Program forms and program form analysers for high-level structured design*

by JAYASHREE RAMANATHAN
University of Houston
Houston, Texas

and

MEERA BLATTNER
Rice University
Houston, Texas

INTRODUCTION

Structured programming is the single most important technique currently used in developing software which is both reliable and inexpensive. It has been observed by various researchers that programmer productivity can be vastly improved if the development of software is split into two, equally important phases.

1. The design phase, and
2. The implementation phase.

In this paper we present PSEUDO LANGUAGE (PL)—a program design tool enforcing the functional programming concept discussed in References 4, 10, 13, 17, 21, 22 and 28. A PSEUDO LANGUAGE PROCESSOR (PLP) is also described. The PLP is a translator which examines source programs in PL and provides a listing of these programs together with a variety of messages. These messages can be used by the programmer both during the design and the implementation phase.

A PL program is a program form which represents a broad class of possible implementations in any of the standard programming languages. PL program forms resemble pidgin English and are therefore very readable. It is easy to program in PL since the programmer can ignore the messy details necessitated by actual implementation languages (FORTRAN, PASCAL, PL1, COBOL, etc.). An important characteristic of the PL design language is that it requires the programmer to explicitly identify the functional components of the programming task at hand. The implementation details of these functional components may be ignored by the programmer. Instead the programmer can concentrate on the logical interaction between these components. PL is designed to enforce structured programming techniques. A PL program can serve as the documentation for the implementation version of the program. The advantages of PL mentioned above should also result in better communications between programmers in a team and contribute towards increased programmer productivity.

The PLP described here is constructed realizing that it is cheapest to correct errors during the earlier stages of software development. This is mainly due to the fact that fewer corrective changes have to be made to debug a design program. The messages generated by the PLP indicate errors in the PL program forms and list the functional components in the program form which have already been implemented. In order for the PLP to provide these messages, the PL program forms must have a recognizable structure. The definition of PL given here attempts to strike a balance between allowing program forms with too little syntactic structure and too much syntactic structure (as in implementation language programs).

Existing design tools for structured programming can be generally categorized as a variation of flowcharting (for examples refer to References 6, 8, 9 and 26) or as English sentences with keywords which identify the control flow in the program. Both types of design tools concentrate on the control logic of the programming task. PL programs also contain fixed control structures and in addition restrict the English sentences to "commands." This restriction on the English sentences forces the programmer to identify the functional components of the programming problem. At the same time, this restriction allows the PLP to employ algorithms for validating PL programs. PLP performs extensive static analysis on PL program forms and uses this analysis to print out messages that can be used by the programmer for validating and debugging the program forms.

The next section introduces the syntax for PL, and the third section discusses how PL enforces structured programming. PLP functions are outlined in the fourth section and the final section presents conclusions and suggestions for future work.

* This research was supported in part by NSF Grant Number MCS77-02470.
DEFINITION OF PSEUDO LANGUAGE (PL)

PL requires that the declarations precede the "executable" statements as in FORTRAN or PASCAL. Using somewhat informal syntax notation we have

a. (program) → (introduction) (body)
b. (introduction) → (directives to PLP) (title)
   INPUT PARAMETERS: (noun list)
   OUTPUT PARAMETERS: (noun list)
   READ INTO: (noun list)
   PRINT FROM: (noun list)
   DICTIONARY: (noun description list)
c. (noun list) → (noun list), (noun)
   / (noun)
d. (noun description) → (noun)
   / (noun) ATTRIBUTES: (attribute list)
   / (noun) ATTRIBUTES: (attribute list)
   INITIAL VALUE: (constant list)
e. (noun description list) → (noun description list);
   (noun description)
   / (empty)
f. (noun) → (identifier)
   / (subscripted identifier)
   / (structured noun)
   / (empty)
g. (structured noun) → (structured noun) (constant) (identifier)
   / (constant) (identifier)
h. (attribute list) → (attribute list), (attribute)
   / (attribute)

The syntax of the introduction may be modified to allow declarations for general data types. We now give an example of an (introduction) for a Bubble Sort program. Comments on any line are preceded by a double slash (//).

