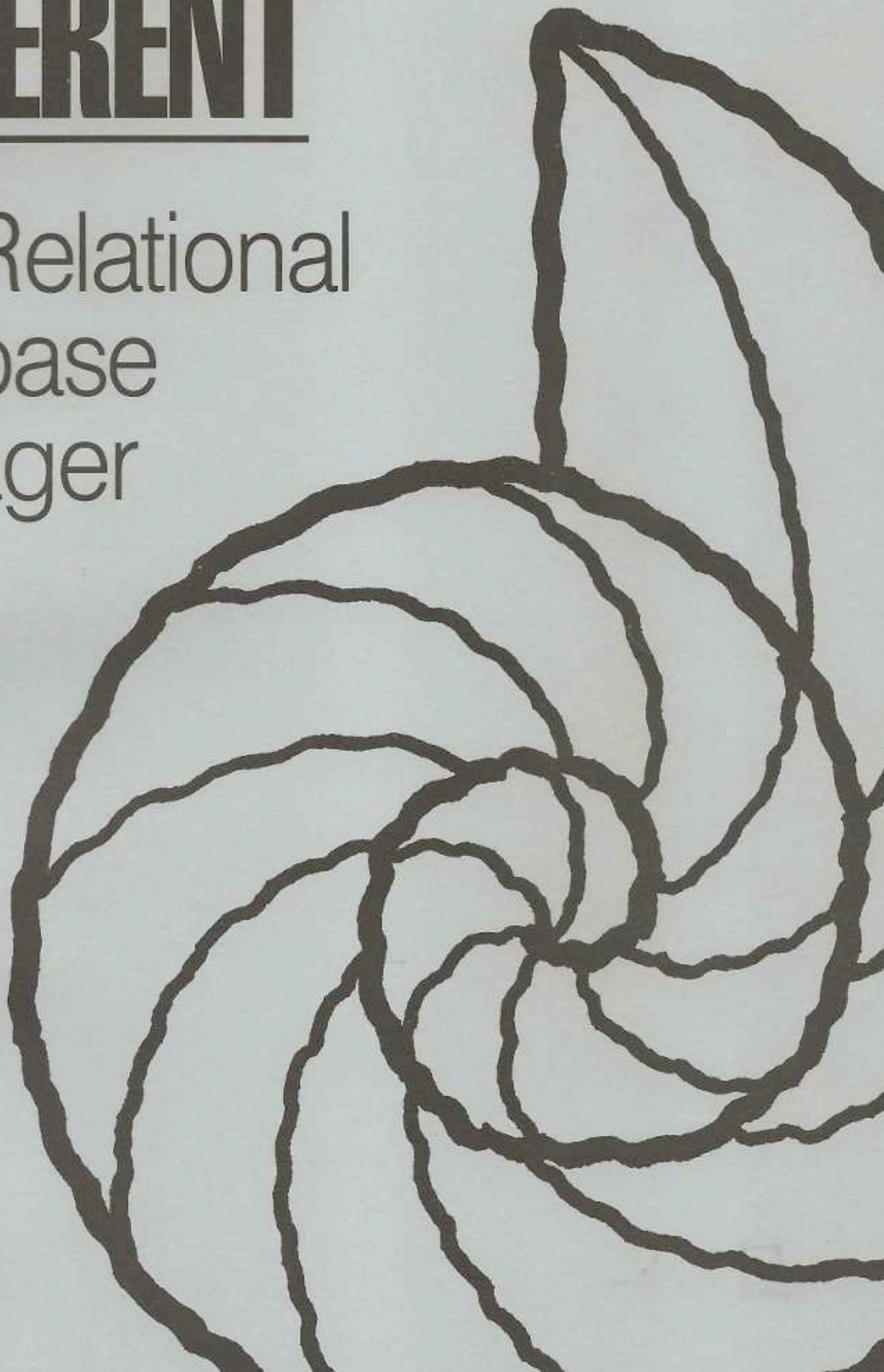
COHERENT™

/rdb Relational Database Manager



Revision 1 Printing 5 4 3 2 1

Published by Mark Williams Company, 60 Revere Drive, Northbrook, Illinois 60062.

E-mail: uunet!mwc!support (Technical Support)
support@mwc.com
uunet!mwc!sales (General Information)
sales@mwc.com
BIX: join mwc
CompuServ: 76256,427

This manual was written under the COHERENT operating system, using the MicroEMACS and **ed text editors. Text formatting was performed by the COHERENT edition of the troff text formatter, using its PostScript output function. Typesetting of this manual, from the table of contents through the index, was performed by one script written in the COHERENT Korn shell. Camera-ready copy was printed on a Hewlett-Packard LaserJet IIP printer using the Pacific Page PostScript cartridge.**

Printed in the U.S.A.

Table of Contents

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Introduction

/rdb is a relational data-base management system for COHERENT, MS-DOS, UNIX, and other operating systems. It is designed to use the full power of the COHERENT shell and supplemental tools, including **awk**, **sed**, and the C language.

With **/rdb**, you can implement a full-featured relational data base, including a menu-driven data input and a report writer. Also, because large portions of **/rdb** are written in the shell and related COHERENT tools, **/rdb** serves as an excellent tutorial in learning how to use the full power of COHERENT.

Installing /rdb

To install **/rdb**, log in as the superuser root. Then **cd** to the directory in which you want **/rdb** to live, say **/usr**. Then use the COHERENT command **install** to install the program. **install** has the following syntax: **install** *version device disks*, where *version* is the version number of the program being installed, *device* is the floppy-disk drive from which the program will be installed, and *disks* is the number of disks that compose the installation.

The version number appears on your **/rdb** manual. The number of disks is one. If you are installing from a high-density 5.25-inch floppy disk in drive 0 (drive A), *device* is **/dev/rha0**; if you are installing from a 3.5-inch floppy disk in drive 1 (drive B), then *device* is **/dev/fva1**. See the COHERENT Lexicon entry for **floppy disks** for a table of disk sizes and devices.

For example, if you are installing version 2.0 from a 3.5-inch disk in drive 0, use the command:

```
/etc/install 2.0 /dev/fva0 1
```

COHERENT will take care of the rest.

Be sure to edit your **.profile** to include the directory **$RDB/bin**, where **$RDB** is the directory into which you installed **/rdb**. For example, if your **.profile** the **PATH** variable as follows:

```
PATH=/usr/bin:/bin:$HOME/bin:.
```

and you have installed **/rdb** into directory **/usr/rdb** (which is the default), then you should edit your **PATH** statement to read as follows:

```
PATH=/usr/bin:/bin:/usr/rdb/bin:$HOME/bin:.
```

European Keyboards

COHERENT version 3.2 lets you install any number of predefined keyboard definitions into your keyboard driver. These let COHERENT support the keyboard layouts and national character sets for most European countries.

If you are using **/rdb** with one of COHERENT's European-ized keyboard definitions, you must rework three programs. These programs are the table editors **ve** and **jve**, and **vindex, ve**'s hash indexer. The European versions of these programs are located in directory **$RDB/europe**.

To use them in place of the default US keyboard versions, put the path **$RDB/europe** before **$RDB/bin** in your **PATH** variable. For example, if **/rdb**'s home directory is **/usr/rdb**, then insert the following entry into your **.profile** file:

```
export PATH=:/usr/rdb/europe:/usr/rdb/bin:"$PATH"
```

Or you can simply replace the US versions with the European ones:

```
cd /usr/rdb
mv ./europe/* ./bin
```

A side effect of the European versions of these programs is that characters that exceed the decimal value of 127 are highlighted. In other words, characters such as umlauted and accented vowels, tilded n's, currency symbols, etc., will be highlighted in the screen displays for **ve**, **jve**, and **vindex**. To suppress highlighting of eight-bit characters, make a new entry in your **/etc/termcap** file by copying the entire description for your terminal type and deleting the **so** (or *standout*) and **us** (or *underscore*) entries, up to and including the ':'. For example, the following copies the **termcap** entry **ansipc**, modifies it for European use, and makes it into a new **termcap** entry called **an-e**. The portions in **boldface** are deleted:

```
ap-e|ansipc-eur|ansipc for european kbs:\
       :al=\E[L:am:bs:bt=\E[Z:bw:cd=\E[O:ce=\E[K:ch=\E[%i%d':cl=\E[2O:\
       :cm=\E[%i%d;%dH:co#80:cs=\E[%i%d;%dr:cv=\E[%i%dd:dl=\E[M:ho=\E[H:\
       :k0=\E[0x:k1=\E[1x:k2=\E[2x:k3=\E[3x:k4=\E[4x:\
       :k5=\E[5x:k6=\E[6x:k7=\E[7x:k8=\E[8x:k9=\E[9x:\
       :kb=^h:kd=\E[B:kh=\E[H:kl=\E[D:kr=\E[C:ku=\E[A:\
       :li#24:ll=\E[24;1H:hd=\E[C:se=\E[m:sf=\E[S:sg#0:so=\E[7m:sr=\E[T:\
       :te=\Ec:ue=\E[m:up=\E[A:us=\E[4m:\
       :KI=\E[5x:KD=\E[3x:Kd=\E[P:KB=\E[6x:KU=\E[4x:Ku=\E[@:KM=\E[7x:KJ=\E[8x:\
       :Kt=\E[Z:KT=\t:KL=\E[1x:KR=\E[2x:KP=\E[U:Kp=\E[V:KX=\E[9x:KC=\E[0x:\
       :KE=\E[24H:KW=^F:Kw=^R:Kr=^N:do=\E[B:
```

Once you have installed the new termcap entry, set your **TERM** variable in your **.profile** to your new **termcap** entry:

```
export TERM=ap-e
```

Data Bases and COHERENT

There are two fundamental ideas that are necessary for application development in a COHERENT environment. One is the relational model, which allows you to see all applications as simple operations on tables. The other is the idea of using COHERENT tools to pass data through pipelines and filter programs, instead of traditional programming.

Tables and COHERENT

First, we will introduce you to tables. All information can be entered into and manipulated in tables. This is the fundamental idea of relational data bases. The COHERENT system itself provides a rich and varied environment in which tables can be manipulated. After covering basic operations, you'll see simple data-base design principles. Then you will learn how to use COHERENT tools. You will also be introduced to advanced data-base problems and solutions. There are a number of examples illustrating common data-base problems and solutions, and a model business operation and accounting-system tutorial and manual is included as a case study. Finally, all the manual pages are appended.

Operating Systems that Support COHERENT-Style Environments

By the COHERENT environment, we mean file redirection, pipes, and COHERENT-style tools like **awk**, **grep**, and **sed**. This environment can be on a computer running COHERENT, any of the various flavors of COHERENT, or MS-DOS with UNIX tools added.

Applications Development

When a person uses a computer in his work, he usually needs to have it programmed for his particular application. A scientist might study the data from his experiment, an entry clerk takes orders or reservations, a business person manges his business, an accountant prepares financial statements and tax forms, a lawyer does research and prepare briefs, and so forth. Some users can

TUTORIALS

develop their own applications with COHERENT tools themselves. But most users need to have computer professionals set up applications for them. In this case, the power of the COHERENT environment eases the job of the developers and extends what can be done.

Data Bases

Almost all applications require some form of data base. Typically, a data-base management system is a computer program that maintains data and performs tasks such as easing the input and validation of data, finding data, computation and transformation of data, reporting, and so forth.

COHERENT Environment

The COHERENT system is not just an operating system. It has hundreds of programs that work together to do most of the common computer jobs. COHERENT tools provide an entire support environment for software application development and operation. It does so much of the work, that the jobs of both the developer and the user are enormously simplified. But since this environment is new to the computer profession, it is seldom used to its maximum potential. Often, developers don't know how to do on the COHERENT system many of the things they are used to doing on other systems. Frequently, developers do things in the COHERENT environment in difficult or slow ways that could be done much better if they knew how.

Many data base systems are now available on the market. But almost all are software prisons that you must get into and leave the power of COHERENT behind. Most were developed on operating systems other than UNIX or COHERENT. Consequently, their developers had very few software features to build upon, and wrote the functionality they needed directly, without regard for the features provided by the operating system. The resulting data base systems are large, complex programs that degrade total system performance, especially when they are run in a multi-user environment.

COHERENT provides hundreds of programs that can be piped together to easily perform almost any function imaginable. Nothing comes close to providing the functions that come standard with COHERENT. Programs and philosophies carried over from other systems put walls between the user and COHERENT, and the power of COHERENT is thrown away.

The shell, extended with a few relational operators, is the fourth-generation language most appropriate to the COHERENT environment.

Fourth Generation Systems

In recent years, a variety of developments in programming language design have emerged. Object-oriented languages are becoming common, and languages explicitly supporting multiple tasks and inter-task communication are also gaining popularity. Unfortunately, these efforts have resulted in productivity increases too small to offset the growth in the size and complexity of software systems. A response to this has been the development of fourth-generation programming languages. Although not commonly thought of as such, the COHERENT shell is one of the most powerful and flexible fourth-generation languages available.

Attempts at a Definition

There is no consensus on the definition of what constitutes a third- or fourth-generation language. Mainstream third-generation languages are typed, procedural languages. They are standardized and largely hardware independent. Operations in the language must be specified in a detailed, step-by-step algorithmic fashion. Third-generation languages do very little implicit processing. Third-generation languages are general purpose, even most of those that were ostensibly designed as special purpose languages.

Fourth-generation languages are usually intended as design tools for a particular application domains. They are usually free form in their use of variables, often not requiring type definitions and allowing dynamic typing of variables. They don't emphasize a modular, procedure-based style

of coding. Instead, they contain a number of predefined procedures for performing various high-level operations. The high-level operations involve large amounts of implied processing. For example, a "sort" operator is usually available. The facilities of a fourth-generation language are usually both more powerful and less flexible than the facilities available in a third-generation language.

A fourth-generation programming language (4GL) should let you simply write what it is you want, rather than a detailed procedure of how to produce it. Although many products call themselves 4GLs today, they are mostly rewrites of COBOL and report writers. They are too low level and tedious. This is definitely not what a 4GL should be.

Previous Generations

The first generation of computer languages was the sequence of zeroes and ones that were the machine instructions. In the beginning people had to code in this way.

The second generation was "assembly language," which has a one-to-one correspondence with machine instructions. For example, this assembly-language code adds register 1 to register 2:

```
add r1,r2
```

One line of code produced one computer instruction. Then, in about 1956, FORTRAN was written to do *formula translation,* and it became much easier to write programs. Each line of code produced several computer instructions. The third generation has come to encompass sophisticated macro assembly languages, and other so-called high-level languages like C, COBOL, Pascal, LISP, PL/1, and many others. There are advanced constructs close to English like **if**, **then**, and **else**, but the types of statements are constrained to mostly arithmetical operations, with limited string capabilities. A typical third generation program consists of statements like:

```
for i=1 to 10
       print i, sqrt(i)
next i
```

The next step comes in describing what you want and letting the computer figure how to give it to you. The fourth-generation has English-like words, but statements typically deal with more than numbers, and are "non-procedural." A fourth-generation environment, for example, reduces a program to sort all the lines in a file to:

```
sort file
```

Fourth-generation language primitives often include relational operators, while third-generation languages generally do not. And, when you need to mix procedural with non-procedural instructions, that is easy to do. For example:

```
for file in file1 file2 file3
do
       sort $file
done
```

At the COHERENT shell level, you can in many cases say what you want without saying how (non-procedural), and you will get it. For example, type the command

```
sort file
```

in the the COHERENT shell and you get a sorted file. Or, type

```
spell file
```

and you get a list of words in your file that are not in the dictionary. One line of commands calls one or more programs, each of which may have thousands of instructions.

TUTORIALS

With the shell, you can put together an application in minutes or hours, instead of the weeks or months required with 3GL code. In a 4GL, you should be able to write most applications in a line or two. With the shell, you can say things like:

```
column Account Amount Description  < file |
grep expenses |
sort |
lpr
```

This short program extracts some of the columns in a file, pipes them through another program to select the rows that contain the string **expenses**, sorts the selected strings, and prints them.

This same report would take tens to hundreds of lines in COBOL, PL/I, C, and most commercial 4GLs. In those languages, you write instructions one at a time, to process records from the file one at a time. This is very tedious compared to writing one instruction to operate on the entire file.

Data Structures in the Data

In an ideal environment, the structure of data is in the data. Newline-separators for records and column-separators for fields can tell any program where the fields and records are. 3GL languages must have the data structure hard-coded into them, so that a program can read only one kind of file.

In a traditional third-generation environment, the structure of the file must be hard coded into the program. In a fourth-generation environment, files have their structure embedded with newline and tab record and field separators. Any program can find a record by just looking at the stream of characters. Add a single character to the data file read by a COBOL program and all is changed or lost, so you must do file conversions in the COBOL environment all of the time. And these are done in COBOL. Any changes require that you edit and recompile all of the programs that read that file.

In addition, there are no file operators in 3GLs, only field-at-a-time instructions. Therefore, you must write loops to process each record. This takes much more code than just processing a whole file at a time.

Most commercial 4GLs are very similar to COBOL. You still have to do record-at-a-time processing. If the COBOL program takes 100 lines, the 4GL will take anywhere from 50 to 100 lines, to do what we did above in one pipeline.

A Revolution in Computing

If you write C programs on COHERENT, you miss most of the advantages of shell-level programming. It's been suggested that since C and other languages on COHERENT give you the **system()** function call, this converts them into 4GLs. This is equivalent to saying that assembler is a 4GL if it has a **system()** function. But on non-COHERENT operating systems like MS-DOS and VMS, there is not as rich a variety of tools available as in COHERENT, except to the extent that COHERENT has influenced these systems.

The COHERENT system itself offers an integral tree index approach to data organization: the hierarchical file system itself. Many utilities traverse these trees, search them, add and delete nodes, and in general provide procedural tools with which to deal with files. The same is true of MS-DOS and Macintosh systems. The opportunity is afforded to avoid re-inventing the wheel.

This really is a revolution in computing. Working with great tools will spoil you, but most of the computing world is still writing COBOL. To have to go back and forth between such environments is painful.

A good 4GL should be written in C — once. It should be written to be so general in purpose and so easy to use that its functionality can be used from then on, rather than recoded in each application. Then these good programs can be used to put together applications, not coding each entire application in C, unless there is some critical need.

As you become more familiar with their environment, you will become more able to use the power of these advanced systems, if only to shorten repetitive command sequences — another key feature of 4GLs. Every computing environment has facilities to collapse a sequence of keystrokes; these include aliasing, scripting, and macro construction.

Marketing people got wind of 4GL and turned it into a big marketing hype. Most data base management systems wrote their own procedural language like COBOL or RPG and called it a 4GL. They are usually worse than COBOL, because you have to learn their new language, rather than use a classic. Few 4GL designers put as much time and energy into designing their language as was put into COBOL.

The driving force behind fourth-generation languages comes from several needs. Programming projects commonly involve man-years of work. The shortage of experienced software engineers and the need to increase productivity pushes us towards tools allowing faster development cycles. The increased use of computers by users who do not have formal computer science education requires very high-level tools that let novice programmers concentrate on algorithms rather then implementation details. As more work is done on computers, there is more demand for single-use programs to perform a specific task. The relatively high expense of coding a software tool with a one time use encourages the use of any method available for simplifying the development process.

As third-generation languages are becoming increasingly less able to meet the diverse needs of computer users, several principles of software design are gaining great popularity, especially within the UNIX/COHERENT community:

- Data should be kept in flat ASCII files, not binary, so that we can always see what we are doing, and do not have to depend upon some special program to decode our data for us.
- Programming should be done in fourth-generation languages, except when the expected heavy use and/or resource consumption of the program justify the expense of a more efficient coding in a third generation language.
- Programs should be small and should pass data on to other programs. Software prisons, or large programs with self-contained environments, must be avoided because they require learning and they make extracting data difficult.
- We should build software and systems to meet interface standards so we can share software and stop dreaming that any individual or company can do it all from scratch.

Approaching software engineering with principles like these does have some drawbacks. The major drawback is that fourth-generation languages almost always produce slower code then third generation languages do. As computers increase in speed and power, this drawback becomes less of a consideration. As improved compiler optimization techniques spread, the difference between code produced by 3GLs and 4GLs will become smaller.

A Paradigm

A programming paradigm is important for ensuring that a language is robust and has a consistent style to its syntax and semantics. Paradigms for fourth-generation languages must meet requirements more stringent than those for third-generation languages. To start, a fourth-generation language should provide a consistent interface to high-level facilities working with a variety of complex data types, while simultaneously providing fundamental low-level language constructs for coding any functionality missing from the predefined facilities. Too many 4GL's are good only for projects within a narrow application area. It's difficult to allow for high-level constructs from a variety of fields without the programmer having to specify the level of detail required in a 3GL.

TUTORIALS

The paradigm we choose for fourth-generation languages is the operator/stream paradigm. In this model, data flows in unidirectional "streams" on which operators are placed. Each operator transforms the data as it passes by. The set of streams in a program form a directed graph, possibly with cycles. This paradigm concentrates on what needs to be done to the data, and deemphasizes the techniques used in the transformation.

Fourth-generation languages that attempt to use only the procedural paradigm of mainstream third-generation languages usually end up being limited to a specific application domain. The procedural model doesn't describe data in an abstract enough way. Different types of operations require too much detailed code to work with, and the languages do not have the simple relation between all data and operators the way an operator/stream paradigm does.

