DESIGN AND IMPLEMENTATION OF REX/3000: A NATURAL LANGUAGE REPORT WRITER

LANCE CARNES
GENTRY, INC
KENSINGTON, CALIFORNIA

This paper presents a natural language report writer, REX/3000 (Report EXpediter), designed for the casual user as well as the experienced programmer. The language design and compiler implementation are discussed from the point of view of meeting the needs of the two user classes. For the casual user, the language is natural and English-like. For the programmer, the language is a mixture of natural and procedural constructs. The PASCAL-based compiler has been implemented to serve each user class appropriately.

INTRODUCTION.

REX was developed to facilitate writing reports from IMAGE databases and MPE files. The users will range from the casual (infrequent) programmer attempting to formulate reports to the experienced programmer with more complex goals.

The time allotted for the design and implementation was eight man-months. The project consisted of the following:

1) Design the syntax of REX, i.e. the form of the language.

2) Design the semantics of REX, i.e. the function of the compiled programs, to properly execute for both the casual user and the programmer.

3) Write a compiler for REX.

This seemingly tight schedule was supported by several resources:

1) The author of a report writer similar to the required package (though unfortunately written in IBM 360 assembler language [1]) was available for consultation;

2) A PASCAL compiler had been transported to the HP by a user and graciously contributed to the user's library [2] ;

3) A copy of the production PASCAL-P compiler source (written in PASCAL) had also been contributed to the users library.

This paper will treat these topics, discussing the design considerations and the solutions implemented.

SYNTAX.

The careful design of the syntax of a language is important. From the compiler writer's point of view, the syntax should be simple and have as few rules and exceptions as possible. From the language user's point of view, especially the casual user, the syntax should be natural and English-like as well as consistent and unambiguous.

The syntax of a language is the way the elements of the language are put together to form statements. For example, in English many statements have the syntax

```
    subject  verb  object .
```

In computer languages the syntax of assignment statements is commonly:

```
    variable  =  number
 or
    variable  =  number + number
 or, in the general case,
    variable  =  expression
```

The syntax of REX lies somewhere between a typical programming language and a natural language. The total logical precision of a programming language is not wanted since the casual user is not trained in the use of these languages. Nor do we want the full complicated syntax of English since the language must be translated by a computer program (the compiler).

The syntax of REX is modeled after the syntax of many of the currently used "natural language" report writers: SYNTAX II [1], QUERY [3], NATURAL [4], RII [5], and RAMIS II [6]. The intent of all the natural language report writers is to avoid the "procedural" or programming details of programming languages and to allow the user to formulate English-like specifications which can be interpreted by a compiler.

Consider Example 1, a REX program which reads and prints selected values from a database called PAYROLL.

```
1     << LIST EMPLOYEES IN SALES DEPARTMENT FROM MARIN COUNTY >>
2     database PAYROLL password "READER" access 8
3       dataset EMPLOYEES
4     title; print "SALES PERSONNEL FROM MARIN COUNTY"; end
5     get EMPLOYEES with DEPT = "SALES"
6       select ZIP isbetween 94100,94199
7       print EMPL-NAME, EMPL-ADDR, EMPL-NUM
8     end.
```

Example 1. A complete REX program.
(Words in lowercase are keywords - in actual practice upper and lower case letters are treated the same.)

This is a complete REX program and will produce a listing of the desired values. (Its actual function will be discussed in the next section, "Semantics". See Appendix B for sample programs with printed output.)

The point here is that REX has an English-like syntax which serves the casual user. Notice that there are no procedural steps (i.e. there is no explicit opening or closing of the database, no explicit test for end-of-chain, etc). In a sense, the syntax closely resembles the specification of the program.

For the experienced programmer, who must deal with more complex programming requests, there are more complete constructs available. However, these constructs are not always "natural language" in nature and would not be used commonly by the casual user. The constructs are available for completeness and are not in ordinary demand by the non-programmer.

