V[illegible]7 NOS. 3 & 4 JULY/DECEMBER 1984

JOURNAL

INTEREX, THE INTERNATIONAL ASSOCIATION OF HEWLETT-PACKARD COMPUTER USERS

JOURNAL

JOURNAL

JOURNAL

JOURNAL

JOURNAL

JOURNAL

JOURNAL

Publications committee members

Chairman: **Dr. John R. Ray,** College of Education, Department of Curriculum & Instruction, University of Tennessee at Knoxville, Knoxville, Tennessee 37996-3400 USA.

Gary H. Johnson, Brown Data Processing, 9229 Ward Parkway, Kansas City, Missouri 64114 USA.

Ragnar Nordberg, Department of Clinical Chemistry, University of Gothenburg, Sahlgren's Hospital, S-41345 Gothenburg, Sweden.

Michael J. Modiz, Hayssen Manufacturing Company, Highway 42 North, Sheboygan, Wisconsin 53081 USA.

Marjorie K. Oughton, Supervisor of Data Processing, Alexandria City Public Schools, 3801 Braddock Road, Alexandria, Virginia 22302 USA.

Interex, 2570 El Camino Real West, Fourth Floor, Mountain View, California 94040, (415) 941-9960.

Editor Christine M. Dorffi
Technical Editor Dr. John R. Ray

Using DICTIONARY/3000 as a COBOL programmer's tool

Leon Leong

Information Networks Division
Hewlett-Packard
Cupertino, California
• USA

INTRODUCTION

Data dictionaries are a central repository for data resources on a computer system. A programmer or database administrator documents databases, files, record layouts, and data items, as well as the applications utilizing those data definitions in the data dictionary.

Data dictionaries can be more than just a documentation tool, however. The information found in a data dictionary can be used as the basis for a set of programmer productivity tools. One example of the type of tools that can be serviced by data dictionaries is the set of new DICTIONARY/3000 utilities recently released by Hewlett-Packard. These utilities aid in the generation of COBOL source code data declarations and in populating the dictionary with data definitions from VPLUS/3000 forms files.

COBOL DATA DECLARATION GENERATION

The new DICTIONARY/3000 utility, Dictcde, will generate COBOL data declarations into a COBOL copylib file. A programmer will specify the entity in the dictionary; Dictcde extracts the definition of the entity, formats the definition into a COBOL data declaration, and writes it to a copylib module that the user specifies. The COBOL statements and clauses generated are dependent on the entity, its dictionary type, the values of its attributes, as well as any relations it has with other dictionary entities.

Dictcde generates a WORKING-STORAGE SECTION level-number 1 record description for IMAGE/3000 data set entities. Every data item that is related to the data set in the dictionary will be generated as a level-number 5 field in the record description. Dictcde gives the programmer the option of also having the data item names as well as the data set name generated. This is useful for constructing the LIST parameter to pass to IMAGE/3000 intrinsics. Another feature is the specification of a prefix term for the fields in the record being generated; this facilitates coding without qualification. See Listing 1 for an example of the source generated for an IMAGE/3000 data set.

When the programmer specifies a dictionary database entity, Dictcde will generate COBOL source declarations for all data sets related to the database. The programmer has the option of putting each data set in a separate copylib module, and of skipping source code generation on selected data sets.

Dictcde generates a record description for VPLUS/3000 form entities similar to that of an IMAGE/3000 data set entity. However, instead of an option for generating data item names, a VPLUS/3000 field number table can be generated instead. If the programmer specifies a forms file entity, Dictcde will generate source declarations for each form related to the forms file.

For MPE and KSAM files, Dictcde gives the programmer the choice of generating declarations for the INPUT-OUTPUT SECTION in the ENVIRONMENT DIVISION, the FILE-SECTION of the DATA DIVISION, the WORKING-STORAGE SECTION of the DATA DIVISION, or a combination of all three. Each section can be generated into a different copylib module. Every file defined in the dictionary has a set of attributes associated with it; these attributes describe pieces of information about the file. Table 1 outlines the mapping between the dictionary file attributes and the COBOL clauses. In addition, if Dictcde finds the File entity related to a Location entity, it will use the Location entity attributes in the ASSIGN clause generated in the ENVIRONMENT DIVISION. See Listing 2 for an example of the source that is generated for an MPE file.

Dictcde generates Element entities as level-number 1 declarations. Nested element definitions start at level-number 5, and each further nesting level increments the level-number by 5. The programmer may specify a prefix for the nested levels to facilitate coding without qualification. The attribute values of the Dictionary Element entity is used to generate the WORKING-STORAGE SECTION clauses. Table 2 shows how the attributes are mapped into the COBOL clauses.

Other options of Dictcde include:

- source code generation of the VPLUS/3000 comarea,
- source code generation of the IMAGE/3000 standard parameters,
- commenting the declarations with creation date, modification date, and person responsible,

Table 1 **Dictionary file attribute mapping**

Dictionary file attribute	COBOL division	COBOL section	COBOL clause
FILE NAME	ENVIRONMENT	INPUT-OUTPUT	ASSIGN
RECORD FORMAT	DATA	FILE	RECORDING MODE
MINIMUM RECORD SIZE MAXIMUM RECORD SIZE	DATA	FILE	RECORD CONTAINS
UNIT MINIMUM BLOCKING MAXIMUM BLOCKING	DATA	FILE	BLOCK CONTAINS
RECORDING MODE	ENVIRONMENT	INPUT-OUTPUT	ASSIGN
DATA STORAGE TYPE	DATA	FILE	CODE SET IS
CCTL	ENVIRONMENT	INPUT-OUTPUT	ASSIGN
DEVICE	ENVIRONMENT	INPUT-OUTPUT	ASSIGN
DEVICE CLASS	ENVIRONMENT	INPUT-OUTPUT	ASSIGN

- source code generation using the alias names in the dictionary,
- generation of explicit and implicit redefines based on the definition found in the dictionary.