Example

//introduction
SORT IN ASCENDING ORDER
INPUT PARAMETERS: SIZE_OF_TABLE, TABLE
OUTPUT PARAMETERS: TABLE
READ INTO:
PRINT FROM:
DICTIONARY:
SIZE_OF_TABLE ATTRIBUTE: INTEGER;
NO_INTERCHANGE ATTRIBUTE: FLAG
FIRST_ITEM ATTRIBUTE: POINTER; // TO TABLE
SECOND_ITEM ATTRIBUTE: POINTER; // TO TABLE
TABLE ATTRIBUTE: ARRAY;
EACH_PAIR

As illustrated in the previous example, (directives to PLP) can be empty. ATTRIBUTES; can be followed by any number of words describing the noun. The syntax for (constant list), (identifier) and (constant) are as in standard programming languages. The syntax for the (body) of a PL program follows. We shall think of the body as being composed of statement forms which specify the control flow in the implementation program and statement forms which specify the executable, functional components of the implementation program. We shall loosely say that the body of the program form is executable.

i. (body) → BEGIN (statement list) END
j. (statement list) → (statement list); (statement)
   / (statement)
k. (statement) → (assignment)
   / (compound statement)
   / (if statement)
   / (case statement)
   / (while statement)
   / (repeat statement)
   / (for statement)
   / (cycle statement)
   / (with statement)
   / (cobegin-coend statement)
   / (I/O statement)
   / (expression)
   / (command)

The fundamental executable components of a program...
Program Form Analysers for High-Level-Structured Design

form are assignments, commands and expressions. The control structures specify the manner in which the fundamental components are to be executed. The non-terminal (statement) goes to the fundamental components and to the control structures containing sequences of fundamental components. Standard (PASCAL-like) syntax details of the control structures are given in the appendix. It should be noted that concurrent execution of statements may be specified by PL. Hence PL allows the design of program forms for a wide variety of applications. In fact, (statement) may be sent to any of the structures supported by the various languages currently in use. Syntax for exit statements, GOTOs and labels may be easily added to the above grammar but this is not done in the grammar presented here.

The fundamental executable component of interest is the command. The following productions give its syntax.

1. (command) (verb) / (verb) (comments) (noun)

Example PL program form for SORT

```plaintext
BEGIN
  IF SIZE_OF_TABLE > 1
  THEN
    WHILE NO_INTERCHANGE = 0
    DO
      FOR EACH_PAIR = 1 TO SIZE_OF_TABLE - 1
        GET FIRST_ITEM IN TABLE;
      END
      IF FIRST_ITEM IN TABLE > SECOND_ITEM IN TABLE
      THEN
        INTERCHANGE FIRST_ITEM IN TABLE AND SECOND_ITEM IN TABLE;
        NO_INTERCHANGE = 1
      END
    OD
  END
END
PRINT TABLE
```

Examine the first command GET FIRST_ITEM IN TABLE. This command begins with the verb GET and it operates on the nouns FIRST_ITEM and TABLE. The word IN functions as a comment which makes the command easier to read. Similarly, INTERCHANGE is the verb in the last command of the above program. INTERCHANGE operates on the nouns FIRST_ITEM, SECOND_ITEM and TABLE. More specifically,

```plaintext
INTERCHANGE FIRST_ITEM IN TABLE AND SECOND_ITEM IN TABLE;
```

INTERCHANGE is the abstraction of a routine which has as inputs FIRST_ITEM, TABLE, and SECOND_ITEM.
The actual FORTRAN implementation for even the simple sort program is much harder to read. The FORTRAN subroutine is given below.

Example

```
SUBROUTINE SORT (ISIZE, TABLE)
  DIMENSION TABLE (100)
  IF (ISIZE.LE.1) GO TO 99
  INTER=0
  DO 20 1=1, ISIZE-1
    IF (TABLE(I).LE.TABLE(I+1)) GO TO 20
    INTER=I
    TEMP=TABLE (I)
    TABLE (I)=TABLE (I + 1)
    TABLE (I)=TEMP
  20 CONTINUE
  IF (INTER. EQ. 1) GO TO 2
  99 WRITE (6,30) (TABLE (1),1=1, ISIZE)
  30 FORMAT (5x2, 10 (F8.3, 2x»
  STOP
END
```

Note the PL program form for the above SORT given in the next example suppresses the details of TABLE, INTERCHANGE, GET and PRINT.