For example, suppose the user wishes to read an MPE file containing employee numbers, look up each employee number in the PAYROLL database, and write a new file with employee name, address and number.

```
1     << READ EMPLOYEE #, LOOK UP # IN PAYROLL DATABASE,
2     << WRITE NAME, ADDRESS, EMPLOYEE #

3     file EMP#FILE = "EMPLNO.PUB.ACCOUNTS" f10  << INPUT FILE >>
4       EMP# 1-4 n7  << 4-BYTE FIELD, 7-DIGIT INTEGER >>

5     database PAYROLL password "READER" access 8
6       dataset EMPLOYEES

7     file EMP-NAME-FILE = "EMPLNAMS.PUB" f72 << OUTPUT FILE >>
8       E-NAME  1-20 a20   << 20-BYTE ALPHA FIELD >>
9       E-ADDR 21-50 a30
10      E-NUM  51-54 n7    << 7-DIGIT INTEGER >>

11    progvar REC-CTR   n8 << 8-DIGIT COUNTER, RECORDS READ >>
12            NOT-FOUND n8 << 8-DIGIT COUNTER, INVALID EMP# >>

13    begin  << MAIN PROGRAM >>

14    read EMP#FILE  << READ NEXT RECORD FROM FILE >>

15      REC-CTR = REC-CTR + 1  << COUNT INPUT RECORDS >>

16      get first EMPLOYEES with EMPL-NUM = EMP#  << LOOK UP >> &
17        at end begin
18                  NOT-FOUND = NOT-FOUND + 1 << LOOK UP FAILED >>
19                  print "EMPL # " EMP# " NOT IN PAYROLL"
20                end
21      E-NAME = EMPL-NAME   << MOVE VALUES TO OUTPUT RECORD >>
22      E-ADDR = EMPL-ADDR
23      E-NUM  = EMPL-NUM
24      write EMP-NAME-FILE    << WRITE OUTPUT RECORD >>
25    loop  << DO NEXT read EMP#FILE >>

26    << ALL RECORDS PROCESSED, PRINT TOTALS >>
27    if NOT-FOUND <> 0 then  &
28      print "TOTAL EMPL #'S NOT FOUND = " NOT-FOUND  &
29    else print "ALL EMPL #'S FOUND
30    print "TOTAL RECORDS PROCESSED = " REC-CTR
31    end.
```

Example 2. A complex program.

This example illustrates many of the constructs available to the programmer or the adventuresome casual user. See Appendix A for a summary of the REX language.

SEMANTICS.

The syntax of a language provides a framework of correctly formatted statements which may then be converted to meaningful, functioning programs. The conversion of statements into functional programs is called the semantics of the programming language. The semantics for REX are such that the constructs used by the casual user require a great deal of implicit function, while the procedural constructs for the experienced programmer are explicit.

For example, in English the request

"Look at your watch and recite the exact hour and minute."

may also be stated as

"Tell me the time."

The first form is correct English syntax and there is no difficulty understanding what is desired, although this is not the usual way to ask the time. The second form has the same meaning and can be understood with slightly more effort; i.e. you must subconsciously recall that you need to look at your watch, and recite the hour and minute. The first form is procedural and explicit; the second form is natural and requires a great deal of implicit function.

The semantics of REX are designed to cover both cases - the casual user can express a natural request while the programmer can code the same function procedurally. To illustrate, recall Example 1. Nowhere was it indicated explicitly that the dataset was to be read using a chained read on the DEPT search item, looping back to the "get EMPLOYEES" statement after each entry had been printed. The explicit statement of this program is shown in Example 3.

```
1     database PAYROLL
2       dataset EMPLOYEES
3     title; print "SALES PERSONNEL FROM MARIN COUNTY"; end
4     find EMPLOYEES with DEPT = "SALES"  << FIND CHAIN HEAD >>
5     get next EMPLOYEES at end stop  << NEXT ENTRY ON CHAIN >>
6       select ZIP isbetween 94100,94199
7       print EMPL-NAME, EMPL-ADDR, EMPL-NUM
8     loop    << END OF get LOOP >>
9     end.
```

Example 3. Explicit statement of Example 1.

The program in Example 3 will produce the exact same result as the program in Example 1. Details are described which were filled in by the compiler in Example 1. The clause "at end stop" indicates that processing is to cease at the end-of-chain. The "loop" statement indicates that the processing loop ends precisely here, so that statements may be written following the loop which are not executed until the loop is terminated (see Example 2).