Dictcde will log a programmer's interaction with Dictcde to a file. The programmer can redirect Dictcde's input device from the terminal to the log file. The feature is useful for regenerating the data definitions whenever they change in the dictionary, with a minimum of effort.

LOADING VPLUS/3000 DEFINITIONS

Another utility recently released with DICTIONARY/3000, Dictvpd, automates the entry of VPLUS/3000 forms definitions. Dictvpd takes definitions of forms and fields in a VPLUS/3000 forms file and loads them into the dictionary. Dictvpd operates in a manner similar to Dictdbd, the utility that loads the dictionary with the definition of an IMAGE/3000 database from a root file. Dictvpd gives the programmer the option of loading all the forms or selectively choosing forms. Dictvpd will load the form and field definitions, as well as the relationships between the two.

PROGRAMMER PRODUCTIVITY

Some of the tasks in developing an application involve the specification of the database and the forms the end user will interface with. The programmer then codes the programming language declarations for accessing the database and forms file. The coding portion of these tasks can take days or weeks on an extensive application involving hundreds of forms and a large database definition.

Table 2 **Element entity attribute mapping into COBOL clauses**

Dictionary element attributes	WORKING-STORAGE SECTION clauses
TYPE	USAGE IS
SIGN POSITION	SIGN IS
SIZE	PICTURE IS (on binary types)
DECIMAL	PICTURE IS
STORAGE LENGTH	PICTURE IS (on alpha types)
COUNT	OCCURS
EDIT MASK	PICTURE IS
BLANK WHEN ZERO	BLANK WHEN ZERO
RIGHT JUSTIFY	JUSTIFIED RIGHT
SYNCHRO-NIZED	SYNCHRONIZED

A programmer can use the data dictionary and a combination of the utilities mentioned above to increase the productivity on coding tasks. For example, a programmer can design forms with Formspec, use Dictvpd to load the information into DICTIONARY/3000, and then use Dictcde to generate the COBOL source declarations for those forms. These new tools can reduce the coding time for declarations to just minutes.

OTHER BENEFITS

Names for data entities can be standardized in the source code more easily, especially across applications and programmers, since the data declarations are being generated from one central location. The dictionary will also provide a good documentation tool and will aid in the analysis of enhancements and maintenance of applications. Other HP products that run off of DICTIONARY/3000 can leverage off the same definitions, such as INFORM/3000 and REPORT/3000. In addition, Dictcde can now be run through HP TOOLSET/3000 to provide programmers with an integrated development environment.

SUMMARY

As data dictionaries become more central in the management of information on a computer system, more and more uses of the dictionary are surfacing. One of these uses is as a programmer productivity tool. DICTIONARY/3000 allows programmers to concentrate more of their time on the design and specification process, using existing data definitions where applicable, and less time on the coding aspects of application development.

Listing 1

```
:run dictcde.pub.sys

DICTIONARY/3000 DICTCDE          HP32244A.02.00 - © Hewlett-
                                 Packard Co. 1983

Type ? at any prompt for help.

Dictionary password>

Copylib file name> lib

File, Element, Parameters, Options, or EXit (F/E/P/O/EX)> f

File name> ordmgt

File ORDMGT is an IMAGE data base

Define all data sets in one module (N/Y)> y

Copylib module for ORDMGT> ordmgt
000100
000200 01  CUSTOMER-DATA.
000300     05 ACCOUNT.
000400        10 FIRST-SALE      PIC  S9(10).
000500     05 FIRST-NAME         PIC  X(18).
000600     05 FIRST-INITIAL      PIC  X(2).
000700     05 LAST-NAME          PIC  X(20).
000800     05 STR-ADDRESS        PIC  X(22).
000900     05 CITYNAME           PIC  X(14).
001000     05 STATE              PIC  X(2).
001100     05 ZIP                PIC  X(10).
001200     05 CREDIT             PIC  X(2).
001300     05 FIRST-SALE         PIC  S9(10).
001400
001500 01  DS-CUSTOMER       PIC  X(9) VALUE "CUSTOMER".
001600
001700
001800 01  PRODUCT-DATA.
001900     05 PROD-NO            PIC  X(8).
002000     05 DESCRIPTION        PIC  X(30).
002100
002200 01  DS-PRODUCT      PIC  X(8) VALUE "PRODUCT".
002300
002400
002500 01  DATE-MASTER-DATA.
002600     05 DELIV-DATE         PIC X(6).
002700
002800 01  DS-DATE-MASTER        PIC  X(12) VALUE "DATE-
002900                                             MASTER".
003000
003100 01  SALES-DATA.
003200     05 ACCOUNT.
003300        10 FIRST-SALE         PIC S9(10).
003400     05 PROD-NO            PIC X(8).
003500     05 TOTAL              PIC S9(11) COMP-3.
003600     05 PURCH-DATE         PIC X(6).
003700     05 DELIV-DATE         PIC X(6).
003800     05 PURCH-NO           PIC X(6).
003900
004000 01  DS-SALES      PIC  X(6)  VALUE "SALES".
004100
004200
004300 01  INVENTORY-DATA.
004400     05 PROD-NO          ·  PIC X(8).
004500     05 BACKORDERFLG       PIC X(2).
004600     05 UNIT-COST          PIC S9(11) COMP-3.
004700     05 SHIP-DATE          PIC X(6).
004800
004900 01  DS-INVENTORY PIC X(10) VALUE "INVENTORY".
005000
005100
005200 01  DB-ORDMGT             PIC X(20) VALUE
                                   "ORDMGT.PUB.DSUSER".
005300 01  ORDMGT-PWD            PIC X(10) VALUE ";          ".
005400
005500 01  DI-ACCOUNT            PIC X(8) VALUE "ACCOUNT".
005600 01  DI-PROD-NO            PIC X(8) VALUE "PROD-NO".
005700 01  DI-DELIV-DATE         PIC X(11) VALUE "DELIV-
                                                     DATE".
005800 01  DI-PURCH-DATE         PIC X(11) VALUE "PURCH-
                                                     DATE".
005900 01  DI-SHIP-DATE          PIC X(10) VALUE "SHIP-DATE".

File, Element, Parameters, Options, or EXit (F/E/P/O/EX)> ex

Save file DICTLOG to keep log of responses.

END OF PROGRAM
:run pscreen.pub.sys
```