A side benefit of using the operator/stream paradigm appears in the design of graphical programming tools. Traditional third-generation languages haven't been well adapted to a graphical programming interface. The problem stems, in part, from the difficulty of expressing the numerous possibilities in an intuitive pictorial way. With operator/stream as the basis for a language, a graphical programming aid can easily convey the process of placing an operator on a stream.

The operator/stream paradigm has proven effective in more domains than just language design. Some UNIX kernels make use of the paradigm to reduce the complexity of the operating-system code. Rather than having one large, complex piece of code handling all the functionality of a particular aspect of the operating system (such as a device driver), data in the kernel is run through a linked set of operators, each operator performing one small, well-defined function. This lets users modify the system by introducing new operators, without having to understand the innards of other operators on the stream.

The Shell

The shell and the set of COHERENT utilities form a fourth-generation language based on the operator/stream paradigm. The critical feature of the shell that puts it in the class of 4GL's is the COHERENT *pipe*, which allows a shell to start a sequence of processes, each reading its input from its predecessor process and writing its output to a successor process. The pipe is one of the major reasons leading to the adoption of UNIX and its offspring, including COHERENT, as the standard multi-user operating system. Unfortunately, few people fully understand the philosophy behind it; most software developers are still producing large, self-contained applications using data formats incompatible with anything else.

For the shell, the COHERENT pipe provides the data streams, and the hundreds of standard COHERENT utilities provide the core set of operators. The power of this approach is tremendous. Because the data streams are flat ASCII, all the operators can read each other's data. COHERENT includes a few standard utilities that can perform most of the data formatting needed to transform one program's output to the form required by another. In addition, using stand-alone programs as operators allows easy use of custom or commercial packages of operators, such as statistics or data base packages. This modularity encourages code reuse, and the flat ASCII stream format makes it easy to get operators from a variety of sources talking to each other. Finally, because the operators can only transform the data stream running through them, side effects can't surprise the software engineer by giving unexpected results.

The COHERENT file system also provides a hierarchal storage medium for data. Because COHERENT files are flat ASCII data files, and COHERENT deliberately attempts to make all data sources look the same, most utilities can't distinguish between data coming from a stream and data coming from a file. This gives great flexibility, allowing the shell to store the results of a pipeline into a file, and then feed that data back into a stream at some future point.

There are two frequent criticisms of fourth-generation languages. It is often noted that 4GL's tend to be suited for a particular application area, and that their low-level facilities are not up to the task of providing complex functions that don't already exist in the high-level library. The shell escapes this problem: COHERENT utilities can be written in any language, from shell scripts to assembly

language. If a tool is needed which isn't currently available, the developer is free to pick the language most suited to solving the problem, whether it's a CASE tool or standard C. This ability to combine the shell with products of all other existing development tools results in a uniquely general 4GL.

Many also complain that fourth-generation languages sacrifice too much efficiency for the sake of short source code and high-speed development. The ability to use operators from any source is an answer to this complaint as well. It allows a shell programmer to code speed-critical routines in a language more suited to efficiency considerations. If an application requires floating-point number crunching, one codes the appropriate routines in FORTRAN and the non—time-critical sections of the code can still be done in the high-level shell code.

With the shell, development is quite easy for even the novice programmer. The interpreted environment allows easy access to the internals of the script as it runs, as well as a fast test-change-test cycle. The flat ASCII data format and lack of operator side-effects make it easy to examine the effect different operators have on the stream of data.

Shell Operators

The shell relies heavily on its operators. For example, it has no built-in ability to evaluate expressions; instead, it uses the **test** utility to to that.

The "string" data type is the only one the shell supports. The shell assumes that there are operators which will do any more advanced data type a programmer might need. Operators exist to perform numerical functions. For multi-field records, operators commonly use the space or tab character as a field separator and the newline character as a record terminator. This allows great flexibility, despite the overhead incurred of converting data into and out of ASCII for non-string operations.

One of the greatest strengths of the shell is the ability to process an entire file with a single command. The shell does allow for defining procedures, as well as execution control constructs like **if**, **while**, and **case.** However, these flow-control constructs are often not needed. For example, the script presented in the following section explicitly performs no looping, because the operators implicitly loop, acting on each line of the program.

Compatibility with MS-DOS

MS-DOS shares key underlying features with COHERENT, enough so that the operator/stream paradigm can be utilized identically in both environments. Except for minor limitations on file name syntax, the MS-DOS hierarchically structured file system appears to the user to be functionally identical to the COHERENT file system. The multi-tasking capabilities of COHERENT, while missing from DOS, are not essential elements of the paradigm. Although DOS shells use intermediate temporary files to implement pipes, the interface presented to users, even using **command.com**, can be described with the operator/stream terminology we use here. The COHERENT shell and **awk** are available as MS-DOS shell replacements and enhancements.

Conclusion

The shell is quite a powerful tool indeed. It is not without limitations, however. First, only a few companies are currently producing tools oriented towards use in shell scripts. **/rdb** remedies the shell's weakness of not being able to store complex data types, and there are many additional tools for for correcting some of the other major limitations, such as numerical computation, statistical analysis, and business graphics output. Although COHERENT has applications dedicated to mathematics and numerical analysis, most are themselves large self-contained programs. A programmer who needed matrix inversions, for example, must adapt existing tools, like System S or SAS, to work within shell scripts, or write a special purpose tool.

The shell needs improvement in the ability to connect multiple pipes together more freely. The original designers didn't anticipate the need for more than linear pipelines. Although more complex, non-linear pipes can be created by the use of temporary files, this method is only barely adequate

TUTORIALS

for constructing complex structures such as cyclic streams. Finally, more operators are needed that allow incoming data to be split between or duplicated on multiple output pipes.

The shell shares another problem with weakly typed languages: errors in the format of that data stream can lead to unexpected output. Because no method of type- or format-checking is built into the shell, the programmer must write code that avoids the problem. The interpreted environment does allow careful examination of the stream data as it passes through each operator, which reduces the difficulty of writing error free code. The shell won't lend itself to the sort of correctness proofs offered by the newer CASE tools unless a formal definition is proffered, specifying not only syntax but all the operators as well.

The operator/stream paradigm has produced a simple, powerful, general purpose tool. It allows one to prototype or generate a proof of concept in hours or days, when it might have required weeks with C. Although the shell produces slower code than a third-generation language, the increasing power of modern computers makes this a minor concern for many tasks. This framework provides an easily visualizable way of manipulating large (or small) amounts of structured data.

Although there is currently a shortage of utilities designed for general-purpose use within shell scripts, awareness of the potential of shell programming is spreading, and more packages are being written outside of the traditional monolithic program tradition. In this way, computing is coming full circle, returning to the original concepts of Von Neumann, whose computing paradigm embodies the stream of sequential memory passing by the operator of the central processing unit.

TUTORIALS

Relational Data Bases

The central idea of relational data bases is that all information can be put into tables and can be manipulated by a few simple operations. This is a revolutionary idea and takes some thinking to understand. It means that all of our information needs can be handled in a simple way. All information, data, knowledge, facts, or whatever you call it, can be put into tables. For example, a table named **expenses** might look like this:

```
Date    Amount  Account  Description
------  ------  -------  ----------------------------
850125   67.00     4120  plane ticket
850126   12.00     4120  meal with client
850127  150.00     4120  hotel bill
```

At the top of the table are the column names. The dashed line makes it easier to read, and separates the header from the data rows. Each row of the table holds information about a particular expense item.

We see information in tables every day — train schedules, scientific data, phone books, restaurant menus, price lists, catalogs, math tables, conversion tables, accounting and financial reports, tax forms, and so on. We all understand tables: we know how to fill them in and how to look up information in them. Tables and our operations on tables are independent of the information they contain. The purpose of this book is to help you to think about your information problems in terms of tables and operations on tables.

In accounting, information is collected in *journals*, then manipulated to create financial statements that you can review to see the state of the business. Scientists have instruments in their labs that collect data and store it in their computers. Statistical programs compress masses of data into simple tables and graphs for the scientist to look at to see responses of systems to various inputs and to discover causal relationships. Employee information is stored in tables and can be extracted to print pay checks, make phone books, and so on.

Files

When we take these abstract ideas to the COHERENT system, we find a friendly environment for storing tables and manipulating them. Computers store tables in files that live on hard disks. They have names by which they can be called up. Tables can be created with a text editor, with COHERENT commands, from other tables, or imported from other COHERENT data base systems, or even foreign systems. Once tables are implemented as ordinary COHERENT files, there is available a plethora of COHERENT utilities for maintaining security, data integrity, and ancillary information about tables.

There are only three rules for creating a table in the COHERENT and **/rdb** environment:

- A tab is inserted between each column.
- On the first line of the table, each column is given a name.
- The second line of the table has lines of dashes for each column, showing the width of the column.

This is much easier for the user than systems that require the overhead of setting up schema and special files. Once tables are created, they can be manipulated by COHERENT and **/rdb**. Using the **ve** forms editor to initially create tables and enter data from your terminal avoids potential problems caused by (among other things) the wrong number of tabs.

Columns

When we try to look up information in a table, we usually find that there is a lot more information there than we want. We have to find the single fact we want from all the rest. Often, there are more columns than we need. Therefore, we want to reduce the table to only those columns we need. When we look at a table with our eyes, we try to ignore the extra columns. Or we use our fingers to scan down the column we want.

When using the computer we have a command called **column**, which is a program that inputs one table and outputs another table that has only the columns you specify. For example consider our **expenses** table. One can project some of the columns of a table by typing this simple command, which says to project columns **Amount** and **Description** from the input **expenses** table:

```
column Amount
```

This produces:

```
Amount  Description
------  ---------------------------
 67.00  plane ticket
 12.00  meal with client
150.00  hotel bill
```

Note that now we have a new, smaller table with only the two columns we want to see. For another example, you might project just the names and phone numbers from the employee table to make a phone book. In other words, the company phone book is a projection of the name and phone columns of the employee table.

Rows

The most common operation is to search the table for a specific piece of information. We want to know when the next plane leaves, the price of the scallops on the menu, the degrees Centigrade for 72˚ Fahrenheit, the ball player with the highest batting average, the grade for a class on a report card, the amount spent on phone calls by the company last year, and so on. Each search requires that we select just one, or a few, rows from a table.

On the computer, you can select only the rows desired with the **row** command. Here we select all rows in the **expenses** table in which **Amount** is greater than 50:

```
row 'Amount > 50' < expenses
```

This produces:

```
Date    Amount  Account  Description
------  ------  -------  ---------------------------
850125   67.00     4120  plane ticket
850127  150.00     4120  hotel bill
```

With one simple operation you can select information from any table containing any data.

Columns and Rows

Often you want to find one or a few rows and see only a few columns. On COHERENT you can put these two commands together. The COHERENT pipe symbol '|' means to take the table that is output from the **column** command on the left and input it into the **row** command on the right:

```
column Amount Description < expenses | row 'Amount > 50'
```

This produces:

TUTORIALS

```
Amount  Description
------  --------------------------
 67.00  plane ticket
150.00  hotel bill
```

Since the output of one program can be piped into the input of another, you can create report programs with one or a few lines of commands and pipes. For example, you can project some columns of a table with the **column** command, pipe the output to a **row** program which will select certain rows, pipe that output to a **sorttable** program which sorts it, then on to a **jointable** to pick up other columns from another table. All of this can be done by typing one line on your terminal. Or this line can be typed into a shell program file with a text editor, and given a name, like **weeklyreport**. Then, by simply typing **weeklyreport**, the report can be produced when you wish. Several reports can be invoked by a higher level shell script. This is called *shell programming*.

These reports, and other programs, can be placed in a menu and selected with a few keystrokes, or if you have a terminal with a mouse, by pointing and clicking. Therefore, you can automate whole operations in hours and days, rather than the months and years that it takes on other systems.

Shell Programming and Flat ASCII Files

The key to this approach is shell programming and flat ASCII files. *Flat* means there is no structure to them. COHERENT thinks of them as a stream of characters. ASCII is a standard code for converting characters to numbers for storage in the computer. The files have variable length records and fields. This is a break from the IBM tradition of structured binary files with fixed length records and fields. Breaking with this old tradition is an essential part of the COHERENT revolution.

This does not mean that **/rdb** cannot manipulate binary data. We will discuss manipulating arbitrary binary objects later.

Lists

In addition to the table format shown previously, **/rdb** also has a list format for data that is hard to fit into a table, such as mailing lists. For example:

```
Number 1
Name   Rambo
Company      One Man Army
Street 123 Mac Attack
City   Anywhere
State  Thirdworld
ZIP    30000
Phone  (111) 222-3333

Number 2
Name   Chuck
Company      ...
```

Note that the list format begins with a blank line, unlike the table format, and that a single tab separates the field name from the data for each field. You can use the COHERENT **vi** text editor or **/rdb**'s **ve** form editor to initially create tables, and to enter and edit data in a table. You can convert from table to list and from list to table format easily with the **tabletolist** and **listtotable** commands.

Review

A relational data base is a collection of relations or *tables* that may be related on one or more common columns. Relational data bases implemented like this are easily transportable from one environment to another.

Relational data bases have a solid mathematical base in relational set theory, relational algebra, and relational calculus. There are theorems in this relational math that prove that any data put into a relational data base can be extracted. The mathematical base also assures that manipulations performed will have correct results, just as arithmetic assures us that the math functions we perform on the computer have correct results.

Normal Relations

An **/rdb** relation, or table, is an ordinary ASCII file. An **/rdb** table has rows, or records, separated by newlines. It has fields, or columns, separated by a tab character. Every row must have the same number of columns. This is *first-degree normalization*.

The first row of a table contains the names of the columns; the second row contains columns of dashes. Any kind of information can be represented in such a table: numbers, words, file names, etc: **/rdb** commands and relational set theory doesn't care about the content of the table — just as long as these rules are followed for the form of tables. Another important rule to remember when designing a data base is:

If many columns are used in a single row to describe the same type of information, it's time to make a new table.

For example, consider a table of family members:

```
id     mom    dad    kid1   kid2
--     ----   ---    ----   ----
1      mary   jack   billy  bobby
2      nancy  joe    terry  susie
3      sally  john   adam
```

In this example there are two *kid* columns in each row. The right way to express this relationship is with two tables: one for parents and one for kids. They are *related* or linked by a common column, *id*.

```
id     mom    dad
--     ---    ---
1      mary   jack
2      nancy  joe
3      sally  john


id     kid
--     ---
1      billy
1      bobby
2      terry
2      susie
3      adam
```

How Is Information Accessed?

Tables are accessed through **/rdb** and shell commands issued at the COHERENT prompt or from within shell or C programs. Here is a list of some common **/rdb** commands which are used or mentioned in these examples.

Selected /rdb Commands

/rdb *Command* *Description*
column, project .....Selectonly certain columns
row, select.....Selectonly certain rows
mean.....Computethe mean of selected columns

TUTORIALS

jointable.....Jointwo tables
sorttable.....Sorta table
compute.....Docalculations on columns
subtotal.....Subtotalselected columns
total.....Totalselected columns
rename.....Changethe name of a column
justify.....Makea table line up properly
headoff.....Removethe first two header rows
report.....Reportwriter
ve.....**vi**-liketable editor

/rdb commands are programs that read tables from the standard input and write tables to the standard output. Suppose there's a table that looks like this:

```
Item   Amount Cost   Value  Description
----   ------ ----   -----  --------------
1      3      50     150    rubber gloves
2      100    5      500    test tubes
3      5      80     400    clamps
4      23     19     437    plates
5      99     24     2376   cleaning cloth
6      89     147    13083  bunsen burners
7      5      175    875    scales
```

Then a sample query might be:

```
column Item Cost Amount < inventory
```

This produces:

```
Item   Cost   Amount
----   ----   ------
1      50     3
2      5      100
3      80     5
4      19     23
5      24     99
6      147    89
7      175    5
```

This is read aloud as: *''Select the Item, Cost, and Amount columns from the inventory table.''* It's important to voice queries because people often type stuff in that they would never say out loud.

The query:

```
row 'Cost > 50' < inventory
```

This produces:

```
Item   Amount Cost   Value  Description
----   ------ ----   -----  --------------
3      5      80     500    clamps
6      89     147    16353.8       bunsen burners
7      5      175    1093.75       scales
```

This reads, *''Select rows where the Cost column is greater than 50 from the inventory table.''* To put commands together:

```
column Item Cost Value < inventory | row 'Cost > 50'
```

This produces:

```
Item   Cost   Value
----   ----   -----
3      80     400
6      147    13083
7      175    875
```

This is pronounced, *"Select the Item, Cost and Value columns from the inventory file and select those rows where Cost is greater than 50."* Inside the apostrophes the **<** and **>** symbols are pronounced *less than* and *greater than* respectively, whereas outside apostrophes they are pronounced *from* and *to*. The pipe symbol '|' is pronounced *and.*

To take the mean of the result while listing each line:

```
column Item Cost Value < inventory | row 'Cost > 50' | mean -l Value
```

This produces:

```
Item   Cost   Value
----   ----   -----
3      80     400
6      147    13083
7      175    875
----   ----   -----
                     4786
```

Creating Tables and Entering Data

There are many different ways to create tables; editors, programs, shell commands such as **sed** or **awk**, etc. Most often, however, when **/rdb** tables are entered from scratch, **ve**, the **/rdb** table editor is used. **ve** allows the creation of tables quickly and in a familiar and easy way. It's a lot like **vi**. The first step in the creation of a table with **ve** is to create a screen definition file with any editor. This can be accomplished with any editor. Here is a 'screen' file for the **states** file:

```
             The States File
st      < st >
state   <  state    >
```

ve uses this screen file to create the table. The rules for screen files are simple:

1. Column names go inside the angle brackets.
2. Anything outside of angle brackets is just text that appears on the screen.

The space between angle brackets is the viewable window over the field, and isn't a restriction on how wide the field can really be. After creating a screen file like this:

```
ve states
```

and the **states** file will be created. Let's say **ve** has been used to add new records to our *states* table so that it looks like this:

```
st     state
--     -----
CA     California
NV     Nevada
NY     New York
```

A mailing list can be created the same way, by making a 'screen' file and then using **ve** to add a few rows:

TUTORIALS

```
      Yet Another Mailing List

Name         <name              >
Street       <address           >
City         <city              >
State        <st>
```

The command **justify** adjusts the mailing list, to make it appear neater. For example, the command

```
      justify < mlist
```

produces:

```
name   address      city   st
----   -------      ----   --
Evan   Main St.     Santa Cruz    CA
Rod    Broadway     Ithaca NY
```

The next command

```
      column st name < mlist |
      sorttable |
      jointable - states
```

reads, "Select the st and name columns from mlist and join it with the states table." It produces the following:

```
st     name   state
--     ----   -----
CA     Evan   California
NY     Rod    New York
```

The **sorttable** command was silent. But it has to be there. Both files to be joined must be sorted. The **states** file is already sorted. The dash in the **jointable** command means use the standard input, just like the COHERENT **join** command.

Reports

For numeric information, **/rdb**'s standard table output adjusted with a **justify** or **trim** command is often sufficient, especially when combined with **tabletotbl** and the UNIX **tbl** and the COHERENT **nroff/troff** formatters. In addition to these methods, **/rdb** has a **report** command that uses a prototype report form and has built-in command processing capabilities.

Let's look at a sample report form. It's like the screen file for **ve**: text is outside brackets, and column names are inside brackets. Other commands can also go inside the brackets. Here's a report form for the *mlist* file:

```
<name>
<address>
< city >, <st>
                    <! date +%D !>
Dear < name >,
This is a computer chain letter.
I am also sending it to:

<!  column name city < mlist |
row 'name != "<name>"' | justify  !>
Bye, <! echo <name> !>.
```

Thus, the command

```
      row 'name ~ /Evan/' < mlist | report mlist.form
```

produces the following output:

```
Evan
Main St.
Santa Cruz, CA
                    09/03/89
Dear Evan:
This is a computer chain letter.
I am also sending it to:

name   city
----   ------
Rod     Ithaca

Bye, Evan.
```

Arbitrary text goes outside the angle brackets; column names go inside angle brackets, and *any* arbitrary command or shell program or shell command(s) can go between exclamation marks within angle brackets, and you can *still* specify columns from the current record therein. You can even have reports within reports.

The Big Text Field Problem

The "bug report," "long text column," and "every word indexed" problems are all facets of the same situation. Let's say a file has some relatively short columns, and one or more long text columns on which you'd like to use **vi**.

Take the case of a bug report data base with associated arbitrarily long narrative descriptions: a solution is to keep the descriptions in a sub-directory called, for example, **bugreports**, one file per record, with the file name being **bugreports/recid** where *recid* is the record identifier from the current record. Then, a control key is mapped in the **.verc** file (analogous to the **vi**'s **.exrc** file) to the command **vi bugreports/<recid>**. This grabs an identifying column from the record, constructs the name of the associated file(s), and pops the user into **vi** on the named file(s). This is quite flexible even if there is more than one file associated with each record, switching between **ve** files with a keystroke, thus effecting multiple screens: map a control-key to write the record and switch the files, and another to switch back.