Example

```
SORT IN ASCENDING ORDER
INPUT PARAMETERS: SIZE_OF_TABLE, TABLE
OUTPUT PARAMETERS: TABLE
PRINT FROM: TABLE
DICTIONARY: SIZE_OF_TABLE; NO_INTERCHANGE;
FIRST_ITEM: SECOND_ITEM: TABLE:
EACH_PAIR
BEGIN
  IF SIZE_OF_TABLE>1
  THEN
    WHILE NO_INTERCHANGE
      DO NO_INTERCHANGE=0
        FOR EACH_PAIR=1 TO SIZE_OF_TABLE-1
          DO GET FIRST_ITEM IN TABLE;
          GET SECOND_ITEM IN TABLE;
          IF FIRST_ITEM IN TABLE> SECOND_ITEM IN TABLE
            THEN BEGIN
              INTERCHANGE FIRST_ITEM IN TABLE AND SECOND_ITEM IN TABLE;
              NO_INTERCHANGE=1
            END
            PRINT TABLE
      END
  END
END
```

STRUCTURED PROGRAMMING USING PSEUDO LANGUAGE

Note that the PL SORT program form implies the hierarchical structure of Figure 1. The PL program form is not concerned with the details of TABLE or its elements. In an implementation, the TABLE may be a file and its elements may be employee records. Also, GET and INTERCHANGE may be complicated, machine dependent subroutines. Here again the PL SORT program form is not concerned about
the details of algorithms implementing the functions GET and INTERCHANGE. SORT is written assuming GET and INTERCHANGE work. The logic of the SORT program form is easy to check out. Therefore, if GET and INTERCHANGE work reliably, then SORT works reliably. To reiterate, SORT TABLE might be the main goal. This goal is refined by the PL SORT program form which in turn calls for two functions, GET and INTERCHANGE. The data structures (or nouns) on which the two functions operate are completely specified in the commands in which they appear in the SORT program.

PL forces the programmer to identify the verbs (or functional components) of the program during the design phase. This ensures the program form can be implemented. Design languages which allow flexible English language statements cannot ensure that the design algorithm can in fact be implemented by a process. Another feature that PL has in common with the existing design tools is that control structures must be identified.

The functional components in the PL SORT program form may themselves be implemented by other PL program forms calling on still other functions (verbs). This stepwise refinement can continue until a level is reached where all the verbs are implemented in a desired implementation language. It is easy to see that the resulting implementation program should be well structured.

### PSEUDO LANGUAGE PROCESSOR (PLP)

PL syntax is designed so that PL programs are amenable to both structural and symbolic (static) analysis. The PL currently being implemented is divided into three logical components—the scanner, the structure analyzer, and the message generator. PLP is designed to generate a variety of messages. The classes of messages generated are listed below together with a brief discussion of how the messages in each class are generated.

- **Messages detecting violations of standard design practices**—The structural analysis of the PL program, by the parser, detects omissions in the introduction and improper use of control structures in the body.\(^1,27\)
- **Cross-reference tables**—Tables indicating variable usage are determined during the three PLP phases and printed out appropriately.
- **Global references are listed**—All nouns in the introduction which have a global attribute are listed. This allows uses of critical, global resources to be monitored.
- **Messages indicating anomalies in the use of variables**—PLP uses various current techniques of high-level data flow analysis techniques on the parse tree to detect anomalies in the use of variables.\(^2,5,12,14-16,18,19,24,25\) The appearance of a variable in the introduction identifies it as a noun. This information is used by the semantic routines to distinguish the nouns in a command from other words which serve as comments. The first word in a command is always a verb. If a noun follows the keyword RETURN it is taken to be defined in the subprocess named by the verb. See Figure 2 for an illustration.

The statement

```
IF FIRST_ITEM IN TABLE > SECOND_ITEM IN TABLE
THEN INTERCHANGE FIRST_ITEM
    IN TABLE AND SECOND_ITEM
    IN TABLE
```

Suppose the verb INTERCHANGE is already implemented by a FORTRAN (or PASCAL) or even another PL routine in the PLP library. PLP will list INTERCHANGE as a possible implementation of the verb. The programmer can examine a copy of INTERCHANGE. After an examination, if the programmer so directs, the library implementation of INTERCHANGE can be used as shown below. PLP library implementation of INTERCHANGE may contain the statements

```
TEMP=FIRST_ARGUMENT
FIRST_ARGUMENT=SECOND_ARGUMENT
SECOND_ARGUMENT=TEMP
```

Based on the programmer’s directive, INTERCHANGE in the PL statement above will be considered as a call to the library routine called INTERCHANGE and the resulting statement synthesized will be
The PLP facility just discussed is especially useful when the verbs are implemented by complicated routines. In such cases considerable programming effort is saved and the program itself is more structured. A drawback of this PLP facility is that the programmer may use a verb which is implemented and contained in the PLP library under a different name. To overcome this, a document containing the implemented verbs may be provided to the programmers.