Listing 2

```
:run dictcde.pub.sys

DICTIONARY/3000 DICTCDE       HP 32244A.02.00 - © Hewlett-
                              Packard Co. 1983
Type ? at any prompt for help.
Dictionary password>
Copylib file name> lib
File, Element, Parameters, Options, or EXit (F/E/P/O/EX)> f
File name> warranty
File WARRANTY is an MPE file

Copylib module for SELECT statement> warrant1

Copylib module for FILE SECTION entry> warrant2

Define WARRANTY in FILE SECTION as FD or SD file (F/S)> f

Copylib module for WORKING-STORAGE record> warrant3

Prefix for data items in WARRANTY> warrant-
000100
000200      SELECT WARRANTY
000300          ASSIGN "WARRANTY"
000400          ORGANIZATION IS SEQUENTIAL.
000100
000200  FD WARRANTY
000300      RECORDING MODE IS F.
000400  01  WARRANTY--REC               PIC X(110).
000100
000200  01  WARRANTY-DATA.
000300      05 WARRANT-ACCOUNT          PIC X(10).
000400      05 WARRANT-PROD-NO          PIC X(8).
000500      05 WARRANT-DESCRIPTION      PIC X(30).
000600      05 WARRANT-TOTAL            PIC S9(11) COMP-3.
000700      05 WARRANT-SHIP-DATE        PIC X(6).
000800      05 WARRANT-WRNTY-NO         PIC X(6).
000900      05 WARRANT-OWNER            PIC X(40).
001000      05 WARRANT-QTY              PIC S9(7) COMP-3.
001100

File, Element, Parameters, Options, or EXit (F/E/P/O/EX)> ex

Save file DICTLOG to keep log of responses.

END OF PROGRAM
:run pscreen.pub.sys

**SCREEN CONTENTS @: SUN, JAN 29, 1984, 2:55 PM (53724)
PSCREEN (B00.03)
```

LISP Tutorial: some basic functions

David J. Greer

Robelle Consulting Ltd.
Langley, British Columbia
Canada

The programmer's world is undergoing massive changes. COBOL and FORTRAN still hold sway in many shops, but so-called fourth generation languages like QUIZ, PROTOS, and TRANSACT are replacing them for many purposes. If a programmer wants to stay employable, he must discover what kinds of programming knowledge will be needed in the future. It is possible that a radically different language, such as LISP, will be the common tongue of programmers in the next two decades.

To illustrate the different styles of programming languages, we have written this tutorial on LISP. LISP stands for *List Processing*, and it is as unlike COBOL as it can be while remaining in the same category, programming languages.

As part of the Pascal/Robelle compiler and program library, we include a complete interpreter for the language LISP, written in Pascal. This interpreter is rich in built-in functions and power, but limited in data space. It is suitable for learning LISP, but not for writing large artificial-intelligence programs. The examples in this paper are based on our LISP interpreter. Although the same functions should exist in most LISP implementations, the specific syntax may require some changes; LISP is *not* standardized. (There is a good textbook on LISP, which inspired some of the ideas below: *LISP*, by Winston and Horn, Addison Wesley, 1982.)

LISP is like a calculator: when you type into it, LISP looks for functions to perform now. When you want to define new functions that will be evaluated *later*, you must switch modes, like switching into PROGRAM mode on your programmable calculator. The difference between LISP and a calculator is that the "something" that LISP looks to evaluate is a *symbolic expression* (s-expression). An s-expression may include numbers or strings, but gains most of its power from the fact that it can consist of words. After LISP has evaluated your s-expression, it prints the result. For example:

```
>10
10
>
```

In this example, the s-expression was the number 10. The greater-than symbol (>) is the prompt character.

Here is an s-expression with an error:

```
>A
    Error: unbound atom
```

The result of evaluating a number is the number itself. The result of evaluating an identifier is the value associated with the identifier. In this example, the identifier A was created the first time that LISP saw it. When this happens, LISP assigns an "undefined" value to A. The evaluation of A caused the error "unbound atom" because A had no value. We can easily assign a value to A:

```
>(SET 'A 10)
```

The quote mark is required here. It tells LISP "do *not* evaluate the s-expression that follows." If the quote mark were left off, we would get the "unbound atom" error. We can obtain the value of A by the following:

```
>A
10
```

LISP uses post-fix notation to do arithmetic. To add two numbers together, we use the predefined function PLUS. For example:

```
>(PLUS 10 20)
30
```

In post-fix, the arguments are after the function name, as in a function call of FORTRAN or Pascal. This contrasts with arithmetic expressions in FORTRAN and Pascal, which use algebraic notation: 10 + 20. HP calculators, on the other hand, use pre-fix notation, with arguments before the function.