A simple report makes a two column table with record id and word for each word in each narrative, allowing for queries like *Give me all the bugs mentioning word 'xyzzy':*

```
#!/bin/sh
(echo "word     id"
echo  "----      --"
for i in [0-9]*
do
       word < $i | awk '{print $1,"'$i'"}'
done) |
sorttable -u
```

Now records having a particular word can be found easily. If speed is a consideration, build an(other) inverted index on the word/id concordance list just created:

```
cd bugreports
wordy > bugwords
index -mi -x bugwords word
echo xyzzy | search -mi -x bugwords word
```

This produces a list of record ID numbers on the standard output. Once you have the record ID numbers, one more search is necessary to find the original record in the **bugs** table. Of course, with the record ID numbers, *no* search is necessary to find the narrative, because the file name *is* the record id.

Non-Text Data Structures

TUTORIALS

Suppose a field is a picture, or a sound, or some other non-textual object. The **/rdb** approach is to identify an object resource, with text, within a field in a table, describing the type and location of the object. Fields from the current record can be referenced in the **.verc** file by the same <*column_name*> specification used in the report program and customized **ve** screen files.

Large Tables

Large tables are often as easily handled as small tables. When working with very large tables some form of indexing is desirable: hash, inverted sequential secondary, binary (sorted relations), or some form of tree (linked list).

The shell approach is to use the COHERENT directory structure as the first (few) levels of tree index. One financial application using **/rdb** involves the 80-megabyte file of time series from the International Monetary Fund. As distributed, it takes several large machine CPU minutes to peruse this big file and extract a single time series. The file was divided into a directory for each country and within each country directory, a file for each time series, with the file name being the time series code as given by the IMF. Each of those time series files is a **/rdb** file, with columns **YR ANN Q1 Q2 Q3 Q4** and so forth. A separate "description" file in each country directory has a line for each file in the directory, giving **CODE DESCRIPTION UNITS**.

Thus, the time to retrieve any time series (if the country and time series code are known) is *independent of the size of the data base.* Queries like "*which countries have this time series in common?*" are answered with the **ls** command. More than one level of index can of course be implemented just by adding directory hierarchies.

COHERENT has many commands to traverse directory trees, and to add, delete, and otherwise manipulate nodes. With this approach, nodes are tables, and the plethora of COHERENT directory and file handling commands are all relational data base manipulation commands.

Architectural Performance Enhancements

Learning to use the COHERENT utilities has a much greater value than learning yet another special programming language. Once a small critical mass of COHERENT familiarity is achieved, application development becomes little more than writing simple yet powerful scripts to perform tasks which used to be laboriously performed by hand, or just not done at all.

All these techniques comprise a marriage of the facilities that come with the COHERENT system itself, and relational capabilities provided by **/rdb**.

This attitude of not reinventing the wheel is the basis of the shell and **/rdb** approach. All COHERENT knowledge is knowledge about data bases, and experiences with data bases teach more about COHERENT. That's why the combination of the **/rdb** extensions to COHERENT and the shell command language is a 4GL most appropriate to the COHERENT environment.

SQL

SQL is another language for querying a data base. It's used as the foundation for many contemporary 4GLs. It does not use the stream/operator paradigm, but "nests" queries to pass data from one operator to another. When SQL was developed, COHERENT had not yet been invented, so an entire environment had to be developed to express queries. SQL is another system to learn, with little use outside of itself, and typically no relation to the operating environment surrounding it.

SQL does not specify any particular file format. Although there is an ANSI standard SQL for expressing queries, implementors are free to store data however they want. In a way, this is a contradiction, because getting away from these walls that stand between data is very important, and was the principal reason that the concept of a data base came about. The idea goes under the name of *integrated* and *modeless* software, and most recently, *interoperability*.

There are reasons why SQL-based systems are popular, even desirable. SQL-based systems are widely available and there is a large body of expertise also readily available. Many U.S.-government agencies require access to corporate data bases via SQL, especially in the defense industry. SQL is valuable in non-COHERENT environments. Partly because SQL is difficult for novices to understand and use, SQL providers typically field a large, helpful support organization. Of course, this drives the price up, and doesn't adequately address the needs that prompted the development of these tools in the first place: making non-experts proficient and productive in the construction of basic data base applications.

SQL queries can be converted to shell scripts:

SQL	*COHERENT and* **/rdb**
`select col1 col2 from filename`	`column col1 col2 < filename`
`where column - expression`	`row 'column == expression'`
`compute column = expression`	`compute 'column = expression'`
`group by`	`subtotal`
`having`	`row`
`order by column`	`sorttable column`
`unique`	`uniq`
`count`	`wc -l`
`outer join`	`jointable -a1`
`update`	`delete, replace`
`nesting`	`pipes`

Fifth-Generation Systems

There is a distinction to be made between fourth-generation languages and CASE tools. The programmer of a fourth-generation system must still specify the fundamental algorithms for completing a task, perhaps at a higher level of abstraction, especially of data types. CASE tools, on the other hand, require only a specification of the task, and generate not only the code, but also the algorithm. CASE tools tend to be of very limited domain. An example would be a screen layout tool. The developer draws the positions of the windows wanted, and the tool generates the code to create the windows, manage the text and graphics inside them, and deal with icons and menus.

Some graphical CASE tools are examples of what we might call 5GLs. Using a graphical user interface, these tools allow applications to be built by example. Some force the user to specify actions algorithmically, and some do not. There's even less agreement about what constitutes a 5GL than there is about 4GLs.

The operator/stream paradigm has produced a simple, powerful, general purpose tool. It allows one to prototype or generate a proof of concept in hours or days, when it might have required weeks with C. Although the shell produces slower code than a third-generation language, the increasing power of modern computers makes this a minor concern for many tasks. This framework provides an easily visualizable way of manipulating large (or small) amounts of structured data.

Although there is currently a shortage of utilities designed for general purpose use within shell scripts, awareness of the potential of shell programming is spreading, and more packages are being written outside of the traditional monolithic program tradition. In this way, computing is coming full circle, returning to the original concepts of Von Neumann, whose computing paradigm embodies the stream of sequential memory passing by the operator of the central processing unit.

References

Kernighan B., Pike R.: *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice Hall, 1985.

TUTORIALS

Kochan S., Wood P.: *UNIX Shell Programming*. Hayden Book Company, 1985.

Prata S.: *Advanced UNIX — A Programmers Guide*. Indianapolis: Howard W. Sams and Co., Inc., 1985.

A. Winston: "4GL Faceoff: A look at Fourth-Generation Languages," *UNIX/World*, July 1986, pp. 34-41.

Misra S., Jalics P.: "Third-Generation versus Fourth-Generation Software Development," *IEEE Software*, July 1988, pp. 8-14.

Manis R., Schaffer E., Jorgensen R.: *UNIX Relational Database Management*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Verner J., Tate G.: "Estimating Size and Effort in Fourth-Generation Development," *IEEE Software*, July 1988, pp 15-22.

Matos V., Jalics P.: "An Experimental Analysis Of The Performance Of Fourth Generation Tools On PCs," *Communications of the ACM*, November 1989, pp. 1340-1351.

Manis R., Meyer M.: *UNIX Shell Programming*. Indianapolis: Howard W. Sams Inc., 1987.

TUTORIALS

Commands, I/O, Pipes

/rdb commands are programs that read tables from the standard input and write tables to the standard output. Once data are entered into tables, they are accessed through **/rdb** and shell commands issued at the COHERENT prompt or from within shell or C programs. This is an easy way to ask questions of, or *query*, the data base. Here is a list of some common **/rdb** commands which are used or mentioned in the following example queries:

/rdb *command*	*Description*
column, project	Select only certain columns
row, select	Select only certain rows
mean	Compute the mean of selected columns
jointable	Join two tables
sorttable	Sort a table
compute	Do calculations on columns
subtotal	Subtotal selected columns
total	Total selected columns
rename	Change the name of a column
justify	Make a table line up properly
headoff	Remove the first two header rows
report	Report writer

Suppose there's a table that looks like this:

```
Item     Amount     Cost     Value     Description
----     ------     ----     -----     -------------
   1          3       50       150     rubber gloves
   2        100        5       500     test tubes
   3          5       80       400     clamps
   4         23       19       437     plates
   5         99       24      2376     cleaning cloth
   6         89      147     13083     bunsen burners
   7          5      175       875     scales
```

Then a sample query might be:

```
column Item Cost Amount < inventory
```

Which produces:

```
Item     Cost     Amount
----     ----     ------
   1       50          3
   2        5        100
   3       80          5
   4       19         23
   5       24         99
   6      147         89
   7      175          5
```

This is read aloud as: *"Select the* **Item**, **Cost**, *and* **Amount** *columns from the inventory table."*

In another example, the command

```
row 'Cost > 50' < inventory
```

This is, *"Select rows where the Cost column is greater than 50 from the inventory table."* This yields:

```
Item     Amount     Cost     Value     Description
----     ------     ----     -----     -------------
   3          5       80       400     clamps
   6         89      147     13083     bunsen burners
   7          5      175       875     scales
```

The next example puts the commands together:

```
column Item Cost Value < inventory | row 'Cost > 50'
```

This is pronounced, *"Select the Item, Cost and Value columns from the inventory file and select those rows where Cost is greater than 50."* This yields:

```
Item     Cost     Amount
----     ----     ------
   3       80          5
   6      147         89
   7      175          5
```

Between the apostrophes, the **<** and **>** symbols are pronounced "less than" and "greater than" respectively, whereas outside apostrophes they are pronounced "from" and "to". The pipe symbol '|'symbol is pronounced "and".

To take the mean of the result while listing each line, do the following:

```
column Item Cost Value < inventory | row 'Cost > 50' | mean -l Value
```

This yields:

```
Item     Cost     Amount
----     ----     ------
   3       80          5
   6      147         89
   7      175          5
----     ----     ------
                    4786
```

You can use the name **select** instead of **row**. The Korn shell has a **select** command, and **/rdb** uses **row** as a synonym for **select**. We give the same program two names by using the COHERENT link command **ln**.

compute: Compute a New Result

The command **compute** lets you calculate one column as a function of other columns, or constants. For example, you could calculate the **Value** column in your **inventory** table as **Onhand** times **Cost** with this command:

```
compute 'Value = Onhand * Cost' < inventory
```

This yields

```
Item      Onhand     Cost     Value     Description
----     -------     ----     -----     ---------------
   1           3        5        15     rubber gloves
   2         100        1       100     test tubes
   3           5        8        40     clamps
   4          23        2        46     plates
   5          99        3       297     cleaning cloth
   6          89       18      1602     bunsen burners
   7           5      175       875     scales
```

For a fancier example, add $5 to the **Cost** of item 4 only, and recompute the **Value** column:

TUTORIALS

```
compute 'if (Item == 4) \
        Cost += 5; Value = Onhand * Cost' < inventory
```

This yields:

```
Item     Onhand     Cost     Value     Description
----     ------     ----     -----     ---------------
   1          3        5        15     rubber gloves
   2        100        1       100     test tubes
   3          5        8        40     clamps
   4         23        7       161     plates
   5         99        3       297     cleaning cloth
   6         89       18      1602     bunsen burners
   7          5      175       875     scales
```

Note that the **Cost** of item 4 is now 7 (2+5). You can also write commands down the page. From the line on which you open the apostrophe until you enter the closing apostrophe, the shell will keep reading lines. You can do lots of powerful calculations with **compute**. See the **/rdb** manual pages for **compute**, and read the **awk** tutorial in your COHERENT documentation.

justify: Align Columns

Often when you type in a table, or for other reasons, the columns containing numbers might be left justified. For example:

```
Item     Onhand     Cost     Value     Description
----     ------     ----     -----     ---------------
1        3          5        15        rubber gloves
2        100        1        100       test tubes
3        5          8        40        clamps
4        23         7        161       plates
5        99         3        297       cleaning cloth
6        89         18       1602      bunsen burners
7        5          175      875       scales
```

In this **inventory** table, you may not like looking at number columns left justified. It is not the way we usually see numbers in tables. But it is an easy way to enter data into a table with a text editor because we do not have to space over to line up our numbers. To make our table a lot prettier, the **justify** command is available. It right-justifies numbers and left-justifies strings. It also blank fills the character columns, so that they can be placed in any order and the columns to their right will still line up.

For example, it might be easier to enter the **inventory** table as shown previously. Then the **justify** command can be used to right justify the numerical columns:

```
justify Onhand Value Cost < inventory
```

This yields:

```
Item     Onhand     Cost     Value     Description
----     ------     ----     -----     ---------------
1             3        5        15     rubber gloves
2           100        1       100     test tubes
3             5        8        40     clamps
4            23        7       161     plates
5            99        3       297     cleaning cloth
6            89       18      1602     bunsen burners
7             5      175       875     scales
```

Only the columns selected were justified, including their column headings. **Item** was not selected and was not justified. Now if we project the columns in **inventory** in a different order, the columns will line up correctly:

```
column Description Item Onhand Cost Value < inventory
```

which yields:

```
Description          Item     Onhand     Cost     Value
---------------      ----     ------     ----     -----
rubber gloves           1          3        5        15
test tubes              2        100        1       100
clamps                  3          5        8        40
plates                  4         23        2        51
cleaning cloth          5         99        3       297
bunsen burners          6         89       18      1602
scales                  7          5      175       875
```

total: Compute Arithmetic Totals

To total columns of a table simply type:

```
total < inventory
```

This yields:

```
Item     Onhand     Cost     Value     Description
-----    ------     ----     -----     ---------------
28       324        212      2980
```

Note that we have totaled the **Item**, **Onhand**, and **Cost** columns. Summing these columns does not make much sense, but **total,** in this automatic or default mode, operates on all numeric columns. Naming specific columns will result in your getting only those columns totaled. Only numeric columns can be totaled.

The **-l** option lets you see the whole table. (The **l** in **-l** is the letter "el", not the number one.) For example, the command

```
total -l Value < inventory
```

yields:

```
Item      Onhand      Cost      Value      Description
----      -------     ----      -----      ---------------
1         3           5         15         rubber gloves
2         100         1         100        test tubes
3         5           8         40         clamps
4         23          2         51         plates
5         99          3         297        cleaning cloth
6         89          18        1602       bunsen burners
7         5           175       875        scales
-------   -------     ----      -----      ---------------
                                2980
```

subtotal: Compute a Subtotal

To subtotal, enter the break column first, followed by the columns to be subtotaled. The *break column* is the column watched by the program. If, on a new row, the value in the break column is the same as the previous value in that break column, the values in the other columns are added. But if the value in the break column changes (breaks), then **subtotal** draws a line and prints the subtotals of the other columns. As long as the value in the break column remains the same row after row, the values in the selected columns are accumulated. When the break column's value changes, the subtotal is printed and a new subtotal is started.

TUTORIALS

For example, we might have the following **ledger** table:

```
Account       Date     Debit     Credit
-------     ------     -----     ------
101         820102     25000
101         820103                 5000
101         820104                15000
130         820104     30000
150.1       820103     10000
211.1       820104                15000
211.1       820102                25000
211.1       820103                 5000
```

In this example, **Account** is the break column and **Debit** and **Credit** are to be summed. (This is done to *foot* the ledger at the end of an accounting period.)

```
subtotal -l Account Debit Credit < ledger
```

This yields:

```
Account     Date     Debit    Credit
-------     ------   -----    ------
101.0       820102   25000
101.0       820103              5000
101.0       820104             15000
-------     ------   -----     -----
101.0                25000     20000

130.0       820104   30000
-------     ------   -----     -----
130.0                30000         0

150.1       820103   10000
-------     ------   -----     -----
150.1                10000         0

211.1       820104              15000
211.1       820102              25000
211.1       820103               5000
-------     ------   -----     -----
211.1                    0     45000
```

Columns **Debit** and **Credit** are named on the command line, so only they are totaled, and not column **Date**, which would be pointless to total. Note that the value in the break column is carried down to identify the subtotal. The **Account** values are not subtotaled. That would also be meaningless.

The **-l** tells **subtotal** to output the whole table, not just the subtotals. Without the **-l,** we would get only the subtotals. For example,

```
subtotal Account Debit Credit < ledger
```

yields:

```
Account     Date   Debit     Credit
-------     ----   -----     ------
101.0              25000      20000
130.0              30000          0
150.1              10000          0
211.1                  0      45000
```

Quoting Commands

Note that the apostrophe (') goes before and after the conditional expression in the **row** command. This protects the expression from the COHERENT shell. The shell is the program within the COHERENT system that reads the commands you type and executes them. Without the apostrophe, the shell would see the left arrow (**<**) and think we were inputting a table named **10**. The apostrophe is also needed to put the whole expression together as one argument for the **row** command. If you need to quote something within the expression, use quotation marks ("). For example, if you had a column name with special characters (including spaces) in it, you must put it in quotation marks. For example:

```
row '"Net Pay" == "Gross-Pay"' < payroll
```

A column name consists of upper- and lower-case letters, numbers, and the underscore character '_'. Therefore, if you put special characters and spaces in column names, you will have to keep putting the quotation marks around them every time you refer to them. (This is true of C language versions of **column** and **row.** It is safest to stick with one-word column names).

Note that we use a double equals '==' instead of just one '='. This comes from the C language and the **awk** program, that the **row** and **compute** commands use. The rule is to use the '=' to mean, *"Assign the value on the right of the equal sign, to the variable on the left."* Use the '==' to mean *logically equal,* as we used it in the **row** command. Another way to remember this, is that you usually use the logical equal '==' in **row** commands, and the assignment equal '=' in **compute** commands. In advanced usage, you will find that you need both in **row** and **compute** for very powerful commands.

Pipes

Pipes '|' are a COHERENT shell mechanism that sends the output of one program to the input of another program. This allows you to perform a series of operations on a table. You now can begin to see the real power of this system. In a single line we can produce a report. If you wished to select only rows where **Value** was greater than 100, but wanted to see only the **Description** and **Value** columns sorted in alphabetical order, you could pipe the **row** command's output into the **column** command, which would be piped into the **sorttable** command like this:

```
row 'Value > 100' <inventory |
column Description Value |
sorttable
```

This yields:

```
Description           Value
---------------       -----
bunsen burners         1602
cleaning cloth          297
scales                  875
```

The pipe symbol '|' means that the output from the program to the left becomes the input to the program on the right, or below. As with quotes, the shell will keep accepting lines ending with pipes. Therefore, you can write long pipe programs down the page, which is easier to read and debug. The **sorttable** program, without any columns named, defaults to sorting on the first and subsequent columns.

Warning: Never use the same file as input and output in one command or pipe. If you want to save the output that normally comes to your terminal screen in a file, use **> tmp**. The **tmp** file name is a common name for a file that has only temporary use. It should always be ok to remove it later. For example:

```
sorttable < inventory > tmp
```

TUTORIALS

You can then move **tmp** to **inventory**:

```
mv tmp inventory
```

Now your **inventory** table is sorted. If you type:

```
sorttable < inventory > inventory
```

your **inventory** table will be zeroed out (have zero characters or lines in it). The reason is that the COHERENT shell, in opening the inventory file for output, zeros the file's size. Then, when the **sorttable** program (or any COHERENT program) reads it in, it gets EOF on the first read. It closes the file, empty! You'll get the same disastrous effect even if you pipe the output through several programs and then try to write it into the original input file. Do not do it! Also, get in the habit of looking at your **tmp** file before moving it onto your original good data. You may have made an error and be moving or copying bad data onto good. Finally, be careful not to type

```
> file
```

when you mean:

```
< file
```

The first wipes out the file, rather than reading it.

Syntax: How To Enter Commands Correctly

In the manual, you will see a synopsis of each command and its syntax. *Syntax* is the rules for forming a correct command. *Semantics* is what the resulting command will do. Computer programmers are familiar with this way of expressing the syntax rules. If you are not, read this chapter. We will start with simple syntax and go to complex. The simplest syntax is a command with no options:

```
commands
```

You just type **commands** and you get all of the commands' descriptions and their syntax rules. Most commands require an input table. For example:

```
column < table
```

Substitute the name of your table for the *table* in the syntax rule. But you type the **column <** exactly as specified. If you have a table named **inventory** you would type:

```
column < inventory
```

Many commands allow you to input **tableorlist**. This means they can handle both table and list formatted files:

```
column < tableorlist
```

Options are placed in brackets '[ ]' because you do not have to type them unless you want them. For example:

```
justify [ column ... ] < table
```

The brackets here mean that you can optionally name columns to be justified. The ellipsis '...' means that you can have many column names. But since there are brackets around the *column ...*, you can also leave them out. In which case, all the columns would be justified. Note that there are no brackets around the **< table**. This part is not optional. You must have it to tell the COHERENT shell which file to open for input.

You should know that whenever you see **< table** you can also use a pipe to input a table into the command:

```
cat table | justify
```

This has the same effect as:

```
justify < table
```

If you see a vertical bar '|' within brackets, it means *or*, not a pipe. Other commands give lists of options in brackets. For example:

```
jointable [ -a[n] -n - ] [ -j[n] column ] table1 [ table2 ]
```

This means that each option may or may not be selected. Note that there are brackets within brackets. This means that if you chose the **-a**, for example, you have the further option of adding a number **n** to specify which table number. The brackets around the **-j***column* mean that the two strings go together, as well as being optional. The *column* should be replaced by the name of one of the columns in one of the tables. Finally, the second table is in brackets, meaning it is optional. But the manual page goes on to explain that if it is not there, and the **-** option is not chosen, then the **jointable** command will get its second table from the standard in.

The most complex syntax is for **compute**. It can only be hinted at with the rule:

```
compute 'column = expression' < table
```

expression can be quite complex. You must look at the manual pages for **compute** and **awk** to get an idea of all of the things you can do.

