, time: yes-or-no];
   if answer = 'yes' then signal fail end if;
   retry {causes read in main program to be repeated}
end break-message;
   .
   .
handler ask-again;
begin
   write 'Invalid book number. Please try again.', \ n;
   retry {causes read in main program to be repeated}
end ask-again;
```

It can be seen from this example that a significant portion of the code in an interactive system must be devoted to management of the user/program dialogue, particularly if one wishes to create user-centered systems that are easy to learn and easy to use [29]. Because careful handling of user errors is critical in such an environment, the exception-handling mechanism is particularly important, and the exception handling features of PLAIN were designed with this requirement in mind.

6. Interactive Information Systems in PLAIN

The combination of database management, string handling, pattern matching, and exception handling within the framework of a language to support and encourage systematic programming is the most significant contribution made by PLAIN. These features work together most effectively in the construction of interactive information systems.

A program schema for the typical interactive information system characterized in the introduction is as follows:

```
program iisschema;
external {names of external objects used by program, such
     as databases and files}
var input: string;
     {other global declarations, including exceptions}
pattern cmdset = [com1, com2, com3, ..., comN, quit];
     com1 = (...);
     com2 = (...);
       .
       .
     comN = (...);
     quit = ('quit');
begin
  loop
     read input![ioerr: abort];
     {terminate on hardware I/0 error}
     case cmdset match input of
        when com1: action1 (...) {parameter list}
        when com2: action2 (...)
          .
          .
        when comN: actionN (...)
        when quit: exit
        when others: write 'illegal command' {pattern match failed}
     end case;
  repeat;
  write 'byebye'
end iisschema.
```

Each of the actions associated with the commands may perform additional decoding or analysis of the command, perhaps splitting the command string into substrings via the string functions or the split operation, and will then carry out the action implied by the user command.

Consider the example of a library information system using the pattern set command and the pattern chkout shown above. The procedure book-checkout would include the following code:

```
procedure bookcheckout (input: string);
imports book, cardholder: readonly; checkout: modified;
  {book cardholder, and checkout defined external to bookcheckout
    as relations in library data base}
var booknum: char [10]; copyno: 1..100; datedue: char [4];
  oldcount, person: integer;
    .
  {handlers bad-book, bad-card, bad-copy, and dberr not shown}
    .
begin
  (#, #, booknum, #, person) := split [chkout]: input;
  {check validity of ISBN number and cardholder}
  assert book [booknum] in book ![assertion: bad-book];
  assert cardholder [person] in cardholder ![assertion: bad-card];
  write 'Copy number:';
  read copyno ![patform, range: bad-copy];
  write copyno;
  {compute due date and save in variable datedue}
    .
  {update set of checkouts}
  checkout :+ [⟨booknum, person, copyno, datedue⟩]![fail, duplicate:
    dberr];
end bookcheckout;
```

This brief example shows how these features combine to incorporate the facilities for interactive systems with such important features as assertion checking for semantic integrity of databases and powerful control structures. These features are easily used in a similar fashion for other similar kinds of examples and greatly simplify the problems of writing this class of programs.

Because of space limitations, we have omitted discussion of the PLAIN **module** facility, which provides facilities for data abstraction. The **module** facility is extremely useful in PLAIN, since it permits type extension of database types as well as other types. It is similar in most other respects to data abstraction facilities found in other modern languages, e.g., CLU.

7. Conclusion

The design of PLAIN combines modern programming language design concepts for creating well-structured programs with an integrated set of innovative features to support the implementation of interactive information systems.

Among the most significant aspects of these innovative features are:

(1) the associative addressing capability of relations, making it possible to access and modify individual data base items, to use data base items routinely throughout the program text, and to perform conversion between data base types and the underlying types of their attributes;

(2) the pattern and pattern set specification facility, making it possible to specify a context-free grammar, using the pattern-matching features to carry out the lexical and syntactic aspects of the text processing;

(3) the procedure-oriented exception-handling scheme, which makes it practical for the programmer to anticipate user errors and to build robust programs that handle these errors.

These features are largely orthogonal and do not interfere with one another in using or implementing the language, even though they are typically used together in practice.

Experience with PLAIN and with other modern programming languages indicates, subjectively at least, that it is much easier to implement interactive information systems with PLAIN than with any of the languages previously used for such applications or any of the other modern languages designed to support systematic programming. Work is continuing to use PLAIN to implement various application systems and software tools, as well as to develop implementations of PLAIN for a variety of execution environments.

References

[1] E. Allman, M.R. Stonebraker and G.D. Held, Embedding a relational data sublanguage in a general purpose programming language, Proc. Conf. on Data: Abstraction, Definition, and Structure, ACM SIGPLAN Notices 11 (Special Issue) (1976) 25–35.

[2] T. Amble, K. Bratsbergsengen and O. Risnes, ASTRAL: a structured and unified approach to data base design and manipulation, in: G. Bracchi and G.M. Nijssen (Eds.), Data Base Architecture (North-Holland, Amsterdam, 1979) pp. 257–274.

[3] A.L. Ambler et al., GYPSY: a language for specification and implementation of verifiable programs, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12(3) (March 1977) 1–10.