We had to surround our LISP example with parentheses. If we had just entered PLUS 10 20, LISP would try to evaluate the identifier PLUS. This would result in the *value* of PLUS, but we wanted the value of evaluating the function PLUS on the arguments 10 20. When LISP sees a parenthesis, it creates a list. When the list is evaluated, PLUS is invoked as a function with two parameters.

We have been entering all of our examples in uppercase letters only. Input to LISP can be in lowercase or uppercase letters; LISP always shifts its input to uppercase.

Before writing our first LISP function, we need two more predefined functions. The function QUOTIENT returns the quotient of dividing the first number by the second. The function DIFFERENCE returns the difference of two numbers. For example:

```
>(DIFFERENCE 10 5)
5
>(DIFFERENCE 5 10)
—5
>(SET 'A 10)
10
>(DIFFERENCE A 5)
5
>(QUOTIENT 10 5)
2
>(QUOTIENT A 2)
5
```

In the example (DIFFERENCE A 5), we did *not* put a quote before A because we wanted the value of A, which is 10.

We will now define a LISP function to convert Fahrenheit temperatures to Celsius. To create a function, we use the predefined function DE, called DEFINE in some LISP dialects:

```
>(DE NAME '(PARAMETERS)
          (FUNCTION CODE)  )
```

There is *no* quote before the name of the function, because DE always takes its first argument unevaluated. Here is our F-TO-C function:

```
>(DE F-TO-C '(TEMP)
          '(QUOTIENT (DIFFERENCE TEMP
           32) 1.8))
F-TO-C
```

The result of the DE function is a new function name. We have only defined the function. To invoke it, we use:

```
>(F-TO-C 75)
23.88889
>(F-TO-C 32)
0
```

Success! We have written our first LISP program.

Our function does convert temperatures from Fahrenheit to Celsius, and the answer 23.88889 is accurate within the limits of the HP 3000 and Pascal/Robelle. But 23.88889 is not a friendly answer. We need a function that will round floating-point numbers to their nearest whole digit.

How do we build this function? When using LISP, we construct one small piece of the solution at a time. First, we need to obtain the fractional part of the real number. We will use the predefined function FIX, which returns the whole portion of a real number.

```
>(FIX 12.5)
12
>(FIX 12.4)
12
>(DIFFERENCE 12.5 (FIX 12.5))
0.5
>(DIFFERENCE 12.0 (FIX 12.0))
0
```

Now we need to be able to make a decision. If the fractional part is less than 0.5 we just return the whole part, otherwise we return the whole part plus one. Fortunately, LISP contains a method for making decisions: the predefined function COND. The general form of the COND function is:

```
>(COND (<CONDITION-1> <STATEMENT-1>)
       (<CONDITION-2> <STATEMENT-2>)
       (<CONDITION-N> <STATEMENT-N>))
```

A condition in LISP is any function that returns the value T or the value NIL. The value NIL is a value used many times in LISP. It can be represented by the empty list ( ).

We need to know if the fraction is less than 0.5. There is a predefined function LESSP that returns T or NIL, depending upon whether the value of the first argument is less than the second. For example:

```
>(LESSP 0.4 0.5)
T
>(LESSP 0.5 0.5)
NIL
>(LESSP 0.6 0.5)
NIL
>(LESSP (DIFFERENCE 12.4 (FIX 12.4)) 0.5)
T
```

The last example shows that the fractional part of 12.4 is 0.4 and that 0.4 is less than 0.5.

Finally, we put these pieces together to create a function called ROUND. Its definition would be:

```
>(DE ROUND '(X)
          '(COND ((LESSP (DIFFERENCE X
           (FIX X)) 0.5) (FIX X))
           (T (FIX (PLUS 1 X)))))
ROUND
```

How does this function work? It obtains the fractional part of the value of X—(DIFFERENCE X (FIX X)). Then, it checks to see if this value is less than 0.5. If the fractional part of X is less than 0.5, the whole part of X is returned (FIX X). The T in the COND function means, if all of the other conditions are *not* true do this statement. Round adds 1 to the value of X and returns the whole portion of the result. To try the round function, do the following:

```
>(ROUND 12)
12
```

```
>(ROUND 12.4)
12
>(ROUND 12.5)
13
>(ROUND 12.999)
13
>(ROUND 12.49999)
12
```

We built the ROUND function so that we could use it in our F-TO-C function. One way would be to do the following:

```
>(ROUND (F-TO-C 75))
24
```

Even better would be to always round the result of the F-TO-C function. The new F-TO-C function becomes:

```
>(DE NEW-F-TO-C '(TEMP)
                '(ROUND (F-TO-C TEMP)))
NEW-F-TO-C
>(NEW-F-TO-C 75)
24
>(NEW-F-TO-C 65)
18
>(NEW-F-TO-C 32)
0
```

This example demonstrates the expressive power of LISP, but a more efficient version of F-TO-C could be written in SPL, Pascal, or even COBOL. The power of LISP is not in its number-processing ability, but in its ability to manipulate symbols.

As an example of how LISP handles symbols, we will build some routines to maintain a list of telephone numbers. Our general scheme is to keep pairs of names and telephone numbers. To start, we need to learn more about the basic elements of LISP.

We create lists by surrounding them with parentheses. For example, we can create some lists using the letters of the alphabet:

```
>'(A B)
(A B)
>'(C D)
(C D)
>'((A B) (C D))
((A B) (C D))
>'(A B C D)
(A B C D)
```

There are two basic elements to LISP s-expressions: lists and atoms. A list is specified with a left or open parenthesis and an atom can be an identifier, a number, or a string. Our first example is a list—specified with a left parentheses (—of the two atoms A and B. The second example is another list with the two atoms C and D.