COHERENT Shell Scripts

You can also put commands (including pipes) into a file and execute the file by simply typing its name. These new commands are called *views* in the data-base literature because they give you a particular view of your data. Use a text editor to type the commands into a file, just as you would at the screen. Use a name for the file that you will remember and that reminds you of what you are trying to do with the series of commands, like **weeklyreport**. After leaving the text editor, type:

```
chmod 755 weeklyreport
```

or

```
chmod a+w weeklyreport
```

This COHERENT command makes the file executable. **chmod** means *change mode*. Whenever you want the series of commands that make up **weeklyreport**, just type:

```
weeklyreport
```

You can also use this new program in other COHERENT shell script files. If you want the new program to be available to others, put it in **/usr/local/bin**, or some other **bin** that is in the path of your users.

Suppose that at the end of each day, after making changes and updates to your **inventory** file, you wanted to fix up the file. If you wanted to recompute the **Value** column, right justify the number columns and left justify (blank fill the character columns); you could pipe together **compute** and **justify**. But since you will need to perform these functions often, you could create a program. The commands to fix up a table could be saved in a file with an easy-to-remember name. For example, the **fix** command could look like this:

```
# fix is a little routine to compute
# and right justify the number columns

compute 'Value = Onhand * Cost' < inventory | justify
```

Remember to type:

TUTORIALS

```
chmod 755 fix
```

or

```
chmod a+w fix
```

to make it executable. If you don't, you'll see

```
Permission denied
```

when you type the name of the file. (You may also see this when you type the **chmod** command if you don't have the right permissions.) To run **fix** just type its name:

```
fix
```

This yields:

```
Item     Onhand     Cost     Value     Description
----     ------     ----     -----     ---------------
   1          3        5        15     rubber gloves
   2        100        1       100     test tubes
   3          5        8        40     clamps
   4         23        2        46     plates
   5         99        3       297     cleaning cloth
   6         89       18      1602     bunsen burners
   7          5      175       875     scales
```

You can see how easy it is to write programs in the COHERENT environment. This is called *shell programming*. **/rdb** is an extension to the COHERENT system that adds data-base commands to the COHERENT commands.

COHERENT is good at this kind of *levels of power*. With most systems, you are limited to the functions the developers programmed into the system. With COHERENT and **/rdb**, you are limited only by your own imagination.

There are other useful COHERENT commands. **pr** will set up your output with head lines, and it will break on, and number, pages. **nroff** and **troff** can format your text and data for attractive printouts.

There are many different ways to initially create tables: editors, programs, shell commands such as **sed** or **awk**, etc. When **/rdb** tables are entered from scratch, **ve**, the **/rdb** table editor, is often used. **ve** allows the creation of tables quickly and in a familiar and easy way. It's a lot like **vi**. The first step in the creation of a table with **ve** is to create a screen definition file with any editor. Here is a screen file for the **states** file:

```
          The States File
st         < st >
state      <  state     >
```

ve uses this screen file to create the table. The rules for screen files are simple:

1. Column names go inside the angle brackets.

2. Anything outside of angle brackets is just text that appears on the screen.

The space between angle brackets is the viewable window over the field, and isn't a restriction on how wide the field can really be. After creating a screen file like this:

```
ve states
```

and the **states** file will be created. Let's say **ve** has been used to add new records to our **states** table so that it looks like this:

```
st     state
--     -----
CA     California
NV     Nevada
NY     New York
```

A mailing list can be created the same way, by making a 'screen' file and then using **ve** to add a few rows:

```
       Yet Another Mailing List

Name          <name               >
Street        <address          >
City          <city               >
State         <st>
```

This yields:

```
name    address     city          st
----    -------     ----          --
Joe     Main St.    Santa Cruz    CA
Bob     Broadway    Ithaca        NY
```

To *"Select the st and name columns from mlist and join it with the states table."* When we run the command

```
column st name < mlist |
sorttable |
jointable - states
```

it yields:

```
st     name     state
--     ----     -----
CA     Joe      California
NY     Bob      New York
```

The **sorttable** command was silent. But it has to be there. Both files to be joined must be sorted. The states file is already sorted. The dash in the **jointable** command means use the standard input, just like the COHERENT **join** command.

Report Writing

As with querying, report writing is also very easy. In fact, any query can be printed as a report by directing its output to your printer. Likewise, any report can be a query by directing it to your terminal (which is usually the default anyway).

For numeric information, **/rdb**'s standard table output adjusted with a **justify** or **trim** command is often sufficient, especially when combined with **tabletotbl** and the COHERENT **nroff/troff** formatters. In addition to these methods, **/rdb** has a **report** command that uses a prototype report form and has built-in command processing capabilities. It can be used to write form letters from a mailing list, produce invoices, packing slips, purchase orders, and so on.

Let's look at a sample report form. It's like the screen file for **ve**: text is outside brackets, and column names are inside brackets. Other commands can also go inside the brackets. Here's a report form for the *mlist* file:

TUTORIALS

```
<name>
<address>
< city >, <st>
                <! date +%D !>
Dear < name >,
This is a computer chain letter.
I am also sending it to:

<!  column name city < mlist |
row 'name != "<name>"' | justify  !>
Bye, <! echo <name> !>.
```

This yields:

```
Joe
Main St.
Santa Cruz, CA
                09/03/89
Dear Joe:
This is a computer chain letter.
I am also sending it to:

name     city
----     ------
Bob      Ithaca

Bye, Joe.
```

Arbitrary text goes outside the angle brackets; column names go inside angle brackets, and *any* arbitrary command or shell program or shell command(s) can go between exclamation marks within angle brackets, and you can *still* specify columns from the current record therein. You can even have reports within reports ...

Conclusion

In school you learned how to add, subtract, multiply, and divide. On the foundation of those simple operations can be built all the rest of arithmetic. These four operations work on numbers. A relational data base gives you a simple set of operations on *tables of information.* With **column**, **row**, **join**, and so on you can do enormously powerful things, but you will have to start thinking about what you want in terms of these operations.

Just as the **add** command does not care if it is adding apples or space shuttles, **column** does not care what the information in the tables is. Therefore, these commands can manipulate any data. This is an incredibly powerful idea that will take some time to get used to. These relational operations are a well thought out, pretty complete set of all of the things you will want to do with your data. It is up to your imagination to discover all of the powerful, yet quick and easy tricks you can do with these extensions of COHERENT.

TUTORIALS

Entering Data into Tables

You have seen how to access your data, now we have to think of how to get your data into the data tables. This is the real work of any data base system, so there are many tools to make data entry as easy as possible.

ve: The Forms Editor

ve was created to serve as the principal data entry and data editing system for **/rdb** data bases. It has much the same interface as the COHERENT text and program editor **vi**. We wrote it that way so as to minimize the amount of user training required to bring a new data entry person up to speed and to minimize the amount of confusion that could be caused by switching back and forth between the COHERENT text editor **vi** and the **/rdb** forms editor **ve**. Like the **vi** text editor, **ve** is a very powerful editor. Unlike **vi**, however, **ve** edits data in **forms** rather than in the free-form manner of **vi** and **ve** has optional features you can invoke to check the validity of your entries as you enter data, create an audit trail of your entries (or changes), automatically number your rows, and much, much more. **ve** also allows you to create **/rdb** tables directly from the forms editor and has a built in help facility so that you can review any of its features or commands in the middle of your editing session.

ve is called a *forms editor*. This means that you, the user, have control over the way your screen looks and the placement of the various fields that you will enter data into on the screen. The form definition of your **ve** screen is defined by a *screen file* or *screen template* that you create. To create a *screen file* use any text editor to set up a one screen form that consists of:

- The labels you want to have on the screen to prompt you for each column, and
- The other text or symbols you want on the screen to identify the row.

This procedure is called *painting the screen*. The labels on your screen are to be followed by the actual names of your columns, enclosed in angle brackets <*columnname*>. This is much the same format that you use to create a **report** template. For example, if you want to create a telephone list form for data entry, you might create a screen form that looks like this:

```
***Black Book****
Name   <Name    >       Phone <Number>
Street <Street        >
City   <City      >     State <State> Zip <Zip    >
```

The column names must be butted up against the left angle bracket. You can have spaces **after** the column name. Because **ve** is designed to handle variable length columns (and rows) you don't need to be concerned if the space you leave between the angle brackets in your form isn't enough to record all the data you plan to put into any particular column. **ve** will allow you to enter data in columns of variable length, even if you exceed the space available between the brackets on your screen, by shifting the *window* containing your data if you key in more data than you have space for on the screen. **ve** also provides a scroll left or right command and a zoom command so that you can see the full text of any column for which there is insufficient space on the screen to display.

Once you have created the *screen form*, you then create the table by invoking **ve**. If you tell **ve** to check the validity of your entries as you enter data into your table (with the **-v** switch), then **ve** will prompt you for validation constraints when you create your table. You can then set limits on column size, eligible characters to be keyed in, and table lookups in other tables to cross-check the validity of your entry. These validation constraints can always be modified later, so you do not have to get it perfect the first time.

ve also supports multi-user data entry or editing of the same table. It provides record locking features that will allow you to avoid the problems that could be caused by having two or more users attempting to edit the same row at the same time. With **ve**'s multi-user record locking, only one user has access to a row at a time. Subsequent requests for a row in use are inhibited, and a message is displayed.

This is a brief and general introduction to forms editing with **ve**. We have omitted a discussion of the special data security features provided with **ve**, issues such as protected fields in **ve**, instructions for optional cursor positioning, a **ve** command summary, a discussion of the default list screen format, auxiliary indexing, and more. These are covered in a subsequent chapter, in the **ve** manual pages, and in the on-line help screens of **ve**.

There are methods for data entry into **/rdb** tables *other* than **ve**.

Text Editor or Word Processor?

Another simple way to start a data table is to use a text editor or word processor. Type the column head lines and dash lines and the first few rows of your table. Then save it, leave the editor and try some of the query programs to see if you have set it up correctly. See the section on *Rules for Table and List File Set up*, below.

This approach is good for starting your data bases and for small data bases, but at some point a database may get too large for text editors. Then special purpose programs are needed. Each has its advantages and disadvantages.

enter: Mass Entry of Data

The **enter** command is an easy but limited way to type in a lot of data. For example, you or your operators might have to enter card files, or mailing lists, or other bulk data. You may not have time to train them to use a word processor or **ve**. With the **enter** program, they just type

```
enter
```

and the name of the file. The program then prompts them for each column of each row, which they type in. Each new row is simply appended to the table. It lets them go back to an earlier column if they make a mistake, but there is no validation or browsing of the table. It requires the lowest skill level and is the quickest to learn.

update: Update a Table

The **update** program lets you browse through a table file of any size, find a record with fast access methods if you wish, edit the record found with any text editor you like, and return the record to the file where it came from, or append the record to the end of the file if it is too big to fit back in where it came from.

Programs and Other Formats

Another great thing about COHERENT and **/rdb** is the ease of converting data files into different formats. The **/rdb** table and list formats can be converted to and from all of the other data base and word processor formats with simple shell, **awk**, **sed**, and other COHERENT program tools. You can use other programs, instruments, computers, tapes, floppy disks, networks, bar-code readers, natural-speech recognizers, expert systems, deep space probes, and more to get your data, then simply transform it into **/rdb** table and list format and use **/rdb** commands to do whatever processing you want. There are, therefore, a multitude of ways to get your data in. After processing, you can transform it again and send it back out.

Rules for Table and List File Set up

TUTORIALS

If you use **ve** to create and enter your tables you won't have to worry a lot about these rules. But if you don't use **ve**, you'll need more detail about setting up a data file. Your data is kept in files which can have either of two formats: *table* or *list*.

Table Format

A table looks like this:

```
Name     Title     Wage
-----    -------   ----
Mary     Pres.     4.98
Shawn    V.P.      2.98
```

To repeat, there are only three rules for making tables that the **/rdb** programs can understand. They are:

- There must be a single tab between each column.
- Put a column name, in the first line of the table, at the head of each column. Be sure there is a tab between each column name.
- Put a dashed line as the second line of each table above each column. There must also be a tab between each column's dashes.
- It is best if the dashed line is as wide as the widest string in each column, but it is not necessary.

The **justify** command will set it to that width for you, if you wish. Remember to use the dash or minus character '-', not the underline character '_'.

There are several ways to make your tables pretty and to handle special problems. These programs all require that you follow the three rules in order to work.

List Format

In addition to table format there is also list format, which looks like this:

```
Name     Mary
Title    Pres.
Wage     4.98

Name     Shawn
Title    V.P.
Wage     2.94
```

The rules for lists are similar to those for tables:

- The first character of a list-formatted file must be a newline. A newline character is what you get when you hit your **<return>** key. It appears on the screen as a blank line. Be careful that there are no unseen spaces or tabs in this first line that would also look like a blank line, but would confuse the programs into thinking they are looking at a table-formatted file. The programs use this first character to decide the format of the file so they can handle both list and table-formatted files.
- The file has exactly two columns. There must be a tab between the column name (i.e., **Name**) and the value (**Mary**).
- Each row (or record, corresponding to a row in a table) must be separated from the next row by a newline character. It looks like a blank line on the screen. (Note the blank line between **Wage** [for **Mary**] and **Name** [for **Shawn**] in the list example). There must also be a blank line at the end of the list file. Remember that a blank line has no characters in it. The programs can detect these two newline characters in a row. That is how they know that the end of the row

(record) has been reached.

Tab Problems

The use of tabs to separate columns in **/rdb** has many advantages, but they create a few problems. The advantages include:

- Ease of input. When you press the tab key, the cursor jumps to the next column. This is very natural to typists.
- Tabs look nicer in tables then printable characters. They line up columns and are a natural separator.
- COHERENT utilities like **awk**, **sort**, and others, know about tabs and handle them correctly.
- File size. No extra characters are needed to fill out the table (unless the table is justified).
- No schema or other special file must be created and maintained to tell the programs where data fields start and end, as is true with most other data base management systems. COBOL programs, for example, are even worse. They have the exact length of each field and record defined at the beginning of their code. This creates extreme inflexibility. If you need to change any of your data, such as making a string longer or adding a column, a COBOL program would have to be modified by a programmer and recompiled. The output of one COBOL program is unacceptable as input to another COBOL program unless the fields and their widths are exactly the same.

Some of the problems and their solutions are as follows:

Seeing the Tabs

You can see the effect of the tab, but not the tab character itself. So, if you make an input error, how do you know? You should often use the **check** program. It is especially important for large files. It will report any lines that don't have the same number of columns (tabs) as the head line. It will also display the line to simplify finding where the missing or extra tabs are located within the row. In most editors there are facilities for seeing **^I** for every tab on the line or in the table respectively. Outside the editor you can also use the **see** command or **od -c table** to see the special characters in the file. The **see** command turns tab characters into **^I**. After running **od**, out comes a strange-looking table showing each character including nonprinting characters converted so that they can be seen. **od** means **octal dump**, but the **-c** option means to convert the bytes of the file into their ASCII character representation. A tab is printed as: **\t**, as in the C programming language.

Table-Width Problems

Your screen is usually 80 characters wide. What if a table is wider than that?

- Use **ve** to view your table. If a field is wider than the screen, you can shift the field left and right. The default screen format tries to make a multiple column list of each field of a record.
- Use the list format when you create your table. See the manual pages for **listtotable** and **tabletolist** commands.
- You might decide simply to live with letting a long row in a table *wrap around* to the next line on the screen.
- Break the table into two or more smaller tables. You may join them back with the **jointable** command if you occasionally need to.
- A table created by **jointable**, in a pipe, can be up to the maximum character width that your available computer memory will hold.

TUTORIALS

- You can *project* only the columns you want to look at from your big table.
- You can get a wider terminal (some are 132 characters wide instead of the standard 80) or printer (some allow compressed print).

Special Characters

You can use special characters in the column names (in the header lines), but it is best not to. If you do, you will need to put quotation marks (") around the names when you type them at the terminal to protect them from the COHERENT shell. The rules for quotation marks, on COHERENT shell command lines, are: apostrophes ('...') absolutely protect everything enclosed and quotation marks ("...") protect every character except the dollar sign ($).

Therefore, put apostrophes around the whole string you are passing to **row**, **compute**, and **validate**. Use quotation marks around column names that contain special characters including spaces. For a complex example:

```
compute '"Item#" == 15 { len = length ("col 1") }' < table
```

Note that **Item#** has a special character in it, and that **col 1** has a space in it. All of these must be quoted ("). Also, the entire command must be enclosed within apostrophes ('). Even when you take these precautions, some COHERENT systems just don't like special characters. It's best not to use them. Limit column names to alphanumerics, don't begin them with numbers, and don't use any of the **awk** reserved words as column names.

Data Validation: How To Get Data Right

Data can be validated in many ways and at many times. **ve** allows you to specify validation constraints at data entry time, checking each entry as it's made to make sure it passes tests on what's allowed and what isn't. There is also a **validate** command which allows you to specify the conditions that are invalid, and the error message that should print out when the invalid data is found. The **validate** command is like the **row** and **compute** commands. They all pass their instructions on to **awk**, which does all of the hard work. First a simple example. If you had an inventory table like this:

```
Item#     Onhand     Cost     Value     Description
-----     ------     ----     -----     --------------
    1          3       50       150     rubber gloves
    2        100        X       500     test tubes
    3         -5       80      -400     clamps
    4         23       19       437     plates
    5        -99       24     -2376     cleaning cloth
    6         89      147     13083     bunsen burners
    7          5      175       875     scales
```

There are several invalid values that have gotten into your table. You might want to check that no **Onhand** item is less than zero. You can type this command:

```
validate 'Onhand < 0 {
                        print "negative Onhand in line " NR
                        }' < inventory
```

This yields:

```
Item#     Onhand     Cost     Value     Description
-----     ------     ----     -----     -------------
1              3     50         150     rubber gloves
2            100     X          500     test tubes
3             -5     80        -400     clamps
negative Onhand in line 3
4             23     19         437     plates
negative Onhand in line 5
5            -99     24       -2376     cleaning cloth
6             89     147      13083     bunsen burners
7              5     175        875     scales
```

You can also put the command into a file and call it something like **checkinventory**. It would then be a shell program that can be run regularly. See the **validate** command in the manual for more examples and information.

TUTORIALS

ve Form and Screen Editor

ve is a table editor designed for use on **/rdb** tables. Users who are familiar with the **vi** text editor will enjoy the advantage of understanding **ve**'s mode of operation and will already know most of the **ve** commands.

This section describes the commands and what they mean, how to set up **/rdb** tables, how to use the optional features to tailor **ve** to your application, and other considerations which will help you in both designing and using your tables.

ve Commands

Most **ve** commands are described in detail in the following sections; a complete command reference list is provided in the first **ve** on-line help page. **ve**'s command language is very much like the command language used by the **vi** COHERENT editor, in that both editors use regular letters and symbols as commands. As such, the keys you hit are interpreted either as commands to **ve** or as text that is being entered into your table depending on the *mode* **ve** is in when you type them: *insert* mode or *command* mode. For a summary of the **vi** editor, see the COHERENT Lexicon's entry for **elvis**, which is COHERENT's implementation of **vi**.

Because **ve** is essentially a *visual* experience, the best way to understand it is by sitting down at a terminal, creating a **ve** table, and experimenting with it.

Moving Around

The following table lists commands that move your cursor to a desired position on the screen. It indicates whether a given command *scrolls*, if it knows about *counts*, and if it can be used as a *target*.

Command	*Scroll*	*Count*	*Target*
<Return> <tab>	no	yes	no
B	no	yes	yes
l <space>	yes	yes	yes
h	yes	yes	yes
k j	no	yes	no
^ $	yes	no	yes
G	no	yes	no
H L	no	no	no
w b e	yes	yes	yes
< >	yes	no	no
f F ;	yes	yes	yes

The following describes each command in detail.

<Return> <tab>
Move the cursor to the next column in top-to-bottom, left-to-right order on your screen. The **<Return>** key also works in insert mode.

B Moves the cursor to the next column in bottom-to-top, right-to-left order on your screen.

l <space>
Move the cursor one character to the right.

h Move the cursor one character to the left.

k j The **k** and **j** commands move the cursor, respectively, up and down the screen one row.

^ $** The **^** and **$ keys move the cursor to, respectively, the beginning and end of the column.

G Move the cursor to the last row in your table. When you precede the **G** command with a count, **ve** displays the row in the *count* position in the table. For example, **100G** means "display the 100th row."

H L The **H** key moves to the beginning of the first column on the screen; the **L** key moves to the the beginning of the last column on the screen.

w b e The **w** key moves the cursor one word to the right; the **b** command moves the cursor one word to the left; and the **e** command moves the cursor to the end of the word.

< > The **>** command scrolls continuously through the column, left to right, until it reaches the end. The **<** command scrolls continuously through the column, right to left, until it reaches the beginning.

f F ; The **f** command looks left to right from your cursor for the character you specify; the **F** command looks looks right to left from your cursor for the specified character. The key you hit immediately after typing **f** or **F** is assumed to be the character you are seeking. When **ve** finds the character, your cursor moves to its position in the column. The **;** command repeats your last **f** or **F** request. It remembers both the character you last *found,* and the direction in which you were moving.

Occasionally, your screen will not be able to display all the data in a column or row because its width exceeds that of the the screen window. To compensate for this, many movement commands *scroll* left and right through the data, allowing you to see the entire contents of columns which are too long for the screen space allowed.