In all aspects of REX, the functions which are most used by the casual user have a "natural language" or non-procedural default. Another example of this aspect of REX is the report block. The following program produces a result similar to the program in Example 1 - except here the output is sorted by employee name, and a report title and column headings are provided:

```
1     database PAYROLL
2       dataset EMPLOYEES
3     report
4       title; print "SALES PERSONNEL FROM MARIN COUNTY"; end
5       select ZIP isbetween 94100,94199
6       list EMPL-NAME, EMPL-ADDR, EMPL-NUM  &
7         sorted by EMPL-NAME
8     end
9     get EMPLOYEES with DEPT = "SALES"
10    end.
```

Example 4. A sorted report.

Here we have specified the contents of a report (report...end) and the method of obtaining items for the report (get EMPLOYEES ...). What is left out of the specification (but included in the implicit function) is the linkage between the "report" and the "get", and the indication of when to sort the data and print the report.

Consider the explicit version of this same program:

```
1     database PAYROLL
2       dataset EMPLOYEES
3     report
4       title; PRINT "SALES PERSONNEL FROM MARIN COUNTY"; end
4       select ZIP isbetween 94100,94199
5       list EMPL-NAME, EMPL-ADDR, EMPL-NUM  &
6         sorted by EMPL-NAME
7     end
8     get EMPLOYEES with DEPT = "SALES"
9       call MARIN-RES   << UPDATE REPORT DATA >>
10    loop
11    print MARIN-RES    << SORT AND PRINT THE REPORT >>
12    end.
```

Example 5. Explicit statement of Example 4.

Notice that the report block now has an identifier "MARIN-RES" which allows it to be "call"-ed, much as a procedure block. The statement "print MARIN-RES" explicitly specifies the exact point at which the report is output.

Thus the semantics of REX are intended to appeal to both the casual and programmer user. The programmer is not saddled with a restrictive language set and may tackle more complicated programs; the casual user may ignore the procedural aspect of REX or may, with some assistance or experimentation, improve his results by venturing into the procedural constructs of the language. Acceptable results may be achieved from either level of expertise.

THE COMPILER.

One of the original goals underlying the implementation of REX was the low-cost development of a compiler which could be transported to other machines. Most of this goal was achieved. The compiler was developed at low cost (eight man-months) and is transportable (written in PASCAL). However, since the object code produced by the compiler is SPL, the package will run only on the HP at present.

A compiler is a program which processes source text in a pre-defined language to produce some form of object code. The syntax and semantics of the language to be compiled can greatly affect the complexity, and therefore the cost, of developing and maintaining a compiler. When aiming at low-cost development, it is important to keep it as simple as possible without sacrificing the utility of the language.

REX is not an easy language to compile. However, the task was reduced because a production compiler was used as a model (the PASCAL-P compiler written in PASCAL). Also, some language design decisions eased the translation effort without reducing the capability of the language.

As an example, the element <expression> was used everywhere that it made sense. This allows any production in the language which uses <expression> to share a common compiling procedure. Of course, if the statement needs an expression of a certain type, say logical in the case of "if <expression>", the compiler will flag an expression of any other type as an error.

In contrast, the language SYNTAX II [1] has several expression types which are compiled distinctly, depending on the expected type. In the case of the assignment statement SYNTAX II has several different syntactic forms depending on the destination data type.

```
1     A EQ B                              A,B CHARACTER TYPE
2     SET A = B + 1 or A = B + 1          A,B NUMERIC TYPE
3     RECODE A TO B                       A,B ANY SAME TYPE
4     T TEST A < B                          T LOGICAL TYPE
```

Figure 1. Sample assignment statements from SYNTAX II.

The result is that there are several additional compiling procedures that must be developed and maintained.

Using a construct wherever it makes sense is desirable from the compiler writer's point of view and also from the user's point of view [7] [8]. When the user must formulate his idea differently for each data type used, the language designer has burdened the user with additional effort. For example, in the SYNTAX II "SET" statement it is legal to write

```
SET A = B * 100
```

while in the "TEST" statement it is not legal to write

```
T TEST A < (B * 100)
```

even though A and (B * 100) are of the same type. The user must remember which operators can be used in which contexts. This requires an additional level of expertise that we wish to avoid. However, if there is only one set of rules for the formation of expressions, the user need not recall a lot of exceptions to the rules.