The quote mark shows up again. We use it to stop LISP from trying to evaluate the letter A. When LISP reads an s-expression it *always* tries to evaluate the expression. If LISP reads a list, that is, (A B), LISP will assume that if the first item in the list is an atom it should be executed as a function. We use the quote mark (which is really a function) to stop LISP from trying to use the atom A as a function.

It is important that the last two examples be understood. The first example shows two lists: the first is (A B), and the second is (C D). The last example is *one* list, (A B C D). For our phone list, we want a list of lists ((A B) (C D)).

You should be able to start your phone list now. We will choose the name PHONELIST to hold the list of names and phone numbers.

```
>(SETQ PHONELIST '((GREEN 5335500)
                   (GREER 7347589)
                   (OVERTON 6883993)))
((GREEN 5335500) (GREER 7347589) (OVERTON
                                  6883993))
```

We have not specified the dash within the phone number. Because the dash represents the negative sign, it is difficult to use within LISP without special programming. For our examples, we will treat phone numbers as one large number (that is, a numeric atom to LISP).

We have introduced a new LISP function SETQ. This function is identical to SET, but the first argument to SETQ is *not* evaluated. This means that we do not have to specify a quote mark before the name that receives the new value. For example, we will set the atom A to the value 20 using both SET and SETQ:

```
>(SET 'A 20)
20
>(SETQ A 20)
20
```

In the second case, we did not need the quote mark before A. From now on, we will use SETQ instead of SET.

Our phone list is small. To find the phone number of a specific person, we just print the entire list and look up the person's name by scanning the list. It would be more efficient if we had a function that would return the phone number for a specific name.

The list that we have built (a list of pairs) is an association list. LISP has a function ASSOC to search an association list.

```
>(ASSOC 'A '((A B) (C D)))
(A B)
>(SETQ LIST1 '((A B) (C D)))
((A B) (C D))
>(ASSOC 'A LIST1)
```

```
(A B)
>(ASSOC 'GREER PHONELIST)
(GREER 7347589)
>(ASSOC 'OVERTON PHONELIST)
(OVERTON 6883993)
>(ASSOC 'SMITH PHONELIST)
NIL
```

The final example shows what happens if a name does not exist on the phone list. While NIL means something to LISP users, it is not friendly to general users. The name ASSOC will be difficult for users to remember when they want to look up a phone number. Therefore, we will write a user-friendly routine to find telephone numbers.

Before starting, we need a method for returning an error message. In LISP, we enclose strings of words within double quote marks ("). For example,

```
>"This is a string"
"This is a string"
>"No such person"
"No such person"
```

LISP treats a string as an atom. The words within the string are *not* examined by LISP. A string can be returned as the value of a function by specifying the string. Unfortunately, LISP prints the string with the quote marks.

Now we can write our *find telephone number* routine.

```
>(DE GETPHONE '(NAME)
                     '(COND ((NULL (ASSOC NAME
                                    PHONELIST))
                               "No such person")
                            (T (ASSOC NAME
                                    PHONELIST))))
GETPHONE
>(GETPHONE 'GREEN)
(GREEN 5335500)
>(GETPHONE 'OVERTON)
(OVERTON 6883993)
>(GETPHONE 'SMITH)
"No such person"
```

How does this function work? We call GETPHONE with the name that we are interested in. We check if the associated name on the phonelist exists. If it does not, we return a message. Otherwise, we return the information associated with the name.

Over time, we will want to add new phone numbers to our list. We need a method of adding a new phone number to the phone list without typing our entire phone list over again. LISP has the function CONS to add new items to a list:

```
>(CONS '(E F) '((A B) (C D)))
((E F) (A B) (C D))
>(SETQ LIST1 '((A B) (C D)))
((A B) (C D))
>(CONS '(E F) LIST1)
((E F) (A B) (C D))
>LIST1
((A B) (C D))
```

Are you still confused about the quote mark? Don't worry, understanding the use of QUOTE is difficult. When LISP sees (CONS '(E F) '((A B) (C D))), it starts evaluating the s-expression that starts with (CONS. It checks that value of CONS and finds that it is a function with two arguments. The CONS function wants both of its arguments evaluated.

LISP then evaluates '(E F). The result if (E F), which is passed as the value of the first parameter of CONS. The second parameter is evaluated and, because of the quote mark ((A B) (C D)), is passed as the value of the second parameter of CONS.

If we had left the quote marks off, LISP would have evaluated the first argument to CONS as (E F). This would mean that E was a function that expected one parameter. If E were really a function, LISP would evaluate F and pass its value as a parameter to the function E.

In our case E is *not* a function, it is an atom. If LISP had tried to evaluate the expression (E F), an error would have been produced. In the expression (CONS '(E F) LIST1), we did not want to put a quote mark before LIST1. Why not? If we had entered:

```
>(CONS '(E F) 'LIST1)
((E F) LIST1)
```

would have been the result. What we wanted to do was to CONS together (E F) and the *value* of LIST1. By not having a quote mark before LIST1, LISP evaluates LIST1 and passes the value of LIST1 as the second parameter to CONS.