Some commands let you to specify how many times **ve** repeats the command before it repositions the cursor. These *counts* are entered as a number before the command. For example, the command **4l** moves the cursor to the fourth character in the column (left to right). The command **2b** moves the cursor two words to the left of your current cursor position.

When **ve** beeps or flashes at you instead of moving, it means that the command is inappropriate or that your cursor is already at your destination.

Displaying Your Data

There are two basic methods for displaying data on your screen. One is to display it sequentially, which is the way they appears in the table. The other way is to look for specific rows by searching the table.

Sequential display: **n, -**

The **n** (or *next*) command displays the row in the table that immediately follows the one now displayed. If **n** is the first command you give **ve**, it begins at the first row in your table. The **-** displays the row in the table that immediately precedes the one now displayed. If **-** is the first command you give **ve**, then it begins at the last row in your table. Both sequential display commands wrap around to the beginning/end of your table after the last/first row has been displayed.

Searching: **/**

When you use the **/** command, **ve** traverses your table row by row, looking for the pattern you specified. It can repeat a search without making you retype the pattern or character. If you have denied access to some columns by excluding their names from your screen file, **ve** will omit these columns in its search.

TUTORIALS

The **/** command begin its search at the position of your cursor in the row and is complete when the pattern you specify is found or when the entire table has been examined. For example, to search for *John Doe* in your **blackbook** table (which is described below), use the command */John Doe*. For you to see what you are typing, **ve** provides the last line of your screen to display the pattern as you type it in. You can use your erase character to edit this line. When you are done typing in the pattern, hit **<Esc>** or **<Return>** to tell **ve** that you are done. If *John Doe* is a pattern in your table, then the row which contains *John Doe* will be displayed on the screen.

Because **ve** remembers your search pattern, you need only type **/** and **Esc>** or **<Return>**, to repeat your search of a specified pattern. Each time you enter a search pattern, **ve** forgets the previous one.

Getting the Right Row

Usually the only requirement **ve** enforces when trying to match a search pattern to data is that the entire pattern is contained in a single column, regardless of which column it is or its position in the column.

Sometimes a search pattern will match data in many rows when you are looking for a specific row. Choosing a search pattern which is unique to the row you want displayed will speed up the search process. If this is not possible, you can limit the way **ve** searches by forcing the pattern to match the beginning or end of a column, or by specifying a particular column to which **ve** should limit the search.

Position matching: **^, $**

The character **^**, when it is the first character of a search pattern, tells **ve** that the pattern following it must match data at the beginning of a column. The character **$**, when it is the last character of a search pattern, tells **ve** that the pattern preceding it must match data at the end of a column.

If you are searching for the '^' character in the first position of your pattern, or the '$' character in the last position of your pattern, you must precede these characters with a backslash (\).

Search column selection: **m, M**

Another way to limit your search is to make **ve** search only one column in each row, using the **m** command. When you *mark* a specific column to be searched, **ve** doesn't bother trying to match your search pattern with data in any other column. As such, the search is quicker and the results are more likely to yield the desired row on the first search.

To use the **m** command, simply position your cursor at the column you want to search and hit **m**. All subsequent searches will be limited to data in that column only. Hitting the **m** key on a different column changes the search column to that column.

If you have marked a search column and you want to go back to searching all of the columns in each row, hit the **M** key to *unmark* your columns.

Inserting Text

Because most **ve** commands are ordinary characters, how do you tell when you are executing a command, and when you are entering text?

When you type one of the insert commands listed below, **ve** switches from *command* mode to *insert* mode. When you are in insert mode, everything you type is data and is displayed on your screen. You know you're in insert mode because the **<Insert/append mode>** message is printed at the bottom of your screen. If you don't see that message, then you are in command mode.

The only way to get out of insert mode is to hit the **<Esc>** key. When you do this, **ve** erases the insert message at the bottom of the screen and returns to command mode.

Several commands put **ve** into insert mode. Some commands destroy existing text (indicated by '*'), whereas others add to it. All of the commands except **O** take effect on the column at which your cursor is positioned, and, with the exception of the **o** command, at the actual position of your cursor in the column. The **C**, **c**, and **s** commands mark your text with a **$** symbol to show you the text that will be changed. This symbol disappears when you type over it, or when you hit the **<Esc>** key from insert mode.

The **r** or *replace* command is the only command which substitutes text without entering insert mode. To use the **r** command, position your cursor at the character in the column you wish to change, type **r** and the character to replace it. The **r** command can be used with counts. For example, to replace the next four characters with the letter 'x', use the command **4rx**.

Command	*Means*
a	Append text to the right of the cursor
A	Append text at the end of the column
i	Insert text to the left of the cursor
I	Insert text at the beginning of the column
o*	Open the column and insert text at the beginning
O	Open a blank row
c*target**	Change the text between the cursor and *target*
C*	Change text from the cursor to the end of the column
s*	Substitute the character at the cursor for text
R*	Replace (overwrite) text character by character

Deleting Text

The delete commands let you delete entire rows and columns, partial columns, or characters from your table:

Command	*Means*
dd	Delete the entire row
D	Delete text from the cursor to the end of the column
x	Delete the character(s) at the cursor
d*target*	Delete text from the cursor to *target*

The **x** command also knows about counts.

Targets

The insert command **c** and the delete command **d** require that you specify a *target*. A target is exactly what it implies — a destination for the command being executed. Commands that can be used as targets are listed in the section on moving around. The table below shows sample **d** and **c** commands and what they mean when used with different targets.

Command	*Means*
3dw	Delete the next three words
de	Delete to the end of the word
c2f*g*	Change the text to the second *g* found moving right
c^	Change the text to the beginning of the column
c$	Change the text to the end of the column
d;	Delete text to character specified by last **f** or **F** command

All targets assume that the delete or change action begins from the position of the cursor and

TUTORIALS

counts must be specified before the actual target.

Yank, Put, and Undo

Sometimes accidents will happen when you are editing — you will hit the wrong key, or decide that you really don't like a modification. You may even delete an entire row and then realize it shouldn't have been deleted. Or perhaps you are entering rows which are almost identical and you are sick of typing in the same thing over and over again. **ve** has a number of commands that help you to efficiently deal with a change of mind after a change of text, mistakes and repetitive data entry. These commands are known as *yank*, *put* and *undo* commands.

Yank commands copy data from the row displayed on the screen into a special buffer. *Put* commands put the yanked data from the buffer into the row on the screen.

Yanking data does not alter the row from which it is copied. Generally, once data has been yanked, it can be put indefinitely without having to yank it again. Because **ve** remembers only the text last yanked for each column displayed, yanking data from the same column in another row overwrites previously yanked data in that column with the text that is currently displayed. When entire rows are yanked or deleted, all yanked columns are overwritten with the text in the displayed row.

Selected columns are yanked with the **y** command. To use it, position your cursor on the column you want to yank and hit the **y** key. Different columns may be yanked from different rows.

All columns on the screen can be yanked with the **Y** command, and it works by hitting the **Y** key from any position on the screen.

Putting data replaces the data in the row or column being displayed with the data last yanked by a **y**, **Y**, or **dd** command. Data can be put only in columns from which it was yanked. For example, you can't yank text from a *phone number* column and put it in a *customer name* column.

The **p** command replaces the data in the column at which your cursor is positioned with data that has previously been yanked from that column.

The **P** command replaces the data in all columns currently displayed with data that has previously been yanked from each column. Since **veconsiders deleted rows to be yanked text, using the P** command after mistakenly deleting a row will restore it.

ve remembers the data in the last column modified prior to the last command which modified it. You can restore the data to that state by hitting the **u** key. The **u** command works only for undo-ing modifications made to the row currently being displayed, and will not undo what's already been undone.

For Your Information

The **#** command displays the list of topics in the on-line **ve** help files available for perusal. To select a help file, position your cursor at the **==>** symbol to the left of the desired topic by using the **j**, **<Return>**, **-**, and **k** keys to move down and up the list. Then, hit any other key (except **q**) to view your selection a page at a time. The **j** and **<Return>** keys display successive pages, and the **-** and **k** keys display previous pages. Any other key (except the **q** key) returns to the table of contents. Hit the **q** key to get back to your data display.

Other commands which provide assistance with or information about the current screen contents:

- The **v** command displays the validation requirements for the column on which your cursor is positioned.
- The **t** command prints the look-up table for the column upon which your cursor is positioned, if such a table exists.

- Use the **S** command to redraw your screen if your data display gets messed up.
- The **z** command draws a window under the column your cursor is on and displays the contents of the column in the window. This command comes in handy when the data for a particular column does not fit in the screen space allowed. The next key you hit will erase the window.

Writing Rows

Whenever you change the data by adding or changing a row or column, you must use a **W** or **ZZ** *write* command to make your changes or additions part of your table. When you use the **W** or **ZZ** command, **ve** re-checks the data to make sure that all validation tests (if any) have been passed.

When you are using the **W** command, and the data does not pass your validation tests (if any), **ve** will complain and refuse to write your row in the table. In this case, your cursor will land on the column which contains the bad data and you can fix it. If there are no tests, or when the data passes your tests, **ve** will write your row in the table, and await your next command.

The **ZZ** command is a quick way to write your row and exit from **ve**. If the row on the screen has been added or modified and it passes your validation tests (if any), **ve** will put it in the table before quitting. It will not tell you whether or not it writes the row.

If your data has been changed and you attempt to display a different row with a before using the **W** command, **ve** will warn you that the row has not yet been written. Repeating the command will override the warning, and the row on your screen will not be written in your table.

Getting Back to COHERENT

There are two kinds of commands which will get you back to COHERENT: *shell* commands (**!,%**) and *quit* commands (**q,ZZ**). Shell commands let you talk to a new COHERENT shell while your **ve** process is suspended in the background, waiting for you to return. Quit commands end the **ve** process and return you to your COHERENT shell.

The **%** command displays a **%** symbol at the bottom of your screen and puts **ve** into suspended animation. When you see your login prompt appear at the bottom of the screen, COHERENT is ready for your command. To resume your conversation with **ve**, hit **<ctrl-D>** (your EOF character). If you are not familiar with COHERENT processes and shells, remember that it is easy to get carried away with COHERENT and forget that **ve** is still patiently waiting for your return. You will know something odd is happening when you try to log out and COHERENT keeps saying *not your login shell*. You must first return to **ve** and give it a proper *quit* command. If COHERENT still complains, it means that you have repeated the **%** command a succession of times, and you must repeat the **<ctrl-D>q** sequence until you can log out.

The **!** command before a COHERENT command executes the COHERENT command at the bottom of the screen and then immediately return to **ve**. To read the results of your COHERENT command, the message

```
[Hit RETURN to continue]
```

appears and **ve** awaits your response before it redraws your screen. The '%' symbol, when used in the '!' form of the shell command, is replaced with the name of the table you are currently editing. To repeat a **!** command, type **!!**.

Quit Commands

When you are finished using **ve**, you can use the **q** or **ZZ** command to exit. The only difference between the **q** and **ZZ** commands is that **q** complains if the row on the screen has not been written after a modification has been made. (A second **q** command simply exits without writing the row.) The **ZZ** command will automatically write your row before exiting **ve**.

TUTORIALS

Colon Commands

People accustomed to using the COHERENT editor *vi* are already familiar with *colon* commands. Colon commands are always preceded by a colon (**:**). When the colon key is hit, it appears at the bottom of the screen and the remainder of the command appears next to the colon as you type it. You must enter at least the first letter of the colon command for it to work (letters enclosed in [ ]'s show how much of the command you don't need to type). You must end colon commands with **<Return>** or **<Esc>** to let **ve** know you are done.

Colon Command	*Is the Same As*	*Described in Section*
:n[*ext*]	**n**	Sequential Display
:o[*pen*]	**O**	Inserting Text
:h[*elp*]	**#**	For Your Information
:w[*rite*]	**W**	Writing Rows
:s[*hell*]	**%**	Shell Commands
:!*command, !!*	**!**	Shell Commands
:q[*uit*]	**q**	Quit Commands

Macros

A *macro* command is defined as one or more **ve** or COHERENT shell commands mapped to a single control key. Each time the control key is hit, the command or sequence of commands defined by that control key will be executed.

Macro commands can be used to simplify an often-repeated sequence of commands into a single keystroke. Because control keys are non-printable characters and cannot be confused with potential data (unlike regular **ve** commands), they work from both command mode and insert mode. The mode from which you execute the macro is restored after successful completion of all the commands which define it. It is even possible to design a set of macro commands that makes **ve** appear to be a single-mode editor. This might be desirable for beginning users who find two edit modes confusing.

A maximum of 23 macro commands may be defined for the **<ctrl-A>** through **<ctrl-Z>** keys. If your erase character or your interrupt character is a control key in this range, then it (they) are excluded from the list of control keys available for macros. Macro commands that include a second macro in their list of commands will replace whatever commands follow the second macro in the list with the list of commands in the second macro. Embedding macros in macros is confusing and it is not recommended.

Macro definitions are stored in a **ve** table named **.verc**. If you want to define your own macros, this file must be in your home directory where it can be shared by all **ve** tables in your account. If you don't have your own **.verc** file, **ve** will use its own.

The first column in the **.verc** file contains the letter which is hit simultaneously with the control key to make the control key code. The second column in the **.verc** file is the command or commands that are executed when the that key is hit. Four non-printable commands which you may want to use in your macros are **<Tab>**, **<Esc>**, **<Return>**, and **<Backspace>**. They should be indicated by **\t**, **\e**, **\n**, and **\b**, respectively. Below is a copy of the default **.verc** file **ve** uses.

Control Key	*Command*
a	#
d	ddn
o	O
q	ZZ
r	S
s	%
t	t
w	W

For example, let's say we're editing the **blackbook** table and we forgot to type in the area codes for local phone numbers with **429** prefixes. If we *mark* the phone column using the **m** command, and the macro

```
p       /^429\eI(408)\eW
```

is a row in our **.verc** file, then every time we hit **<ctrl-P>**, **ve** finds rows in which the first three characters of the phone column are **429**. **ve** will then insert **(408)** at the beginning of the column and write the row.

Shell Macros

If the command column in the **.verc** table begins with '!', then the remainder of the command column, up to the end of the column or another '!', is passed directly to the shell. If the command contains valid column names enclosed in '<' and '>', then the contents of the named column(s) currently displayed on the screen will be substituted *before* the shell is called.

Using the **blackbook** table, let's add a macro which will dial the emergency phone number when we hit **<ctrl-E>**:

```
e       !dial <phone>\n
```

Like regular shell commands, shell macros also expand '%' to the name of the current table.

For example, let's say we have a subdirectory named **Personnel** containing text files about each employee in the **personnel** table. To keep these files distinct and unique, each is named by the Social-Security number of the employee. See if you can figure out what the following shell macro will do:

```
g       !cat 'echo % | cap'/<ss number>\n
```

ve will echo the interpreted shell command on the message line, send it to the shell for execution, and then print the

```
[Hit RETURN to continue]
```

message before refreshing your **ve** screen. Placing a '-' after the '!' in the shell command line will suppress the command echo, the pause and the screen refresh.

To see a list of the **.verc** commands in effect while you are using **ve**, hit **<ctrl-?>**.

The Command Line

The files **ve** uses and the way it displays and formats your data are determined by entering a single command line at the prompt. The syntax for **ve** is:

`ve [`*data* `[ -s [`*file*`] -h [`*file*`] -a [`*file*`] -v [`*file*`] -n[`*n*`] -f`*c* `-m`*c* `-i]] [-b]`

Dashes which are followed by a single letter are called *switches* (**-s, -h, -a, -v, -n, -m, -i, -b**). Each switch tells **ve** to perform a specific job when it is invoked.

TUTORIALS

Words, symbols, and letters that directly follow each switch, when substituted for file names, numbers, and characters, further define how **ve** should do the job indicated by the switch. The word *data* shows where you would type in the name of your table; the *file* words show where you would substitute the names of any files you might use. The **n** symbol shows where you would substitute a number, and the *c* (as in **-f***c* or **-m***c*) shows where you would put a letter or character.

The brackets ([ ]) are used to indicate items on the command line that are *optional*, and should not be used in an actual command line. Note that with the exception of the word **ve**, everything on the command line is optional. Thus, the command

```
ve
```

is a perfectly valid **ve** command that displays the on-line help file index.

File Options

The first four switches in the command line apply to files that, with the exception of the screen file, are in **/rdb** table format. They let the user define how the screen displays the data (**-s**), separate the header lines from the data rows (**-h**), keep track of each modification made to the table (**-a**), and impose various restrictions on the data as it is entered (**-v**). Each of these switches can be used with the name of a file.

File Creation

How do we use the file options when the table or the files don't exist? The best and simplest way is to let **ve** create, format and store the necessary files. This can be done in a single command, so that all files are created and initialized at the same time. The following table shows the conditions under which each type of file is created:

ve *will create*	*When*
The table	There is a screen or a header file
The header file	There is a screen file and no table
The audit file	There is a screen or a data or a header file
The validation file	There is a screen or a data or a header file
The screen file	Never — the user must do this

By examining the table above, it is apparent that the only file **ve** really needs in order to create all of the other files, including the table, is a screen file. The preferred method for initializing a **ve** table (starting up from scratch with no files), is:

A. Create a screen file, making sure to include all of the column names you want to be part of your table.

B. On the **ve** command line, type in:

1. The name of your table,

2. the **-s** switch with the name of the screen file you've just created,

3. and the file option switch followed by a name for each file you wish **ve** to create.

For example, let's say we want to create a table named **blackbook**. Also, we want to impose certain limitations on how the data is entered in some of the columns, and put these limitations in a validation file called **limits**. So far, neither of these files exist.

The first thing we do is create a screen file which we name **display**. In the **display** file, we include all of the column names we want in our **blackbook** table. When we finish making our screen file, and type in the command

```
ve blackbook -s display -v limits
```

the following steps are performed:

A. **ve** looks in the **blackbook** file for the header rows. But *blackbook* doesn't exist, so **ve** creates it.

B. **ve** then reads the screen file we have created named **display.** It creates the header rows from the column names in the screen file, and puts them in the new **blackbook** table.

C. **ve** creates a validation file named **limits** and uses the column names in the first header row to create the rows in **limits.**

When **ve** is finished making all the new files, it draws your screen according to the *display* screen file, and puts you in **ve** edit mode, exactly as if the files had already existed before issuing the **ve** command.

Typing in the **ve** command line can be time consuming and tedious when using the file option switches, especially if you are using any of the other switches as well. **ve** has a built-in method of naming optional files associated with a table. When you take advantage of this feature, the command line is greatly simplified.

Using the Default File Names

Whenever a **ve** command is issued with the name of a table, **ve** looks for files with names that begin with the name of the table and end with each of the four file option switches. These files are called default files, or files which are used by **ve** in the absence of specified files. For example, consider the table named *blackbook*:

File Switch	*File Type*	**ve** *Looks For*
-s	Screen	**blackbook-s**
-v	Validation	**blackbook-v**
-a	Audit	**blackbook-a**
-h	Header	**blackbook-h**

If any of the default files exist, **ve** automatically includes them in the command line.

Specifying a file option switch with a file name overrides **ve**'s inclusion of the existing default file in the command line. As a corollary to this, files that are to be included in the **ve** editing session which do *not* have default names must *always* be typed in the command line, following the appropriate file option switch. For example, suppose you want to use a screen file named **myscreen** instead of the default screen file named **blackbook-s**:

```
ve blackbook -s myscreen
```

Creating files from scratch also becomes much easier when using the default **ve** file names. The same rules apply as above for creating files, but the names of the files do not need to be typed in. Let's assume that we are starting out fresh with no files. We want to start a new table named **blackbook**, with a validation file and an audit file. If we create a screen file and name it **blackbook-s**, then the command line

```
ve blackbook -v
```

is all that is necessary to create the table named **blackbook**, and the validation file named **blackbook-v**.

Automatic Row Numbering (-n)

The **-n** switch tells **ve** to give each row a unique numerical identifier automatically as it is written. **ve** looks for and stores this number in the first column of each row.

TUTORIALS

If you are initializing a new table and you use the **-n** switch, you must reserve the first column for this number. Of course, you may name the column anything you like — it just has to be there. **ve** will start the row numbering at one.

If your table already exists, **ve** figures out what to number new rows by scanning the first column of each row in the table and adding one to the largest number it finds.

The **n** option to the **-n** switch, when substituted with a number, tells **ve** to start new rows with that number. If the number you specify is less than the greatest numbered row in your table, **ve** simply ignores your request, since it would mean duplicating existing row numbers.

Once the **-n** switch has been specified in the command line for a particular table, it is not necessary on subsequent calls to **ve** to include **-n** in the command line (unless you are resetting the number with the *n* option), since **ve** keeps track of the largest numbered row. The command

```
ve blackbook -n2001
```

is an example of using the **-n** switch to invoke automatic row numbering beginning with 2001. When you are editing a table that uses automatic row numbering, and the first column is displayed on the screen, your cursor will not land on it. This is **ve**'s way of making sure that you don't mess it up.

Start-up Mode Specification

The **-m** switch tells **ve** to enter the editing session in a particular mode. This mode is specified by a character which must follow the **-m** switch, and can be **i** for insert mode, **/** for search mode, or **n** for next mode. **ve** will return to this mode whenever a row is written. If, for example, we want **ve** to start up in the insert mode, we would use the command:

```
ve blackbook -mi
```

Initialization Option

ve creates and maintains an index of the rows in your table. During **ve** edit sessions, this index keeps track of which rows are being accessed by other users and which rows have been deleted. When the table is quiet, **ve** uses the index to clean up the table by removing the deleted rows and putting the updated rows in order. The index is also used to store the last automatic row number (if applicable) and the column separation used by your table.