[4] D.M. Berry, R.A. Kemmerer, A. von Staa and S. Yemini, Toward modular verifiable exception handling, Comput. Languages 5 (2) (1980).

[5] E.F. Codd, A relational model of data for shared data banks, Comm. ACM 13 (6) (June 1970) 377–387.

[6] E.F. Codd, Further normalization of the data base relational model, in: R. Rustin (Ed.), Data Base Systems, Courant Computer Science Series, Vol. 6 (Prentice-Hall, Englewood Cliffs, 1972) pp. 35–63.

[7] J.B. Goodenough, Exception handling: Issues and a proposed notation, Comm. ACM 18 (12) (December 1975) 683–696.

[8] J. Ichbiah et al., Reference manual for the ADA programming language, Advanced Research Projects Agency, U.S. Department of Defense (July 1980).

[9] B.W. Lampson et al., Report on the programming language EUCLID, ACM SIGPLAN Notices 12 (2) (February 1977) 1–79.

[10] R. Levin, Program structures for exceptional condition handling, Ph.D. Dissertation, Computer Science Department, Carnegie Institute of Technology, Pittsburgh, PA (1977).

[11] B. Liskov et al., CLU reference manual, Lecture Notes in Computer Science, Vol. 114, Springer, Berlin 1981.

[12] B. Liskov and A. Snyder, Exception handling in CLU, IEEE Trans. Software Engrg. SE-5 (6) (November 1979) 546–558.

[13] D.C. Luckham and W. Polak, ADA exception handling: an axiomatic approach, ACM Trans. Programming Languages and Systems 2 (2) (April 1980) 225–233.

[14] M.D. McLaren, Exception handling in PL/I, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12 (3) (March 1977) 101–104.

[15] P.M. Melliar-Smith and B. Randell, Software reliability: the role of programmed exception-handling, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12 (3) (March 1977) 95–100.

[16] J.G. Mitchell, W. Maybury and R. Sweet, MESA language manual, version 5.0, XEROX Palo Alto Research Center, Palo Alto, CA (1979).

[17] C.J. Prenner and L.A. Rowe, Programming languages for relational database management, Proc. AFIPS 1978 NCC, Vol. 47, pp. 849–855.

[18] B. Randell, System structure for software fault tolerance, IEEE Trans. on Software Engrg. SE-1, (2) (June 1975) 220–232.

[19] R.P. van de Riet, A.I. Wasserman, M.L. Kersten and W. de Jonge, High level programming features for improving the efficiency of a relational database system, ACM Trans. on Database Systems 6 (3) (1981), in press.

[20] L.A. Rowe and K. Shoens, Data abstraction, views, and updates in RIGEL, Proc. of ACM 1979 SIGMOD Conference, Boston, MA, pp. 71–81.

[21] J.W. Schmidt, Some high level constructs for data of type relation, ACM Trans. on Database Systems 2 (3) (September 1977) 247–261.

[22] J.E. Shopiro, THESEUS – a programming language for relational databases, ACM Trans. on Database Systems 4 (4) (December 1979) 493–517.

[23] A.I. Wasserman, Online programming systems and languages: a history and appraisal, Techn. Rep. No. 6, Laboratory of Medical Information Science, University of California, San Francisco CA (1974).

[24] A.I. Wasserman, Embedding database management operations in programming languages, Conference Digest – IEEE COMPCON Spring 1976, pp. 79–82.

[25] A.I. Wasserman, USE: a methodology for the design and development of interactive information systems, in: H.-J. Schneider (Ed.), Formal Models and Practical Tools for Information Systems Design (North-Holland, Amsterdam, 1979) pp. 31–50.

[26] A.I. Wasserman, The data management facilities of PLAIN, Proc. ACM 1979 SIGMOD Conference, Boston, MA, pp. 60–70.

[27] A.I. Wasserman, Software tools and the user software engineering project, in: W.E. Riddle and R.E. Fairley (Eds.), Software Development Tools (Springer Verlag, Heidelberg, 1980) pp. 93–113.

[28] A.I. Wasserman, The design of PLAIN – support for systematic programming, Proc. AFIPS 1980 NCC, Vol. 49, pp. 731–740.

[29] A.I. Wasserman, User software engineering and the design of interactive systems, Proc. 5th International Conference on Software Engineering, San Diego, 1981, pp. 387–393.

[30] A.I. Wasserman et al., Revised report on the programming language PLAIN, ACM SIGPLAN Notices 16 (5) (May 1981) 59–80.

[31] A.I. Wasserman and T. Booster, String handling and pattern matching in PLAIN, Techn. Rep. No. 50, Laboratory of Medical Information Science, University of California, San Francisco, CA (1981).

[32] A.I. Wasserman and M. Dippé, Design and evaluation of a procedure-oriented exception-handling mechanism, in preparation.

[33] N. Wirth, The programming language PASCAL, Acta Inform. 1 (1) (1971) 35–63.

[34] W.A. Wulf (ed.), An informal description of ALPHARD (preliminary), Techn. Rep. CMU-CS-78-105, Department of Computer Science, Carnegie-Mellon University (1978).