Note that CONS is nondestructive. Remember, LISP works like a calculator: It always evaluates an s-expression and prints the results. Using cons does *not* change the value of LIST1. To replace LIST1 with LIST1 plus (E F), we must do the following:

```
>(SETQ LIST1 (CONS '(E F) LIST1))
((E F) (A B) (C D))
>LIST1
((E F) (A B) (C D))
```

For our purposes, it doesn't matter if we add new phone numbers to the beginning of our phone list or to the end. We will create a function that adds the phone number to the beginning of the list (later we will show why we made this choice).

```
>(DE ADDPHONE '(NEWPHONE)
                   '(SETQ PHONELIST (CONS
                    NEWPHONE PHONELIST)))
```

```
ADDPHONE
>(ADDPHONE '(SMITH 8884567))
((SMITH 8884567) (GREEN 5335500) (GREER
                    7347589) (OVERTON 6883993))
```

There is a problem with this routine. Every time that we add a new phone number, the *entire* phone list is printed. It would be better if ADDPHONE returned only the name and phone number of the new person.

To do this we must introduce a new LISP function. The function CAR returns the first element of a list. The strange word CAR comes from the early history of LISP, and its meaning must be memorized. Some examples of using CAR:

```
>(CAR '(A B))
A
>(CAR '((A B) (C D)))
(A B)
>(CAR '((SMITH 8884567) (GREEN 5335500)
                          (GREER 7347589)))
(SMITH 8884567)
```

Using CAR, we can modify the ADDPHONE function to return the new name and phone number only:

```
>(DE ADDPHONE '(NEWPHONE)
                '(CAR (SETQ PHONELIST
                              (CONS NEW-
                        PHONE PHONELIST))))
ADDPHONE
>(ADDPHONE '(SMITH 8884567))
(SMITH 8884567)
>(ADDPHONE '(JONES 4572311))
(JONES 4572311)
```

We now have the ability to check our phone list for the phone numbers of specific individuals. We can also add new phone numbers. To obtain a complete list of phone numbers, we enter the name of our phone list.

There are problems with this. One is "information hiding." By building a nice user interface to our phone number list, the user should *never* need to know the name of our phone list. Another problem is that the user has become used to entering questions within parentheses and typing the name of the phone list is different from all of our other functions. We may want to change the format of the listing for the phone list. Defining a function will permit us to change the format later *without* affecting the user.

Here is a function to print out the list of telephone numbers:

```
>(DE PRINTPHONE '( ) 'PHONELIST)
PRINTPHONE
>(PRINTPHONE)
((GREEN 5335500) (GREER 7347589) (OVERTON
                                  6883993))
```

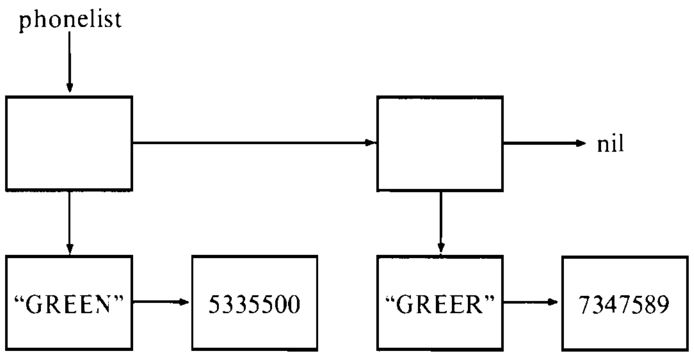
There are two special points to note about this function. There are *no* parameters to PRINTPHONE. The parameter *list* must be specified, but we do not put any names in the list (that is, it is empty).

The second point is that the body of the function is *not* a list! We used 'PHONELIST to stop LISP from evaluating PHONELIST when it was defining the PRINTPHONE function. When we enter (PRINTPHONE), LISP evaluates PHONELIST, which just returns its value.

We would like to be able to delete phone numbers. This requires more functions from our LISP bag of tricks. Our new function is CDR, the opposite of CAR. While CAR returns the first element of a list, CDR returns the rest of the list.

```
>(CDR '(A B))
(B)
>(CDR '((A B) (C D)))
((C D))
>(CDR '((GREEN 5335500) (GREER 5347589) (OVER-
                              TON 6883993)))
((GREER 5347589) (OVERTON 6883993))
```

A picture may help you understand what we plan to do with our internal phone list. Assuming that we have two names on our list, it would look like this:



In fact, this diagram is abbreviated. The lists that start with GREEN and GREER both end in nil. Actually, all lists in LISP end in nil.

This diagram should help show how CAR and CDR works. If we do:

```
>(CAR PHONELIST)
(GREEN 5335500)
```

we obtain the *list* with GREEN and his phone number. If we do:

```
>(CAR (CAR PHONELIST))
GREEN
```

we obtain GREEN because we went down two levels in the internal LISP structure. To delete a person from our phone list, we must do two things: We must find the person's name on our list, and we must generate a new list with everything before the person and after the person on our phone list.

We will need a LISP function to test if one item is equal to another. The name of this function is EQUAL.

```
>(EQUAL 'GREEN 'GREEN)
T
>(EQUAL 'GREEN 'GREER)
NIL
>(EQUAL 'GREEN (CAR (CAR PHONELIST)))
T
>(SETQ NAME 'GREEN)
GREEN
>(EQUAL NAME (CAR (CAR PHONELIST)))
T
```

It is tiresome to always type (CAR (CAR PHONE-LIST)). Because this is a very common operation in LISP programs, there is a function to do this for us, CAAR:

```
>(CAAR PHONELIST)
GREEN
>(EQUAL 'GREEN (CAAR PHONELIST))
T
```

How should our DELPHONE function work? It should be given the name of the person to delete and a list to search and delete from. We check the first item on the list to see if it matches the name. If it does, we return the rest of the list (the CDR). Otherwise, we take the beginning of the list (the CAR) and add it together with the rest of the phone list (the CDR) that does not contain the name (using our old friend CONS).