If your table has been modified using using a method other than **ve**, the index must be recreated. This is done with the **-i** switch, which means **i**nitialize the table. It is not necessary to use the **-i** option when you are creating the table.

The Screen File

A screen file is an ordinary text file which is created with any editor. It visually describes the way the data are displayed during a **ve** edit session. If there is no screen file, **ve** displays your column names in columns on the screen with the corresponding data adjacent to the name. There are several reasons for using a custom screen file:

- It is the simplest and most accurate way to create your table.
- It allows you to take advantage of **ve**'s security features.
- It allows you to customize or expand your column labels and helps to make your screen display match specific hard copy forms.
- It allows you to rearrange the order or placement of your column names to accommodate the size of data in the column.

- it allows you to set the column on which your cursor will land each time a row is displayed.

Screen File Format

The screen file should be formatted to look the way you want **ve** to display your data. Use the following guidelines when creating a screen file:

A. For each column, indicate the beginning and ending positions for data display in the screen file with the '<' and '>' angle brackets, respectively. Both brackets must appear on the same line. Type the name of the column between the brackets. If **ve** identifies text surrounded by **<>**'s to be a valid column name, then data in that column will be displayed between the brackets, and the brackets themselves will become blank.

B. If you are using the screen file to create a table, keep in mind that the names of all columns must be included in the screen file, and that their top-to-bottom, left-to-right order on the screen will determine the order in which **ve** arranges your data. (This consideration is usually important only when you are planning to use automatic row numbering or an audit file with your table. In that case, the uppermost, leftmost column name will be assumed to be the name of the number column.)

If your table already exists, you must be sure that the column names in the screen file are identical to those listed in the first header row of the table.

C. The entire screen file should be no longer than 23 lines and no wider than 80 characters, since this is the most information that standard screens accommodate (the 24th line is reserved by **ve** for messages).

Here is the screen file for our address-book table (**blackbook**):

```
                    *** Black Book ***

         Name:  <name          > number: <phone       >
      Address:  <street        >
City/State/Zip:  <city          >  /  <state>  / <zip  >
```

Protecting Columns

Once a table has been created, a screen file can be used to selectively protect specific columns by denying or limiting access to those columns. When you **ve** a table with a screen file, *only* the data for columns which are named in the screen file will be displayed. Thus, the simple omission of the names of confidential columns in the screen file protects those columns from being displayed. Data in columns not named in the screen file are also excluded from searching.

This feature comes in handy when you want to use **ve** to edit a specific subset of columns and do not need to see the entire row.

By limiting access, we mean simply disallowing modification of designated columns. In the screen file, column names preceded by an exclamation point (!) indicate to **ve** that they cannot be modified.

To illustrate the security features, let's use the **personnel** table described earlier. For the sake of this example, we'll add two columns for the name of the person to contact in an emergency, and the phone number of that person. The header rows for such a table look like this:

```
employee   ss number   emp. date   salary   emergency   phone
--------   ---------   ---------   ------   ---------   -----
```

Let's assume that this table exists, and we simply want to update it. We want to be able to search for employees by their name or their Social Security number, but we don't want these columns modified. Also, because the salary column is confidential, we don't want it displayed at all. Our screen file might look like this:

TUTORIALS

```
               Company X-Y-Z Employee Information Card
                ____________________________________

           Employee !<employee    >
    Social Security !<ss number   > Date employed <emp. date >
In case of emergency <emergency    >        number <phone     >
```

Setting the Cursor

Unless you are searching for a particular pattern, **ve** will position the cursor at the uppermost, leftmost column name on the screen.

The column name on which your cursor automatically lands can be changed by preceding the desired column name with an asterisk ('*'). For example, if we were updating the emergency name and phone number columns in the **personnel** table, we could make the cursor land automatically on the **emergency** column as each row is displayed, by changing the last line of the screen file to look like this:

```
In case of emergency *<emergency    >        number <phone     >
```

The Validation File

A validation file describes column-by-column requirements that data must meet as it is entered in the table. There are three different kinds of tests that you may force your data to pass before allowing it to become part of your table: character range specification, column length restriction, and table look-up.

Validation File Format

The validation file is a regular **ve** table. The header rows look like this:

```
name       characters      length       table
----       ----------      ------       -----
```

For each column in your table which must pass one or more validation test, there is a row in the validation file. The first column of each row in the validation file contains the name of the table column.

Validation File Creation

ve can create your validation file, but you must supply the validation parameters. To make this task easier, **ve** calls on itself to edit the validation file, before opening up and displaying your table.

If you don't want to make a column pass validation tests, simply delete the row which contains the column name from the validation file. When you are finished editing your validation file with **ve**, your table will be displayed on the screen, and the validation parameters will take effect immediately.

Character Range Specification

This validation option allows you to specify exactly which characters are acceptable (or not acceptable) in a named column. For example, the *ss number* column of the **personnel** table should be comprised of numbers and dashes. The **employee** column can be letters (upper case and lower case), dashes, periods, commas, and blanks. The **zip** column of the **blackbook** table must be numbers only.

ve allows you to specify these parameters in terms of ranges, or as single characters, in the second column of your validation file. A range is defined as a low limit character and a high limit character separated by a dash '-'. When more than one characters or range of characters are specified, they are separated by a comma ','. Using the above examples, the character specifications for **ss number**

is '0-9,-'; for **employee** is 'A-Z,a-z, ,-,.,.'; for **zip** is '0-9'.

Let's say we have a column which can contain anything except upper case letters:

```
a-z,0-9,?, ,,,.,:,;,!,=,+,<,>,[,],(,),*,&,^,%,$,#,@,{,},|,~,',-
```

The '!' symbol, when it precedes a range or character, means *not* this range, or *not* this character. Sometimes, as in the example above, the '!' syntax can greatly simplify the parameter list: '!A-Z'.

The '!' can also be used to exclude specific members of a range. For example, one way to specify all numbers except **8** is:

```
0,1,2,3,4,5,6,7,9
```

A simpler way is:

```
0-9,!8
```

Column Length Restrictions

ve looks at the third column in each row of the validation file for column length limits. These limits are used for several reasons. For example, we know that zipcodes should be exactly five numbers long. We know that Social-Security numbers are exactly 11 characters. Data entered in these columns which are not the proper length make them meaningless.

Also, it is sometimes the case that other limits indicate the necessity for a column-length restriction. If, for example, we use the **blackbook** table to produce labels, then we should restrict the length of the columns in the table to conform to the size of the labels.

Another use of column length might apply when you want to force a column to contain data, but you don't care (or don't know) what the length should be. For example, in the **date** column of the **personnel** table, you might require that something be entered, but it might be of indeterminate length (October 1981, 10/1/81, 10/81, etc.).

Four specifications describe exactly how **ve** should pass data for the column length test:

=*n*	The column length must be exactly equal to *n*
<*n*	The column length must be less than *n*
>*n*	The column length must be greater than *n*
!	The column can be blank (length 0)

Using the above examples, the column length for **zip** column in the **blackbook** table is **=5**, which means *exactly equal to five characters.* If the size of our labels is 32 characters, then the column length for the **name** and **street** columns of the **blackbook** table is **<33**, which means *less than 33 characters.* The column length for the **date** column of the **personnel** table might be **>0**, which means *there must be at least one character in this column.*

When more than one of these specifications is used, they are separated by a comma ','. The '!' column length specification, when used in conjunction with other specifications, tells **ve** to ignore other length specifications if the column is blank. For example, if you specify the **zip** column length mentioned above to be **!,=5**, then **ve** will allow you to skip over that column — however, if you enter any data in this column, **ve** will force you to make it exactly five characters long.

You can also indicate column length ranges by using the **<** and **>** specifications together. (In fact, another way of saying **!,=5** is **!,<6,>4**.) In the **personnel** table, the least number of characters needed to specify **date** is four (1/82); the greatest number is 18 (September 11, 1949). You may wish to place an upper limit on this column simply to preclude the user from typing in something silly like, "Gosh, I don't know, I guess Bud joined the firm round about September, a few years back, because I first met him at the company's Labor Day barbeque." In any case, the specification is **>3,<19**, which means any data in this column must have more than three characters and less than nineteen characters.

TUTORIALS

Look-Up Tables

The fourth column in the validation file is for the name of a look-up table. If you use table look-up validation on a particular column, then each time you enter data in that column, **ve** compares it with data in the table. **ve** makes this comparison based on how you specify the name of the table: if it is preceded by '!', then it passes the table look-up test only if it is *not* in the table; otherwise, it passes only if it *is* in the table.

ve updates all tables created from a table as you change, delete, and add data to columns in that table. As such, you need to create each table only once. As long as you always use **ve** to maintain your table, your tables will be current.

Table Creation

The **vindex** command is used to create look-up tables. To use **vindex** to create a new look-up table (or overwrite an existing one), you must specify the name of the table and the names of the columns in the table you want to index.

For example, if we want to create look-up tables for the **ss number** and the **employee** columns of the **personnel** table, we would use the command:

```
vindex personnel employee "ss number"
```

vindex creates two files for each table it makes, using the naming convention *column-A* and *column-B*. If the name of the column you are vindexing is longer than 14 characters, it will be truncated to 14 characters before the **-A** and **-B** extensions are appended to the resulting index files. Spaces are converted to '_' and special characters which may be part of the column name are omitted. In this example, the index files for the **ss number** table would be named **ss_number-A** and **ss_number-B**. The **ss number** table is referred to in the validation file simply as **ss_number**.

Decode Columns

An optional feature when creating and using look-up tables is the *cross-indexing* capability. When you are creating a table for a column in a table, you can select a second column which should be displayed when your data matches a pattern in the table. The displayed data is called a *decode* column. The decode column must also be in the table, and, because each indexed or key column can have only one associated decode column, **vindex** assumes that data in the key column is unique. Similarly, when **ve** is used to update tables which contain decode columns, the contents of the key column must be unique.

For example, consider the following *states* table:

```
state code    name
----------    -------------
NY            New York
CA            California
MA            Massachusetts
```

In each row of the **states** table, the **state code** column contains a standard state abbreviation: the **name** column contains the name of the state that corresponds to the abbreviation. To create a look-up table on the **state code** column and cross-index the **name** column, we use the command:

```
vindex states "state code" :  name
```

The colon ':' character in the command above is used to indicate that the following column name (**name**) is a **decode** column for the column name preceding the colon (**state code**).

Unique Columns

We can use the **ss_number** table created previously to illustrate the use of look-up tables to enforce **unique** column entries. In order to make sure that we don't enter duplicate employee rows, we can make **ve** check each Social Security number as it is entered against a table of those already in the table by specifying the name of the look-up table in the fourth column of the validation row for **ss number**, using the '!' symbol:

```
name         characters     length     table
---------    ----------     ------     ----------
ss number    0-9,-          !,=11      !ss_number
```

In this case, the column passes the test if **ve** does *not* find an identical Social Security number in the table. When we are editing the **personnel** table using **ve**, each time we add a new Social Security number **ve** will look it up in the table: if it's already there, **ve** will complain; otherwise, **ve** will add the new number to the table.

Column Inclusion

The **blackbook** table will illustrate how table validation can be used to make sure that the data we enter in the **st** column are **included** in a table which lists all of the standard state abbreviations.

We specify the name of the **State_code** look-up table in the **blackbook** validation file. Each time we enter a state abbreviation in the **st** column of the **blackbook** table, **ve** will look it up in the **State_code** table. If it's there, **ve** will print the name of the state next to the abbreviation.

```
name    characters    length    table
----    ----------    ------    ----------
st      A-Z           =2        State_code
```

Multi-User Considerations

ve is a *multi-user* table editor. This means that any number of users can be entering or editing data in the same table at the same time. When a row is displayed on your screen, it *belongs* to you until you release it by moving to another row, writing the row, or deleting it. That means that anyone else who may be using the table at the same time will be prevented from viewing or editing that row until you are finished with it.

Because **ve** knows which rows are in use at any given time, you will sometimes see the message

```
1 other match(es) currently in use
```

when you search for a pattern in a row which *belongs* to someone else. This is to let you know that the pattern is in your table, but the row which contains the pattern simply isn't available.

When you use a sequential display command (**n**, **-**), **ve** will skip over rows which *belong* to other users until it finds a free one. When **ve** has to skip over rows, it will let you know how many. Similarly, the **G** command will let you know that it can't land on a specified row because either it is currently in use or it has been deleted during the edit session.

/rdb-ve Compatibility

If you are planning to use **/rdb** on tables that are created and maintained by **ve**, taking the following precautions when setting up your tables will help you avoid trouble later on.

Because many **/rdb** commands use table column names as command line arguments, care must be taken to exclude symbols or words which have special meaning to the shell from your column names. Specifically, stay away from the '|', '#', and '!' characters; if column names can be confused with shell commands (ie., **date**, **who**), remember to protect them from shell interpretation by enclosing them in quotes (") when you use them as arguments to **/rdb** commands.

TUTORIALS

When using **ve** in a shell script, care must be taken not to access the original file until **ve** is done with it. You can check for this by testing for the existence of a *data***+lck** file. If the *data***+lck** exists, wait until it's gone.

Some **/rdb** commands use the **awk** program, which has its own command language. Below is a list of **awk** words which should not be used as column names:

```
BEGIN     exit      in        next      sqrt
break     exp       index     print     substr
continue  for       int       printf    while
else      getline   length    split
END       if        log       sprintf
```

Limits

Two basic size limits imposed by **ve** are the the maximum number of columns and the maximum number of characters or bytes in each row. On the COHERENT installation, the maximum number of columns is 66 and the maximum row length is 2,048 bytes.

ve will be able to accommodate most tables. If you need more than 66 columns in a row we recommend that you create two separate tables, each with a common column, and use the **jointable** command to merge the two tables after data entry has been completed.

There are no limits on the number of rows in your table or on the size of columns within a row, as long as the 2,048 byte per row limit is not exceeded.

If you use the look-up table validation feature, you should be aware that a 64-byte limit is imposed by truncation for the the lengths of the key and decode columns.

ve Validation

The validation table is used to specify, column by column, requirements that data must meet as they are entered into the data table by associating a shell-level command with the name of the column to be validated.

By default, all columns with associated validation commands are tested when the column contents change or when the record is written, by executing the associated command and using its return status to determine success or failure. A non-zero return status means the test failed. **ve** will beep and the cursor will remain at the column until the contents are modified to comply with the validation requirements.

Unless the # symbol precedes the command, the command is also executed whenever the cursor passes through the field, unless nothing has changed since the last time the field was passed. However, if the '@' symbol precedes the command in the validation file, the command is executed even if nothing has changed. If the '#' symbol precedes the command, the command is executed only when the record is written (or deleted).

Unless the command is preceded by '—', the exit status of the command determines whether the data in the field pass the validation test.

The shell that executes the validation command has some variables and constructs made known to it by **ve**. Fields from the current record are passed to the validation command by enclosing the field name in angle brackets, as in the screen file. The construct **<@field>** is used to denote the contents of a field when the record was first displayed on the screen, before any changes were made to it. The '%' symbol in the command line expands to the current file name; the shell variables **$ROW** and **$COL** are defined in the environment for the executed shell to correspond to the start position of the current field, and **$COLEND** is defined as the first available white space after the field.

The validation file is a table with two columns:

```
column command
------ -------
```

The first characters of the 'command' column have special meaning:

@ This means execute the command even if nothing in it changes This is typically used in conjunction with the '!' prefix, to put something into the field without having the operator type it in, like the date.

— This means ignore the exit status of the command and always pass validation. This is used when the command to be executed doesn't really have anything to do with validation. For example, it can be used to pop up a new window and display an image. The window might be thrown away, producing a "fail" exit status, but we really don't care.

! This means replace the field contents with the output of the command and can be used to force the output of the date command into a field, or to calculate a field based on one or more other fields.

This indicates that the command should only be executed *only* when the record is written or deleted, and *not* when the field is merely changed or passed through.

The standard error of the validation command is sent to the status line of the ve screen, so you can customize the error message to include the field being validated or other arguments passed to the validation command.

These commands used to be called "user exits" when this technique was used on IBM systems.

Here's a sample validation file for the inventory file:

```
name     command
----     -------
number   lookup inventory <number> $ROW $COLEND
Item     unique Item % <Item> <number>
Amount   valid.chars <Amount> "[0-9]"
Value    @!echo "2k <Amount> <Cost> * 1.25 *p" | dc
```

This validation file looks complicated, but only because it's used to give examples of what we've discussed. Suppose we want to look up the description for an inventory item, and place it next to the record number. The first field, **number**, is validated by running the program **lookup** with four arguments: first, the file name in which we want to look; **<number>** is replaced by the contents of the **number** field, and **$ROW** and **$COLEND** become the location on the screen of the first available white space after the viewable window over the **number** field as specified in the screen file. Bear in mind that **$COLEND** becomes the last column on the screen if there's no screen file, so it shouldn't be used without a screen file.

The lookup program moves the cursor to the specified position and prints the **Description** field:

```
#!/bin/sh
FILE=$1 NUMBER=$2 ROW=$3 COLEND=$4
cursor $ROW $COLEND
echo $NUMBER |
search -ms $FILE number |
column Description |
headoff
```

This guarantees uniqueness. Here's the unique program:

TUTORIALS

```
#!/bin/sh
COLUMN=$1 FILE=$2 VALUE=$3 NUMBER=$4
FAIL=1 SUCCESS=0
exec 1>&2
if test -z "$VALUE"
then
       echo  -n $COLUMN may not be empty
       exit $FAIL
fi
FOUND=‘row "$COLUMN == \"$VALUE\" && number != $NUMBER" < $FILE | \
       column number | headoff‘
if test ! -z "$FOUND"
then
       echo  -n "DUPLICATE $COLUMN $VALUE in $FILE (number $FOUND)"
       exit $FAIL
fi
exit $SUCCESS
```

In all these commands, the standard error is directed to the status line. That's the purpose of redirecting the standard output to the standard error with the first **exec** command. The algorithm to determine uniqueness looks to see that there's no other record with this *value* besides the current one.

One of the most common validation procedures is to limit the allowable characters. The following script accepts a string to test, and a regular expression in the standard form expected by the **tr** command of valid characters:

```
#!/bin/sh
exec 1>&2
VALUE="$1"; RE="$2"

SUCCESS=0 FAIL=1
LEFT=‘echo $VALUE | tr -d "$RE"‘

if test -z "$LEFT"
then
       exit $SUCCESS
else
       echo $LEFT from  $VALUE is not "$RE"
        exit $FAIL
fi
```

The effect of this command is to delete the allowable characters from the field, and if there's anything left over, they must be invalid characters, and the script exits with a FAIL status. Similarly, the script for specifying invalid characters deletes the complement of the invalid characters (the ones that are *not* invalid), and if there's anything left over exits with a FAIL status.

The validation command for the **Value** column runs the **dc** command to calculate **Value** on the basis of **Amount** and **Cost**, putting the calculated result into the field. This is how you compute one field based on the value of one or more others.

To illustrate how the '#' can be used in the validation file, let's consider the case where we need to update other tables *if and only if* we actually make a change in the current table — say for auditing purposes. Certainly we do not want to have to check each field as we pass through it — it is costly and there is no guarantee that the user will commit the changes made on the screen to the table. We only know and can only timestamp and record changes as the user make that commitment by *writing* the row. Therefore, we use the '#' validation key, which wakes up and performs the indicated operations only when records are *committed* (changed, deleted, or added) to the disk. Because auditing does not apply to a particular column, we can link it to any column in the validation file that doesn't require other validation.

We'll use a customers table as an example and we'll keep track of changes in contacts and their phone numbers. First, the table layout:

```
cust_id name    company street  city    st      zip     phone
------- ----    ------- ------  ----    --      ---     -----
```

And its verification table:

```
column  command
------  -------
cust_id @lookup customer "<cust_id>"
name    #audit hist o "<@name>" "<@phone>"; audit hist n "<name>" "<phone>"
company #putkey customer "<cust_id>" "<@cust_id>" "<company>"
city    !exchange "<city>" "Santa Cruz"
st      @lookup st "<st>"
```

The **audit** command takes an output file name argument (**hist**), an initial key argument, (**o** for *old* values and **n** for *new* values), and selected columns to be recorded. Thus, whenever we write a row, **audit** gets the **o** key and the original contents of the name and phone columns; then we hand audit an **n** key, with the current contents of the **name** and **phone** columns. Let's see what **audit** does:

```
#!/bin/sh
USAGE="$0 audit_file key <[ @ ] column>... "

AUDITFILE=$1
shift

echo -n ‘date +%y%m%d%H%M%S‘ >> $AUDITFILE
echo -n $1 >> $AUDITFILE
shift

while [ $# -gt 0 ]
do
      echo -n "    $1" >> $AUDITFILE
shift
done
echo "" >> $AUDITFILE
```

Another good example of when to use the '#' form of shell validation is when you're updating hash keys and decodes created by **jvindex**. This is another time when you only want to record changes which have been committed, rather than contemplated. The **putkey** command updates hash keys in **jvindex**'ed look-up tables. By executing **putkey** as key columns are modified, we are able to maintain a fast and up-to-date index as we edit. In the example above, **putkey** replaces the original contents of the **cust_id** column (**<@cust_id>**) with the current contents of the **cust_id** column (**<cust_id>**) in the **customer** look-up table. The **jvindex** command which created the customer look-up table is:

```
jvindex customers -ocustomer cust_id : company
```

Similarly, the lookup program used to display the **cust_id** calls **getkey** to locate the key column **cust_id** in the lookup table **customer**. Here's what this version of lookup does:

```
#!/bin/sh

if [ $# -lt 2 ]
then
      exit 0
fi
```

TUTORIALS

```
TABLE=$1; KEY="$2"

DECODE=‘getkey $TABLE $KEY‘
STATUS=$?

if [ $STATUS -gt 0 ]
then
       if [ "$DECODE" != "" ]
       then
              cursor $ROW $COLEND
              echo -n "$DECODE"
       fi
       exit 0
else
       exec 1>&2
       echo "$KEY: no such $TABLE"
fi
exit 1
```

The **exchange** script substitutes the last argument for the first if the first argument is blank, and puts the result in the **city** column. For example, pressing **<Return>** at the **city** column will cause exchange to insert the words **Santa Cruz**. This is handy for columns which are often, but not always, a single value. Here's what **exchange** looks like:

```
if [ "$1" = "" ]
then
       echo $2
else
       echo $1
fi
```

Control-Key Mapping

Control-key mapping allows you to define a set of **ve** commands or shell commands to be executed when you strike the specified control key. Control-key definitions are usually tailored to reflect often-repeated command sequences used by particular applications.