The REX compiler was implemented in PASCAL, a high-level recursive language [9]. The production PASCAL-P compiler source code [10] was used as the basis and as a model for writing the REX compiler. SPL was chosen for the compiler's object code primarily because it is the language closest to the machine level. SPL compiles to produce the most resource-efficient run-time programs. It runs with the least memory usage and fastest execution time compared to other compiled object code (FORTRAN or COBOL). Output in relocatable code was not chosen since there is no documentation currently available from HP on its format.

The appearance of the compiler from the user's standpoint is much like any other HP compiler. There are UDC's for invoking the compiler which resemble the commands for invoking the SPL, COBOL or FORTRAN compilers. Error messages are clear and point to the place in the text where the error occurred:

```
select ZIP isbetwee 94100,94199
                    ^
****  UNKNOWN SYMBOL     (GCPERR 72)
```

Should the user want more information, he can refer to the user's manual under "GCPERR 72".

CONCLUSIONS.

REX has many powerful features which will allow the user to generate accurate, timely, and complete reports with minimal program development effort. Results with comparable packages show a decrease in development time by as much as a factor of ten [6], compared with developing the same application in COBOL or FORTRAN.

The casual user can benefit greatly from bypassing the data processing department and writing his own ad hoc reports. The programmer can increase his productivity by reducing the development effort required to produce routine reports and queries.

The initial response from the first users has been enthusiastic. Estimates indicate program development time has been reduced overall by 80%, compared to producing the same program in a typical procedural language (COBOL or FORTRAN). Maintenance and revision of production program source code has been reduced by 80% also. Programmers tend to write more reliable code indicated by the fact that most bugs are problem-based rather than implementation-based.

The casual users (users who are not programmers by profession but who have a good grasp of basic data processing concepts) have been surprisingly successful using REX to generate reports from databases. Using existing programs as a reference, they have been able to turn out complex reports within a few hours time. Typically very little training time has been required to get the casual user to the point where he or she can produce useful results.

Future extensions include full IMAGE access (DBPUT, DBDELETE, DBUPDATE), and full KSAM access.

REX will be marketed by

GENTRY INC.
609 Kearney Street
Kensington, California 94530
U.S.A.
(415) 527-4451

The first release of REX will be available January 1980.

ACKNOWLEDGEMENTS.

My thanks to John Fitz, Richard Gentry, and Ron Frankel for many valuable discussions during the language definition phase of the project; and my special thanks to Grace Gentry, who provided me the opportunity to do this project and who assisted in the preparation of this paper.

REFERENCES.

1. John Fitz, SYNTAX II USER'S GUIDE. University of California, 1977.

2. Bob Fraley, PASCALP, HP User's library.

3. QUERY. Hewlett-Packard Product # 32216A.

4. NATURAL Users Manual, Software AG, 1978.

5. RII, Computer Sciences Corp.

6. RAMIS II, Mathematica Corp.

7. G. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.

8. W.M. McKeeman, "Programming Language Design" from Compiler Construction, F.L.Bauer,ed., Springer-Verlag, 1976.

9. K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, 1974.

10. Urs Ammann, PASCAL P4 compiler source code, Zurich, 1976.

Appendix A: Summary of REX/3000 Language.

```
REPORT                                          Sorted report.
  LIST ...
    SORTED BY ...
END

TABLE                                           Cross-tabulation.
  ROW ...
  COLUMN ...
END

PROCEDURE name                                  Procedure.
  ...
END

TITLE name                                      Report title.
  ...
END

DATABASE basename                               IMAGE declaration.
  DATASET setname

FILE filename                                   MPE file declaration.
  field1 ...
  field2 ...
  ...

PROGVAR var1 ...                                Variable declaration.
        var2 ...
        ...

                                                Database access.

GET   setname [WITH searchitem = expression]
FIND          [AT END statement]

READ  filename [AT END statement]

WRITE filename                                  File access.

PRINT expression [, expression, ...]            Unformatted print.

                                                Sort/Merge.
SORT filename [INTO filename]
         BY key1, key2, ...

MERGE filename, filename [, filename, ...]
         INTO filename
         BY key1, key2, ...
```

```
IF expression THEN statement              Control statements.
             [ELSE statement]

GOTO label

CALL blockname

REPEAT statement [; statement; ...] UNTIL expression

WHILE expression DO statement

FOR ident = expression TO expression DO statement


BEGIN [statement; statement; ...] END  Compound statement.

ident = expression                        Assignment statement.


+ - * / DIV MOD                           Arithmetic operators.

GT GE LT LE EQ NE ISBETWEEN CONTAINS   Relational operators.
>  >= <  <= =  <> IB        <*>

AND OR NOT                                Logical operators.

                                          Data types.

Nw     w <= 10          Integer
Fw.d   d <= w <= 15     Real
L                       Logical
Aw     w <= 256         Alpha
```