The statement of how this function works is recursive. We will keep calling our DELPHONE routine with "the rest of the list," until we reach the end of the list or we have found the name that we want to delete. The resulting LISP code looks like:

```
>(DE DELPHONE '(NAME PLIST)
              '(COND ((NULL PLIST) NIL)
                     ((EQUAL NAME
              (CAAR PLIST)) (CDR PLIST))
                     (T(CONS(CAR PLIST)
                           (DELPHONE
                     NAME (CDR PLIST))))))
DELPHONE
```

This is the most complicated function that we have developed. Work through each line of code and check that you understand what is going on. Use the diagram on page 10 and the following LISP function calls to test your knowledge of the DELPHONE function:

```
>(DELPHONE 'GREEN PHONELIST)
((GREER 7347589))
>(DELPHONE 'GREER PHONELIST)
((GREEN 5335500))
```

This function is nondestructive because we use CONS. A *new* phone list is created when we call DELPHONE. We need an intermediate function that sets the value of the phonelist for us and that is more user friendly.

This routine will be called DELPHONE; the one that we just developed is DELPHONE1. We will use code from GETPHONE to check that the name we want to delete already exists on our phone list.

```
>(DE DELPHONE '(NAME)
              '(COND ((NULL (ASSOC
                       NAME PHONELIST))
                       "No such person")
                     (T (SETQ PHONELIST
            (DELPHONE1 NAME PHONELIST)))))
```

With these functions you can add, retrieve, list, and delete telephone numbers for individuals. More importantly, you should be able to develop your own LISP functions.

Preparing the HP 3000 data center for an audit*

Betsy Leight

Operations Control Systems
Palo Alto, California

INTRODUCTION

The word auditor has unfortunate connotations. The data-processing professional facing an internal audit is often reminded of an Internal Revenue Service agent meticulously scrutinizing tax forms in a desperate search for illegal deductions.

In fact, the DP auditor is more correctly defined as an efficiency expert. By analyzing the various aspects of the DP department from an objective viewpoint, the auditor can make recommendations that will benefit both DP operations and the company as a whole.

The responsibilities of the DP auditor can range from questioning the integrity of an immensely complicated software system to advising DP personnel to take their coffee cups off the line printer. The auditor's concerns will vary with his or her expertise. While ideal DP auditors are as comfortable with the internal workings of an HP 3000 as they are with basic accounting principles, ideal auditors are scarce. But whatever the extent of his or her technical knowledge, the most important qualifications for effective DP auditing are nothing more complex than common sense, a systematic approach, and some basis for comparison with similar data-processing environments. As long as the auditor isn't afraid to ask questions, all the pertinent information will be available somewhere.

Various articles on the auditing function have divided the DP auditor's concerns into lists ranging from 7 to 65 relevant items. There are actually only two: *efficiency* and *security*. These two categories do split into quite a few subsections, and the purpose of this paper is to provide a list of the concerns most often overlooked by the typical HP 3000 shop. In addition, we will focus specifically on data center operations rather than on other areas, such as systems and programming.

STANDARDS AND PROCEDURES

The DP auditor can be particularly helpful in ensuring the existence and enforcement of proper standards and procedures. If everyone does things the same way, it's even money they're doing them right.

Standards and procedures in the data processing center should include getting the right program in operation at the right time; inserting changes into programs at the right time: getting the correct data to the right program at the right time; protecting the data and program from accidental or intentional destruction; and determining that the data processed is complete and accurate.

Other standards should specify methods of physically moving inputs and outputs; procedures for controlling data, programs, and the flow of work; methods of scheduling work; methods of getting work rerun in the event of error or disaster; record-keeping of work accomplished and resources available to do the work; determining that there are sufficient resources available to do the work; and maintenance and other housekeeping associated with the operation of the computer center.

Furthermore, the DP auditor should verify that formal standards exist for system development and maintenance, program/system testing, file conversion, program/system change control, library operations, computer operations, and documentation.

OPERATIONAL WORKFLOW AND CONTROLS

There are quite a few items for the DP auditor to investigate in this area. Firstly, is input data from other departments complete and entered on time? Does the data center keep job-accounting information, and is it evaluated and used by management?

The control of errors is scanned: Is anyone notified in case of a production processing error? Are errors documented? Are error statistics accumulated or ignored? Finally, the auditor checks that errors are followed up on so that they do not reoccur.

The auditor also checks that downtime is reported and statistics maintained on it. There should be a log of late reports and/or jobs.

Communication between departments is studied. The auditor monitors whether there is a formal communications channel between operations and other departments. He or she also looks into whether operation tips and gotchas get passed around to all operators through a formal channel.

*Adapted from a recent presentation

Similarly, the auditor checks that problems encountered at the computer are documented, along with the effective action taken to prevent recurrence, if any. Operators must get feedback as well on reported problems. The auditor further checks that headers and $STDLIST information gets used and checked.

Distribution and disposal is scrutinized: How is it known whether all reports and/or microfiche have been distributed to the proper user? Have procedures been established to control the distribution of sensitive output? Do procedures include the method of disposing of confidential reports when they are no longer required?

Finally, the auditor reviews whether jobstream-run instructions are kept up-to-date.

SCHEDULING

Efficient production scheduling is extremely important in providing a high level of reliability and predictability to the data-processing operation. Ensuring efficient scheduling is an important function of the technical auditor.

Among the points the auditor will raise are whether daily processing activities are scheduled and if there is a daily contingency schedule. For batch production, are actual run times recorded? Is this data used to calculate expected run times for a given day? Are expected run times compared against actual run times to ensure that runs have not been terminated abnormally? The auditor further notes whether a firm nightly/batch schedule has been established and adhered to.

The auditor investigates whether unscheduled runs are supported by a work request or some other written authorization. Documentation of schedule deviations and follow-up by a supervisor are also checked.