Both edit commands *and* shell commands may be combined for execution by a single control key. For example, let's say you want a command to display a particular file (an image, perhaps) during a **ve** edit session, and then refresh your screen when you are finished examining it. Your **.verc** file might look like this:

```
CTRL-key   command
--------   -------
p          !-pic part.sn112 | preview!S
```

In the command above, the first '!' means that a shell command follows; the '-' tells **ve** to be silent, forcing the suppression of

```
Hit RETURN to continue
```

messages and shell newlines that roll your screen up. The remainder of the line, up to a newline or another '!', (whichever comes first) is handed directly to the shell. All information after the '!' are **ve** edit commands. In this example, 'S' will redraw the screen.

The '%' symbol in the mapped command means *substitute the name of the current database at this position.* It works both interactively and in **.verc** shell commands. Try typing the sequence **:!cat %** next time you're editing with **ve**.

You can pass column contents from the record being displayed by **ve** to the shell via the **.verc** file. The syntax is similar to that used by the **report** program: column names enclosed in '<' and '>' are replaced by the contents of the column when the command is executed. We can illustrate a simple solution to the "long text field" problem. In this example, **ve** is used to maintain a table of bug reports:

```
number    date  customer        staff   product
------  ------  --------        ------  -------
     1  891212  cray            evan    /rdb
     2  891107  sun             wright  ve
     2  891107  sun             wright  ve
```

Because the description of bugs and fixes might be a few paragraphs, we do not want to confine it to a column in a table. How, then, can we tie it to our data base and manipulate it from **ve**?

We use the contents of the (unique) number column as the name of a text file which contains the bug description. To keep these files in an orderly manner, we will collect them in a separate directory and use the name of the data base to name the directory. To distinguish between the name of the table and the name of the directory of reports, we'll capitalize the first letter of the directory name (**Bugs**). Because the reports will be open-ended text files, we will use a text editor to enter and modify their contents.

All of this can be accomplished by mapping a control key to a shell command which edits the desired file name in the appropriate directory:

```
CTRL-key   command
--------   -------
e          !vi 'echo % | cap'/<number>\n
```

If we are on record number 43 and we hit **<ctrl-B>**, '%' will be replaced by **bugs**; **<number>** will be replaced by **43**; the shell will evaluate

```
'echo bugs | cap'
```

to be **Bugs**, and the resulting path name, **Bugs/43** will become an argument to **vi**.

Finally, use **<ctrl-?>** to see the contents of your **.verc** file from **ve**.

The approach of allowing broad shell access from within **ve** opens the door to all kinds of powerful and time-saving data base routines that can be designed and implemented by relatively naive users. The following rules should be followed to avoid pitfalls and confusion:

1. Do not put spaces between the enclosing angle brackets and the column name when you are specifying a column name for content substitution. This will cause ve to assume that '<' is *redirect the standard input* and '>' is *redirect the standard output*, and the column name will be passed to the shell uninterpreted.
2. Use '\' to send the '%' symbol through to the shell without being replaced by the name of the data base.
3. Use '-' after the '!' in shell commands to suppress the normal messages and newlines issued by **ve** and the shell. This is handy when you don't want to disturb the current screen or window — for example, if you simply want to pop up a subwindow in an unused portion of your screen and you do not wish **ve** to redraw over it.
4. Use '!' to ensconce shell commands between edit commands.

Command-line Options

ve recognizes the following command-line options:

TUTORIALS

-d This was implemented to prevent accidental deletion of rows, both by the **dd** command, and by attempting to write a record in which each column has been deleted.

+<*cmd*>
A more flexible way to start up **ve** with a particular command (as opposed to the **-m** or mode option) is with the '+' command. The remainder of the '+' argument is handed directly to **ve**'s command parser. Of course, because it is part of a shell command, it must be constructed to escape shell interpretation. An example of this is a command which searches for the word **sun** in the bugs table and appends the word **shine** to it:

```
ve bugs +/sun\\eAshine\\e
```

The **\e** at the end of **sun** ends the search pattern; the **\e** at the end of **shine** exits insert mode. The extra **\'s** are swallowed up by the hungry shell.

Screen Size and Column Limits

The column limit of 66 fields has been expanded to 512.

ve screens can handle up to 158 columns by 64 rows. The size of the window in which ve is executed determines how ve sets up default screens and uses blank space. To override the absorption by **ve** of available window space, screen files should be terminated by a final blank row. This row is interpreted by ve as the vertical position, after which the message line is to be displayed.

User Column in Audit Files

The login name of the user is now recorded in the second column of the audit file. Existing audit files must be modified to reflect the addition of the new column prior to appending rows with the new version of **ve**.

Fast-Access Indexing From ve

Hash look-up on columns has been implemented for **ve**. Key columns are associated with the row number stored in the **-i** file, effectively linking searches on keys to the **G** command. The following rules apply for initiating key searches:

1. The key column must be marked with the **m** command prior to specifying the search pattern.
2. Column contents must contain unique information.
3. The entire column must be entered in order for a key search to be initiated (which makes it a good idea to keep key columns short).

Key columns are created with **vindex** by preceding the column name with the **-k** switch. For example, to key the number column of the **bugs** table and have it decode to the customer column:

```
vindex bugs -k number : customer
```

Searching for key columns is done in the ordinary way: simply precede the search pattern with '/'. If it matches a hash key, the row is retrieved by it's indexed position in the table. Otherwise, row-by-row pattern matching is performed. The method **ve** uses to find hits is not obvious to the user (except in the case of large tables, where indexed searching is significantly faster) and requires no additional expertise.

What Is the Cost of Indexed Searching?

When the last user exits an edit session on a table, the **squeeze** program is executed in the background to clean up the table, putting records back in entry order and removing the holes left by deleted records. This process reconstructs the **table-i** file, which keeps track of row positions. Because indexed keys also use this information, they too must be updated after the table has been "squeezed." This process happens automatically when **squeeze** is finished. The cost is the amount of time it takes to re-**vindex** each of the key columns for the entire table.

To protect the table during the squeeze and vindex processes, a lock file (*table***+lck**) is created. When both the squeeze and vindex processes are complete, the *table***+lck** file disappears. It is important not to tamper with the table until the lock file is gone. Depending on the size of the table and the number of indexed key columns which must be updated, this can take a few minutes. So, the more keys you create, the longer it will take to put it all back together after everyone has exited. For transaction processing, the best strategy is to keep the table active (at least one **ve** user) during busy times.

TUTORIALS

Data-Base Design

When you try to design your first data base, you are likely to be quite confused. Which data go into which tables? Can you have just one big table? Should you have lots of little tables? How do you find out? It turns out that there is a simple way to decide how to lay out your data base. Once you learn it, any application will be easy. This chapter will show you how.

One-to-One Relationships in One Table

When you look at your data, you will notice that some information has what we call a one-to-one relationship. For example, each person has a first name, a last name, a birthdate, a sex, an ID number, and so on. All of these items of information can be put into one table. Here is such a table that is called **employee**:

```
Id      First      Last      Birth      Sex
--      -----      ----      -----      ---
1       Howard     Ho        450503     Male
2       Jane       Dobbs     540129     Female
```

The rule then is put all one-to-one relationships into a single table. Another example is **department**:

```
Dept      Name      Head      Address
----      ----      ----      -------
1         act       Jones     Basement
2         sales     White     4th Floor
```

One-to-Many Relationships in Two Tables

Another relationship is one-to-many. People have a one-to-many relationship with their children. A person can have from zero to many children. Relational data base theory insists that one-to-many relationships cannot be put into one table, but require two. In addition to the **employee** table, you will need a **child** table:

```
Parent      Name
------      -----
1           Sally
1           Lynn
```

Note that Howard Ho (Id 1) has two children, Sally and Lynn. However, Jane Dobbs (Id 2) has no children, because her Id (Parent) number is not in the **child** table. To put this information back together, use the **jointable** command:

```
jointable employee child
```

This yields:

```
Id      First      Last      Birth      Sex      Name
--      -----      ----      -----      ---      -----
1       Howard     Ho        450503     Male     Sally
1       Howard     Ho        450503     Male     Lynn
```

Here we have the two tables, **employee** and **child,** joined together on the **employee Id** and the **child Parent** key columns. These connect the two tables for the **jointable** command.

Note that since Howard Ho has two children, he has two rows in the table. Also note that Sally Dobbs was not included in the table because she has no children.

You can begin to see why we need to keep these kind of data in separate tables. We need information on employees, even when they have no children. Of course, we could include Sally Dobbs and leave her children column empty, but that will complicate processing.

More seriously, we have all of the information on Howard Ho repeated for each child. This not only wastes space, but requires that when we update the table we have to find every entry for Howard Ho and correctly make the update. The extra time and effort, plus the risk of errors, make the single table approach unacceptable. Therefore, we must keep one-to-many relationships in separate tables and join them together only when needed.

Many-to-many Relationships in Three Tables

There are also several many-to-many relationships. For example, students take zero to many courses and courses have zero to many students. This relationship requires three tables.

First we need a **student** table:

```
Student     First     Last     Year
-------     -----     ----     ----
      1     Jim       Clark       2
      2     Mary      Witte       1
```

We also need a **course** table:

```
Course     Credit     Room     Day     Time
------     ------     ----     ----    ----
art-1a          3     RB-8     MWF     10am
chem-1          4     HA-18    TTh     2pm
```

To connect these two tables, a third table is needed that we will call **course.student**:

```
Course     Student
------     -------
art-1a           1
chem-1           1
chem-1           2
```

We can join these tables together with two **jointable** commands. First let's use the **jointable** command to see individual joins:

```
jointable course course.student
```

This yields:

```
Course     Credit     Room     Day     Time     Student
------     ------     ----     ----    ----     -------
art-1a          3     RB-8     MWF     10am           1
chem-1          4     HA-18    TTh     2pm            1
chem-1          4     HA-18    TTh     2pm            2
```

Next, the command

```
jointable -j Student student course.student
```

yields:

TUTORIALS

```
Student     First     Last     Year     Course
-------     -----     ----     ----     ------
      1     Jim       Clark       2     art-1a
      1     Jim       Clark       2     chem-1
      2     Mary      Witte       1     chem-1
```

All together now:

```
       jointable -j Student student course.student | \
       jointable -j Course - course
```

which gives us:

```
Student  First  Last  Year  Course  Credit  Room   Day   Time
-------  -----  ----  ----  ------  ------  ----   ----  ----
      1  Jim    Clark    2  art-1a       3  RB-8   MWF   10am
      1  Jim    Clark    2  chem-1       4  HA-18  TTh   2pm
      2  Mary   Witte    1  chem-1       4  HA-18  TTh   2pm
```

So we can bring it all together when we want to see it, but we keep many-to-many relationships in three separate tables.

Planning

When you look at your application, look for the kind of relationships that you have and group your data accordingly. Continuing the college example above, we will also need tables for instructors, rooms, and so on. There is a one-to-many relationship between instructors and classes: instructors often teach several classes, but classes have one instructor (unless there is team teaching).

Note that you have to think of exceptions. If the exceptions are very rare, you might choose to ignore them. But if there are more than a rare instance of classes having more than one instructor, it becomes a many-to-many relationship. Then you will need three tables, because you will need a connector table.

Our college data base will also need a **room** table. What is the relationship between rooms and courses? Can you see that it is one-to-many? A course is in only one room. (Is it always? What about labs? Or are labs separate courses?) But rooms have more than one course in them at different times, unless we have a terribly inefficient college. So how many tables will we need? One-to-many requires two tables. Which tables do we need? In this case a **course** table and a **room** table. The **course** table will have the room number in it, but the **room** table will have no mention of the courses in it because there are many.

To find the size of the room for each course, join the **room** table with the **course** table:

```
Course      Credit      Room      Day      Time      Teacher
------      ------      ----      ----     ----      -------
art-1a           3      RB-8      MWF      10am            3
chem-1           4      HA-18     TTh      2pm             1
cs-101           4      DB-1      MWF      2am             2
econ-1           3      RB-8      TTH      1pm             2
his-10           3      HA-18     MWF      11am            3
```

And we need a **Room** table:

```
Room      Type      Size      Hall
----      ----      ----      ----
DB-1      Lecture    250      Daddy Bucks
HA-18     Lab         17      Hillary Addler
RB-8      Tacky       23      Roberta Bucks
```

Then, use the commands **sorttable** and **jointable** to combine them:

```
sorttable Room < course |
jointable -j Room - room |
column Course Room Size > room.size
```

This creates the table **room.size**, as follows:

```
Course     Room     Size
------     ----     ----
cs-101     DB-1      250
chem-1     HA-18      17
his-10     HA-18      17
art-1a     RB-8       23
econ-1     RB-8       23
```

So, it turns out that designing a data base is easy. Just practice thinking about these principles with different applications.

Normalization

The approach we have just discussed is the easiest way to think about designing your data base. However, most of the data base literature approaches the problem from a different angle. To help you understand that literature, these other approaches are discussed in the following sections. It is more technical and can be skipped on first reading, or if you are just beginning to use data bases. In the technical relational data base literature, this process of correctly grouping data into tables is called *normalizing*.

Functional Dependency

Before discussing normalization, several concepts must be understood. *Functional dependency* refers to whether the data in one column determine the data in another. In other words, if you know the data in one column, can you tell what the data in another column will be? If the data in one column are repeated, are the corresponding data in the other column also repeated? If so, there is a functional dependency, otherwise not.

For example, look at these two tables. First, the **room.size** table we previously created:

```
Course     Room     Size
------     ----     ----
cs-101     DB-1      250
chem-1     HA-18      17
his-10     HA-18      17
art-1a     RB-8       23
econ-1     RB-8       23
```

Note that when a value in the **Room** column is repeated, the corresponding **Size** value is also repeated. (See **RB-8** is in the **Room** column with the same **Size** of 23. Also the same for HA-18 and 17). Therefore, **Room** functionally determines **Size** or an equivalent way of saying it, **Size** is functionally determined by **Room**. Ordinarily, we do not want to have tables with functional dependencies in them. It is acceptable here, because we used a join to put this table together. It is a query of our data base and not a proper table of the data base.

Keys

Key columns are the unique identifiers of each row in the table. The key column or columns of a table should always determine each of the other columns of the table. The function of a key is to uniquely identify the data in the other columns. So the rule is that only key columns can functionally determine other columns. Nonkey columns should not functionally determine other columns in a normalized data base.

TUTORIALS

Universal Relation

The opposite of normalization is the universal relation. Imagine one large table that has all of the tables joined together. It has all of the columns of all of the tables across the top. It is the worst case of a unnormalized data base. Why? What is wrong with such a table?

We need to avoid unnormalized tables because they create severe problems for us.

Redundancy Problems

One problem is that we have a lot of redundant data. Look at the **room.size** table again and note that the size information for each room is repeated:

```
Course     Room     Size
------     ----     ----
cs-101     DB-1      250
chem-1     HA-18      17
his-10     HA-18      17
art-1a     RB-8       23
econ-1     RB-8       23
```

This takes up more space than necessary and slows down the programs that must process it.

Update Problems

Another problem is the extra work and errors that result from trying to update such a table. Suppose we add more chairs to **HA-18**. We have to update the new **Size** twice. In this simple table, that is not much work, but in a large data base it is overwhelming. And what happens when we make an error in our updating? It is hard to find and correct. After a while, we will not know which conflicting value is correct. Our data base will become hopelessly corrupted.

Insert and Delete Problems

There are also insert and delete problems. Suppose we want to add a room and its size to this data base. If it does not yet have a course in it, we can't put it into the table unless we give it a blank course. This creates problems when searching and joining. How do you handle blank or null values?

Likewise with deleting. What if a course is still assigned to the room after the building has been demolished? You don't want to drop the course, but you need to delete the room from the data base.

First Normal Form

For relational functions to work, tables must be simple or, as it is called in the literature, normalized. There are a number of steps to simplifying, or normalizing, a table. Most importantly, you must never have a variable number of columns in a table. For example, in a file of college employees, you might be tempted to have a column for the first name of the children of employees:

```
Emp#        Name        Dept        Child
-------     -------     -------     -------
1           Martin      1204        Sally, Fido
2           Moore       1339
3           Mapes       1045        Jan, Shawn, Peter, Barbara
```

This creates many problems, not the least of which is that the relational functions will not work. The answer is to have two files, one for employees and another for their children. The **child** file would look like this:

```
Emp#     Child
----     -------
   1     Sally
   1     Fido
   3     Jan
   3     Shawn
   3     Peter
   3     Barbara
```

When you need to put the two files together, you do so with a join (**jointable** in **/rdb** because there is a COHERENT command named **join**, which works on files without headers). This is another way of saying that one-to-many relations must be represented in two tables. When all of the one-to-many relations are in separate tables, we say that the data base is in first normal form.

Second Normal Form

Both second and third normal forms require that nonkey functional dependencies be removed by creating additional tables. In the case of second normal form, we look for dependencies on columns within the key. Keys can consist of more than one column. For example, consider a **grade** table:

```
Course   Student  Grade
------   -------  -------
art-1a   1        3
art-1a   4        4
chem-1   1        3
chem-1   2        2
chem-1   3        2
cs-101   2        1
cs-101   4        4
econ-1   1        3
econ-1   2        3
econ-1   3        2
his-10   1        3
```

Note that it takes two columns to make a unique key for each row. Both the **Course** and the **Student** columns make up the key. **Grade** is a nonkey column. What we must look for is a dependency between one of the columns in the key and the other columns. In this case there are none, so this table is in second normal form.

But suppose we added a column for class rooms. Since the **Room** column depends upon the **Course** column, we would not be in second normal form. This is called a *partial dependency* because columns are dependent on a partial key, that is, less than all of the columns of the key.

Third Normal Form

Third normal form requires that we also eliminate any dependencies between nonkey columns. These are called *transitive dependencies*.

Consider, for example, the **room.size** table:

```
Course     Room     Size
------     ----     ----
cs-101     DB-1      250
chem-1     HA-18      17
his-10     HA-18      17
art-1a     RB-8       23
econ-1     RB-8       23
```

Here the **Size** column depends on the **Room** column. To reach third normal form we must make two tables: one called **course.room**, with the **Course** and **Room** columns; and the other called **room**, with the **Room** and **Size** columns.

TUTORIALS

Normalizing Example

Let's start with a simple universal table and normalize it step by step. Suppose we go to the dean of the college to work out the college data base for grades. Since the dean does not know about normalization, she lists all of the items she wants to keep track of, but our job is to put them into tables. The universal table she builds might look like this:

```
Course   Student  Grade  Teacher   Title     Child
------   -------  -----  -------   -----     -----
chem-1         1      3        1   Assist.   Sally, Mike, Joy
chem-1         2      2        1   Assist.   Fred
chem-1         3      2        1   Assist.
cs-101         2      1        2   Prof.     Fred
cs-101         4      4        2   Prof.
econ-1         1      3        2   Prof.     Sally, Mike, Joy
econ-1         2      3        2   Prof.     Fred
econ-1         3      2        2   Prof.
his-10         1      3        3   Assoc.    Sally, Mike, Joy
```

Look at this unnormalized table and see if you can normalize it before we show you how.