- **Messages on the misuse of a sub-process**—The introduction of each sub-process provides a detailed description of the input parameters and the output parameters of the sub-process. When PLP links together sub-processes or when it recommends (or uses) a particular implementation of a verb, it will check to see whether the actual parameters in the calling sub-process are compatible with the formal parameters in the definition of the sub-process. At the same time PLP will check access rights of each sub-process. These checks will be made semantically by examining the introduction of the two sub-processes involved.³

The last two PLP capabilities listed are currently being designed. The remaining capabilities are being implemented.

Finally, PLP uses the structural analysis of a PL program for creating an output listing with proper indentations.

**CONCLUSIONS**

A design language called PSEUDO LANGUAGE (PL) has been presented. Programs written in PL are called program forms. Program forms avoid implementation details and are therefore easily readable and understandable. PL also forces the programmer to identify the control structures as well as the functional components of the program system during the design phase.

The cost of finding an error in software increases as the software development comes nearer to completion. Errors found during specification are relatively inexpensive to correct as compared with errors found during total system integration.²³ The PSEUDO LANGUAGE PROCESSOR (PLP), currently being implemented, is an automatic tool for analysing specifications written in PL and printing out messages that indicate

- Violations of good design practices
- Errors
Program Form Analysers for High-Level-Structured Design

• Incorrect interfacing between programs
• Existing, potentially useful sub-processes that the programmer can use

Current research involves developing the theoretical framework for modeling the translation of PL program forms to implementation programs. Work is also proposed on the PL syntax. There may be many different implementations of a verb in the PLP library. Therefore, the syntax should allow a verb to be further qualified. For example, BUBBLE.SORT implementation would differ from MERGE.SORT. PLP can be designed as an interactive system which aids program form validation and implementation program synthesis. Valuable statistics on the usage of verb implementations can be obtained by the PLP. This would point to verbs that perhaps could be implemented by hardware.

REFERENCES


APPENDIX

k.0 (statement) → (assignment) / (compound statement) / (if statement) / (case statement) / (while statement) / (repeat statement) / (for statement) / (cycle statement) / (with statement) / (concurrent statement) / (command) / (I/O statement)

k.1 (compound statement) → BEGIN (statement list) END

k.2 (if statement) → IF (expression) THEN (statement) ELSE (statement)

k.3 (case statement) → CASE (expression) OF (constant) : (statement list) ;

k.4 (while statement) → WHILE (expression) DO (statement) OD

k.5 (repeat) → REPEAT (statement list) UNTIL (expression)

k.6 (for statement) → FOR (identifier) = (expression) TO (expression) DO (statement) OD

k.7 (cycle statement) → CYCLE (statement list) END
k.8 (with statement) \( \rightarrow \text{WITH} \ (\text{variable}), \)
\( \ (\text{variable}), \ldots \ DO \)
\( \ (\text{statement}) \)

k.9 (command) as in text

k.10 (concurrent statement) \( \rightarrow \text{COBEGIN} \ (\text{statement list}) \)
\( \text{COEND} \)

k.11 (I/O statement) \( \rightarrow \text{PRINT} \ (\text{noun list}) \)
\( \text{READ} \ (\text{noun list}) \)

The terminals of PSEUDO LANGUAGE are the underlined symbols, the special characters (:, ;, ,, =) and the digits. The nonterminals, which are not defined in the above syntax, are described below.

\( \text{(title)} \) : any title

\( \text{(directives to PLP)} \) : list (possibly empty) of directives to the PLP processor

\( \text{(attribute)} \) : any relevant attribute

\( \text{(identifier)} \) : as in standard FORTRAN but no length limit

\( \text{(subscripted identifier)} \) : as in standard FORTRAN but no length limit

\( \text{(number)} \) : as in standard FORTRAN

\( \text{(constant)} \) : integer

\( \text{(expression)} \) : as in FORTRAN

\( \text{(verb)} \) : any verb in the English language

\( \text{(comment)} \) : any group of English language words which are not verbs or nouns