With regard to the online environment, the auditor notes whether user-submitted jobs are recorded to allow forecasting of future schedules, resource requirements, and special processing considerations.

All jobs should be submitted through or controlled by operations. Another review item is whether all output is routed by operations to the appropriate destination or picked up by the users.

Lastly, the DP auditor checks that there are standards for the type, quality, and quantity of forms kept on hand.

DATA SECURITY AND ACCESS CONTROL

It may come as a surprise, but many successful businesses have information in their databases that should be protected from loss and kept from the competition. With this end in mind, the DP auditor investigates several concerns.

Firstly, are data-processing employees instructed as to their responsibilities concerning confidential information? Management must further periodically review and update controls and security provisions relating to data. Live production programs should be physically separated from development programs, and all staff should be prohibited from running test programs against live files. Operations personnel should also be denied access to sensitive data files.

Are procedures in effect covering the acceptance and transference of programs from development into production? Are program library changes approved and accounted for? The auditor also checks that the acceptance testing of changed programs is approved by operations before transference to production libraries, and that the approval includes assurance that production documentation is updated.

Operators should be prohibited from renaming or transferring programs without prior supervisory approval. Internal labels must be used for all data and program files.

In addition, accounts, users, and data files should be protected by passwords and lockwords. Auditors expect procedures that will maintain passwords and ensure that they are changed on a periodic basis. Access violations must be logged and reported to the security manager. An automatic logoff feature assures that unattended terminals no longer present security threats. Passwords, lockwords, dates, and constants should be inserted into jobstreams as they are launched, eliminating the need to "hardcode" sensitive data into jobstreams.

The auditor looks into the area above the suspended ceiling in the computer room—is it accessible only from that room?

Finally, the auditor investigates blank checks and other negotiables: Are they issued on run-schedule basis only? Are they kept in a secure area when unattended? Are they controlled by access forms and periodically inventoried?

EQUIPMENT UTILIZATION AND EFFICIENCY

One unit of measurement for DP efficiency is the U.S. dollar. Auditing is, after all, a business function, and one measure of business success is money earned. So when we say efficiency, we mean cost-efficiency.

Once it has been determined that the entire data-processing department is following an impeccably implemented set of standards and procedures, the auditor's attention turns to efficient equipment utilization. This can be a particularly tricky area, especially if the auditor is an MBA graduate with no previous computer experience. The key here is for the auditor to ask questions openly about procedures and daily phenomena.

How much machine time is spent on reruns, for example. Are reruns analyzed? Are certain jobs especially susceptible to reruns? The auditor should also check whether there are programs or jobs that are inefficient in the area of file design or utilization. Another item to check is if the full multiprogramming capability of the HP 3000 is being utilized for batch production. Are multiple job streams run concurrently? Are CPU-bound and I/O-bound jobs mixed to maximize overall throughput? The auditor then reviews whether many jobs are restartable without rerunning the entire job.

PERSONNEL UTILIZATION AND EFFICIENCY

This is a sensitive subject. The auditor is not likely to make many friends if he or she wanders around the DP department with a clipboard in hand and jots notes while the operators discuss football scores. Here are some delicately phrased personnel questions that the auditor may use to guide his or her investigation.

Do operations personnel require extensive training and experience in order to be effective in processing daily production work? Do operators require extensive knowledge of each application they run?

Is there a system to schedule and monitor normal daily processing? Is the system effective? Does it operate without excessive manual involvement? Or do operators spend a large percentage of their time tracking jobs in execution, replying to program messages, changing the fences, and so on? Are operators required to modify jobstreams at run time?

Are all necessary tapes, forms, and other resources available when needed?

Is there excessive turnover? Is daily production dependent on any specific individual(s)?

Is the operations department treated as a stepchild of the data-processing department? (After all, the HP 3000 doesn't require an operator!)

DISASTER PLANNING AND RECOVERY

This is a catch-all category that includes everything from proper insurance planning to keeping terrorists out of the computer room. The first item to investigate is the existence of an emergency plan that is adequate in relation to the risk. This plan should be kept current and distributed on a "need-to-know" basis only.

The plan should include action for offsite storage of files and documentation. It should specify

- the conditions for use of offsite processing
- application priority
- resource requirements
- job scheduling
- run documentation
- and required tapes, forms, and supplies.

Finally, formal written procedures for hardware backup should be instituted.

ENVIRONMENT

Now here is something for the nontechnical auditor to sink teeth into. It doesn't take a great deal of scientific expertise to mention that people shouldn't have to climb over the disc drives to get to the water cooler.

A basic concern is that the workspace is adequate. In addition, it should be neat, and supplies should be quick to locate when needed.

Tape drives should be cleaned and reels certified regularly.

Auxiliary items located outside the computer room, such as bursters and decollators, should be accessible for the flow of work of the department. Tapes, discs, and the like should be stored in a closed, fire-protected, and limited-access area.

ASSISTING THE AUDITOR

The best way to assist a DP auditor, assuming you are so inclined, is to provide as much information as possible. This can be done by reviewing the previous pages and answering any questions or gaps in advance.

A more efficient method of providing the auditor with information is to implement a software system that leaves clearly defined audit trails and that issues a wide variety of reports. This saves valuable time from being wasted ferreting out information.

In fact, perhaps the first suggestion of each data-processing audit report should be the installation of a better system to gather the proper data and make the job easier the next time.

Journal
Interex
2570 El Camino Real West, Fourth Floor
Mountain View, California 94040
U.S.A.

BULK RATE
U.S. POSTAGE
PAID
PERMIT NO. 382
MOUNTAIN VIEW, CA
and Other Locations