APPENDIX B: SAMPLE REX/3000 REPORTS.

The following are several reports generated by REX/3000 from MPE files and IMAGE databases.

Sample 1. Sorted report using MPE file.

```
FILE PARTS  F80
  PN 1-4 AS "PART NO"
   PD 5-14 AS "PART DESC"
   PL 15-17 AS "LOCATION"
   QTY 18-22 n5 AS "QUANTITY"
REPORT
TITLE; PRINT "WAREHOUSE PARTS SUMMARY"; END
   LIST PN, 5X, PL, 2X, PD, 2X, QTY        &
      SORTED BY PN, PL                     &
      SUMMARIZING QTY ON PN                &
      TOTALING "TOTAL", QTY
END
READ PARTS  << READ FILE RECORDS >>
END.
```

```
1785BOLT 1 X 1/4 101 2000
2142BRACKET      100  750
3122MANUAL #177  101  100        Contents of the
2142BRACKET      102  250        PARTS file.
2142BRACKET      101  100
1785BOLT 1 X 1/4 100 1000
```

The REX/3000 report:

WAREHOUSE PARTS SUMMARY

PART NO	LOCATION	PART DESC	QUANTITY
1785	100	BOLT 1 X 1/4	1000
	101	BOLT 1 X 1/4	2000
1785			3000
2142	100	BRACKET	750
	101	BRACKET	100
	102	BRACKET	250
2142			1100
3122	101	MANUAL #177	100
3122			100
TOTAL			3200

Sample 2. Sorted report with IMAGE/3000 database.

The following is an IMAGE/3000 schema for a database to hold the same information from Sample 1:

```
BEGIN DATABASE WAREHOUSE;

ITEMS:
 PART-NO,     X4;
 PART-DESC,   X10;
 PART-LOC,    X4;
 PART-QTY,    I2;

SETS:

 NAME: PARTS, DETAIL;
 ENTRY:
   PART-NO,
   PART-DESC,
   PART-LOC,
   PART-QTY;
 CAPACITY: 100;

...

END.
```

The following REX/3000 report specification will produce the same report as in Sample 1:

```
DATABASE WAREHOUSE
  DATASET PARTS

REPORT
  TITLE; PRINT "WAREHOUSE PARTS SUMMARY"; END
   LIST PART-NO, 5X, PART-LOC, 2X, PART-DESC, 2X, PART-QTY N5    &
      SORTED BY PART-NO, PART-LOC                                 &
      SUMMARIZING PART-QTY ON PART-NO                             &
      TOTALING "TOTAL", QTY
END
GET PARTS  << READ DATASET ENTRIES >>
END.
```

Sample 3. Cross-tabulation with IMAGE/3000 database.

This sample uses the same database as in Sample 2.

```
DATABASE WAREHOUSE
  DATASET PARTS

TABLE
  TITLE; PRINT "PARTS DISTRIBUTION"; END
  ROW PART-NO BINS("1785","2142","3122")
  COLUMN PART-LOC BINS(100,101,102) ACCUMULATE PART-QTY
         TOTAL-PARTS LABEL "TOTAL PARTS" ACCUMULATE PART-QTY
END
GET PARTS  << READ FROM DATABASE >>
END.
```

The following cross-tabulation will be produced:

PARTS DISTRIBUTION

	PART-LOC			
PART-NO	100	101	102	TOTAL PARTS
1785	1000	2000	0	3000
2142	750	100	250	1100
3122	0	100	0	100