First normalization requires that we get rid of the one-to-many relationship between students and their children to remove multiple values in the **Child** column. We create a new table for children of the students:

```
Student     Child
-------     -----
      1     Sally
      1     Mike
      1     Joy
      2     Fred
```

We could add more columns about each child to this table as long as we are careful not to include information about the student-parent which should be put into the **student** table. We also remove the **Child** column from the big table. The command **column** accomplishes this task:

```
column Course Student Grade Teacher Title < universe
```

This yields:

```
Course   Student  Grade  Teacher   Title
------   -------  -----  -------   -----
chem-1         1      3        1   Assist.
chem-1         2      2        1   Assist.
chem-1         3      2        1   Assist.
cs-101         2      1        2   Prof.
cs-101         4      4        2   Prof.
econ-1         1      3        2   Prof.
econ-1         2      3        2   Prof.
econ-1         3      2        2   Prof.
his-10         1      3        3   Assoc.
```

For second normal form we have to realize that we have a multi-column key: **Course** and **Student**, which uniquely determines each row. Let's look for dependencies between either of these columns and the other columns. As you can see, **Course** determines **Teacher**. So, let's use **column** to create a new **course.teacher** table:

```
column Course Teacher Title < universe > course.teacher
```

This table appears as follows:

```
Course     Teacher     Title
------     -------     -----
chem-1           1     Assist.
chem-1           1     Assist.
chem-1           1     Assist.
cs-101           2     Prof.
cs-101           2     Prof.
econ-1           2     Prof.
econ-1           2     Prof.
econ-1           2     Prof.
his-10           3     Assoc.
```

And, then use **column** to create a table called **grade**:

```
column Course Student Grade < universe > grade
```

This yields:

```
Course     Student     Grade
------     -------     -----
chem-1           1         3
chem-1           2         2
chem-1           3         2
cs-101           2         1
cs-101           4         4
econ-1           1         3
econ-1           2         3
econ-1           3         2
his-10           1         3
```

Note that there are many redundant rows in the **course.teacher** table. We can reduce the table to unique rows by using the COHERENT **uniq** command to remove duplicate rows:

```
uniq < course.teacher
```

This yields:

```
Course     Teacher     Title
------     -------     -----
chem-1           1     Assist.
cs-101           2     Prof.
econ-1           2     Prof.
his-10           3     Assoc.
```

The **grade** table is now in second and third normal form, but the new **course.teacher** table is only in second normal form. Note that **Title** depends upon **Teacher.** We need another table; so we'll use **column** to create it, as follows:

```
column Teacher Title < course.teacher | uniq > teacher
```

The new table, called **teacher**, contains the following:

```
Teacher     Title
-------     -----
      1     Assist.
      2     Prof.
      3     Assoc.
```

Now, we'll use **column** to check the contents of **course.teacher**:

```
column Course Teacher < course.teacher
```

This writes the following onto the screen:

TUTORIALS

```
Course     Teacher
------     -------
chem-1           1
cs-101           2
econ-1           2
his-10           3
```

So now the **teacher** and **course.teacher** tables are in third normal form. How many tables do we have? What are their names? What columns are in each table? How would you explain to the dean what you have done to her nice big table? Why have you done it?

Complex Queries with Joins

When data are distributed to many tables, how can we get the information we want? It is easy to project columns and select rows from a single table and join two tables. But what if the information we want is widely separated?

As an example, let us imagine that you are an instructor in a college and want a phone book of all of your students, just in case you need to call to tell them that the class was canceled or that the final exam date or room had changed. Suppose all you know is your own name and the different tables in the data base:

Table of College Data Base Tables

```
Table           Columns
-----           -----------
teacher         Teacher First Last Address City State Phone Title
teacher.course  Teacher Course
course          Course Credit Room Day Time Teacher
course.student  Course Student
student         Student First Last Year Phone
```

Such a phone book or listing can be produced with a single pipeline. First let's think through the solution. Then we will build the one-line query.

You know your name, so you can find your ID number in the **teacher** file. With your teacher ID number, you can get a table of all of your class ID numbers from the **teacher.course** connector file. Then you can pick up the student ID numbers from the **course.student** connector file. Finally, you can project the information you want about each of your students.

Pipeline Join

Now let's look at the pipeline. It has been written down the page to make it easier to read and modify. When the Bourne and Korn shells sees the pipe symbol at the end of a line, they know to continue to the next line because this line is not finished:

```
row 'First == "Joy" && Last == "Xi" ' < teacher |
column Teacher |
tee tmp1 |
jointable - teacher.course |
column Course |
sorttable |
tee tmp2 |
jointable - course.student |
column Student Course |
sorttable |
tee tmp3 |
jointable - student |
column Course Last First Phone |
sorttable > phonebook
```

This probably looks overwhelming, but don't worry, we are going to go through each part.

First, let's look at the output so that we will know where we are going with all of this.

```
Course     Last     First     Year     Phone
------     ----     -----     ----     --------
cs-101     Dunce     Boris       2     765-4321
cs-101     Early     Mary        4     123-4567
econ-1     Early     Mary        4     123-4567
econ-1     Farkel    Freddy      3     123-1234
econ-1     Knott     Why         1     123-7654
```

Note that the command

```
tee tmp
```

has been inserted in three places in the pipeline. The COHERENT **tee** command writes the standard input to the standard output, but also write the stream of characters into the file *tmp*. By looking at the *tmp* files, we have a peephole into the pipeline so that we can see how the data look at each point. Therefore, we can see and discuss the data at three points in the pipeline.

tmp1

Assuming your name is "Joy Xi", you can select the row in the teacher file in which the first name, **First**, is "Joy" and the last name, **Last**, is "Xi". Then you can project the **Teacher** ID number column only. **column** throws away the other columns. This is to reduce the amount of data flowing through the pipe to speed up the execution. It also avoids conflicts in the names of columns:

```
row 'First == "Joy" && Last == "Xi" ' < teacher |
column Teacher |
tee tmp1 |
```

Now let's see what the data look like by looking at the **tmp1** file:

```
Teacher
-------
      2
```

Here we have gotten the **Teacher** ID number in a little table that we can now use to find more data.

tmp2

The next four lines join the single-line table above with the **teacher.course** table to pickup the courses that "Joy Xi" teaches. Then only the **Course** column is projected. It must be sorted for the next join:

```
jointable - teacher.course |
column Course |
sorttable |
tee tmp2 |
```

By taking a look at our data now, we see that "Joy Xi" teaches two courses:

```
Course
------
cs-101
econ-1
```

TUTORIALS

tmp3

Now we can join this two-row table to the **course.student** table to pick up the students in her classes. This time we project not only their ID numbers but the courses because we want that information carried on to the phone book. The **Student** column also must be sorted for the next join. We have to tell the **sorttable** command that the **Student** column is numeric, instead of string, to get a correct sort:

```
jointable - course.student |
column Student Course |
sorttable -n |
tee tmp3 |
```

Here is the output of these lines:

```
Student     Course
-------     ------
      1     econ-1
      2     cs-101
      2     econ-1
      3     econ-1
      4     cs-101
```

Note that we have only the **Student** and the **Course** columns, and that the **Student** column is correctly sorted.

Phone Book

The last lines join this table above with the **student** file to get the student information. Then the columns we want for the phone book are projected. Note that we project the **Course** column that was carried along as well as columns from the **student** table. We must sort the new phone book by course so that the teacher can see all of the students in one class:

```
jointable - student |
column Course Last First Phone |
sorttable > phonebook
```

Let's see the final output phone book again:

```
Course     Last     First     Year     Phone
------     ----     -----     ----     --------
cs-101     Dunce     Boris       2     765-4321
cs-101     Early     Mary        4     123-4567
econ-1     Early     Mary        4     123-4567
econ-1     Farkel    Freddy      3     123-1234
econ-1     Knott     Why         1     123-7654
```

Note that we have done this in such a way that some students' information is repeated if they are taking more than one class from this teacher. Mary Early, in the table above, is an example. We accept this repetition so that the teacher can call all of the students in a particular class. If we had just a list of all the students, we would not know which student was in what class. But you can change this example many different ways by changing the **column**, **sorttable**, and **jointable** commands.

TUTORIALS

Shell Programming

The great advantage of the COHERENT system is that most of the work is done for you. Most data base systems create a new language that you must learn, but **/rdb** uses COHERENT programs and the shell programming language. If you already know these COHERENT tools, there is little more to learn. If you don't know the COHERENT system, it is much better to learn the general-purpose COHERENT system than a special language that is only good for a single data base package. COHERENT tools as they are delivered on your system can do much of the data base application. **/rdb** only extends the COHERENT system by adding more than a hundred commands that will make data base handling easier and faster.

Data-Base Programming in COHERENT Shell Language

It is the shell programming language that make this approach so easy and powerful. The shell is the user interface to the COHERENT system. It prompts you for commands, rewrites your command line to save you typing, and executes the programs you request. It is extremely simple to learn. You start by putting commands that you would type at the terminal, into files called shell programs. These files can be executed simply by typing their names.

But the shell is also a powerful string-oriented programming language with control flow statements like **if**, **for**, and **while**. You can mix procedural with nonprocedural statements.

Procedural programs are the traditional step-by-step instructions that tell the computer *how* to do something. *Nonprocedural* statements simply tell the computer *what* to do. For example, a traditional procedural program will tell the computer how to sort a file. A nonprocedural statement in the COHERENT system would be

```
sort file
```

where *file* is the name of the file we want sorted. The COHERENT **sort** command knows how to sort a file, so we get what we want with out worrying about details. If we had to write the sort program, it would take ten to 100 lines of code, or more. Nonprocedural is much easier and saves a lot of time. It is also much faster to fix. Options to the **sort** program can reverse the sort. Editing a sort program is much trickier.

Of course, nonprocedural is best, but if a system is all nonprocedural, it has to anticipate everything you will ever want to do! Most fourth-generation systems have this problem: they give you a lot of functions, but lack procedural language statements to let you program what they don't have. Third-generation languages let you program anything, but force you to carefully code too many basic functions over and over. The COHERENT system gives you a nice mix of predeveloped programs that cover all of the basic actions, and the shell programming language to allow you to code more complex functions by building up basic functions.

In the COHERENT environment, you are constantly in a process of automating your work. First, you try out a simple program. Then call it, and other programs, from a higher-level program. Soon you are able to do many things with simple keystrokes.

There are a number of books on shell programming, in addition to the COHERENT documentation. Before we discuss shell programming further, we will need some COHERENT programs to build applications.

COHERENT Utilities

In addition to the **/rdb** commands, there are several COHERENT utilities that you will probably need often. Your COHERENT documentation provides detailed descriptions of these commands; we briefly describe some of their data base uses.

awk: Language to Produce Complex Reports

awk is the heart of several **/rdb** commands. **awk** provides the engine for the **row**, **compute**, and **validate** commands. It can be used by itself to do powerful things to tables. For more advanced applications, you must refer to the **awk** Lexicon entry and tutorial.

cat: Display a Table or List File

cat is used all the time to list out your tables.

echo: Repeat a Statement

This command repeats text. It lets you easily pass arguments to command and expand environmental variables.

grep: Find All Rows That Contain a Given String

grep lets you search a table or any text file for a string of characters. If a match is found, the line is printed. It is a primitive search, but easy to use.

od -c: Octal Dump All Bytes as Characters

od lets you see special characters and can be used to see if your tabs are in place. (The **/rdb see** command is better for this purpose.)

sed: Stream Editor to Edit File in a Pipe

sed is a very powerful program for editing your tables in a pipe stream. Many of the **/rdb** shell programs use it. You can perform almost any editor command with it.

sh: COHERENT Shell Programming Language

sh is the COHERENT shell program that interfaces to the user. It prompts with a dollar sign ($) or other prompt, then reads the commands that the user types. It is a powerful string-oriented programming language. You can create any complex application with it and the COHERENT and **/rdb** commands.

ksh is the other COHERENT shell programming language. It is, for the most part, a superset of **sh**. It differs from **sh** mainly in that it lets you recall, edit, and resubmit command that you had typed earlier, using MicroEMACS-style editing commands. See the COHERENT manual's Lexicon entries on **sh** and **ksh** for descriptions of each language, and how they differ.

spell: Check Spelling in a Table or List File

You may find this quite useful for checking spelling in a file. It is also a good example of the power of shell programming.

tail: Display Bottom Rows of a Table or List File

tail outputs the end of a file. You can use it to "behead" a table. Consider table **mailtable**:

```
for the same purpose.
Number  Name             Company        Phone
------  ---------------  -------------  --------------
     1  Ronald McDonald  McDonald's     (111) 222-3333
     2  Chiquita Banana  United Brands  1234
```

The command

TUTORIALS

```
tail +3 mailtable
```

yields:

```
1  Ronald McDonald  McDonald's     (111) 222-3333
2  Chiquita Banana  United Brands  1234
```

As you can see, the first two "head" lines of the table have been thrown away.

There is also the **/rdb** command **headoff**

wc: Word Count

This command counts the characters, words, and lines in a file.

Text Editor to Enter and Update Files

A text editor is important to your work, because you will use it to set up tables and lists and to write your shell program applications. COHERENT comes with three interactive text editors:

ed This is a powerful, line-oriented editor. This editor is best for making mass transformations of a file, because you can use *normal expressions* to tell it what to do. **ed** is also useful in editing files that are too large for a screen-oriented editor to handle, at least in the 80286 edition of COHERENT.

me This is the MicroEMACS editor. It is the preferred editor of most COHERENT users. As its name implies, it is a smaller, more efficient version of Richard Stallman's famous EMACS editor. With it, you can display multiple files in multiple windows upon your screen, move text from one file or window to another. Unlike **ed**, MicroEMACS lets you address the text to be edited by moving the cursor to it — hence, it is considered to be a "screen-oriented" editor.

vi This is the COHERENT implementation of the UNIX standard editor. It has features of both **ed** and MicroEMACS, plus many features of its own.

Each editor is summarized at length in the COHERENT Lexicon. The COHERENT manual also contains tutorials for **ed** and **me**. Any of these three editors will do the job for you; which you select is largely up to your personal preference. Each editor has its strengths and weaknesses; an editing job that takes hours with one editor might take only minutes with another. You would do well to learn at least the riduments of each editor, so you can select the right editor for a given editing job.

Reading and Writing Data Base Files

When you write programs, you have to open files. In the COHERENT environment, files usually come to the program from standard-in and output is sent to standard-out. But sometimes you have a shell program that is talking to the user at the terminal through the standard-in and -out, but has to open one or more files for reading and writing. This is done with the **exec** command. This code opens *filein* for input (read) and assigns the file descriptor 3. *fileout* is opened for output (write) and assigned file descriptor 4. Then programs can use those file descriptors with the **<&3** and **>&4** conventions:

```
exec 3< filein
exec 4> fileout
cat <&3 >&4
```

The COHERENT shell's **read** command cannot have its input redirected to a file other than standard-in, except by a trick:

```
exec 0< inventory
read HEAD
```

Here, we have set the environmental variable **HEAD** to contain:

```
Item   Amount Cost   Value  Description
```

In a loop you can read a whole file line by line. The **while** statement is a control statement that executes the lines within the **do** and **done** until **read** tries to read the end-of-file and returns a status code indicating false.

```
exec 0< inventory
while read HEAD
do
       echo "$HEAD"
done
```

This yields:

```
Item   Amount   Cost   Value   Description
----   ------   ----   -----   --------------
   1        3     50     150   rubber gloves
   2      100      5     500   test tubes
   3        5     80     400   clamps
   4       23     19     437   plates
   5       99     24    2376   cleaning cloth
   6       89    147   13083   bunsen burners
   7        5    175     875   scales
```

Parsing Rows

You can use the **set** command to parse strings in the shell. For example, if you want to get at any field in a row you can read the row into a shell variable and set the words in the row to the shell positional arguments **$1** and **$2**:

```
exec 0< inventory
read HEAD
set $HEAD
echo "The first field is $1 and the fifth field is $5."
```

The final command, **echo**, displays its first and fifth arguments. This prints the following:

```
The first field is Item and the fifth field is Description.
```

Tables to Shell Variables

It would be very nice if we could use the column names from a table to refer to the column values in a shell program. Often people assume they have to walk through the row with **for** and, perhaps, **shift** commands. This slow method is not necessary. Here is a neat trick for using column heads as variables in shell programs.

Consider, to begin, table **inventory**:

```
Item   Amount   Cost   Value   Description
----   ------   ----   -----   --------------
   1        3     50     150   rubber gloves
   2      100      5     500   test tubes
   3        5     80     400   clamps
   4       23     19     437   plates
   5       99     24    2376   cleaning cloth
   6       89    147   13083   bunsen burners
   7        5    175     875   scales
```

Now consider the shell script **onhand**:

TUTORIALS

```
USAGE='usage: onhand Item < inventory'
FOUND=0
NOFOUND=1
read HEAD
read DASH
while read $HEAD
do
       if test "$Item" -eq "$1"
       then
              echo We have $Amount $Description on hand.
              exit $FOUND
       fi
done
exit $NOFOUND
```

Now, the command

```
onhand 2 < inventory
```

yields:

```
We have 100 test tubes on hand.
```

Now, let's walk through **onhand** and see just what it does.

The first three lines

```
USAGE='usage: onhand Item < inventory'
FOUND=0
NOFOUND=1
```

set some environmental variables that are used for printing an error message should a mistake occur.

Next, the line

```
read HEAD
```

reads the first line of **inventory** into a shell variable called **HEAD**.

The next line

```
read DASH
```

reads the dash line into an environmental variable called **DASH**. We want to throw away the dash line, and this is an easy way to do it.

Next comes the **while** loop:

```
while read $HEAD
do
       if test "$Item" -eq "$1"
       then
              echo We have $Amount $Description on hand.
              exit $FOUND
       fi
done
```

Look carefully at the **read** statement. It examines **HEAD**, which the shell expands into the list of column names, *before* **read** is called. So **read** really sees all of the column names as arguments:

```
read Item Amount Cost Value Description
```

read assigns the first word it reads from standard-in to **Item**, then the next to **Amount**, and so on.

Therefore, it has automatically assigned the column values to the column names for us and we can use them in the shell program with **$** in front.

The **test** and **echo** commands both use the variables that hold the values from the current row. Let's see this with the shell execution trace option **-x** turned on:

```
+ read HEAD
+ read DASH
+ read Item Amount Cost Value Description
+ test 1 -eq 2
+ read Item Amount Cost Value Description
+ test 2 -eq 2
+ echo We have 100 test tubes on hand.
We have 100 test tubes on hand.
```

This lets you see how the column names and values are rewritten as each row is read, tested, and processed.

Lists to Shell Variables

Here is an example of how to get records from a list formatted file and make each column name-value pair a shell *variable=value* pair. It uses the **/rdb** program **listtosh** to convert the list record to shell format. Consider the list **maillist**:

```
Number              1
Name                Ronald McDonald
Company             McDonald's
Street              123 Mac Attack
City                Menphis
State               TENN
ZIP                 30000
Phone               (111) 222-3333
```

The command

```
listtosh < mail.list
```

yields:

```
Number='1'
Name='Ronald McDonald'
Company='McDonald\'s'
Street='123 Mac Attack'
City='Menphis'
State='TENN'
ZIP='30000'
Phone='(111) 222-3333'
```

See the tabs converted to equal signs '=' and the data protected absolutely by apostrophes ('). Also note that each apostrophe within the data is protected with a backslash '\'. This is the format for the shell *variable*=*value* assignments. This program can be executed in a shell program and the *variable*=*value* pair will become known to the program. The command:

```
column Name Phone < mail.list | listtosh
```

yields:

```
Name='Ronald McDonald' Phone='(111) 222-3333'
```

This one line output from the command substitution is exactly the format the shell uses to assign values to its variables. As you can see, the shell can read multiple assignments on one line. The apostrophes allow anything to be in the data without being expanded by the shell. In this example, we projected only **Name** and **Phone**, to keep from having a long line and to show that we can use

TUTORIALS

any command to get the variables we want.

In the next example, **eval** is used to rescan and assign the line:

```
eval ‘column Name Phone < mail.list | listtosh‘
echo "$Name’s phone number is $Phone."
```

This yields:

```
Ronald McDonald’s phone number is (111) 222-3333.
```

Now we can use the values of the variable in a shell program. This example contains an **echo** statement to see that the variable had been correctly assigned to the current shell, and not to some subshell.

The next example is more complex. A table is searched for a record and converted to list and then used in the shell program:

```
echo ‘row ’Item == 3’ < inventory | tabletolist | listtosh‘
eval ‘row ’Item == 3’ < inventory | tabletolist | listtosh‘
echo "We have $Amount $Description."
```

This yields:

```
Item=’3’ Amount=’5’ Cost=’80’ Value=’400’ Description=’clamps’
We have 5 clamps.
```

Report Writing

UNIX World magazine printed two articles by Alan Winston about fourth-generation programming languages (July 1986 and April 1987). Several competing companies were invited to produce a sample report using their 4GL systems. Most of these languages looked more like COBOL or RPG. Here are two ways of producing the sample report with the COHERENT shell and **/rdb**.

The first example is a simple default report (which we think looks better than the report format required in the article):

```
number   fname     lname      code   hours   rate   total
------   -------   --------   ----   -----   ----   -----
     1   John      Wilson        2       3     75     225
     1   John      Wilson        2       4     75     300
------   -------   --------   ----   -----   ----   -----
     1                                  7            525

     2   Fred      Jackson       1       4     85     340
     2   Fred      Jackson       2       5     85     425
------   -------   --------   ----   -----   ----   -----
     2                                  9            765

     3   Anne      Rowan         2       5     75     375
     3   Anne      Rowan         1       6     75     450
------   -------   --------   ----   -----   ----   -----
     3                                 11            825

                                                    2115
```

Following is the shell script that produces this default report. Note that it only takes only 11 lines of simple, readable code. Actually, it could be written in two lines, but we put each command on a separate line. This example does not require counting columns or characters. There is no “line-at-a-time” processing, as with the other so-called 4GLs — rather, whole files are processed at once:

```
jointable hours employee |
       sorttable code |
       jointable -j1 code -j2 number - task |
       sorttable number |
       column number fname lname code hours rate total |
       compute 'total = hours * rate' |
       justify > tmp
subtotal -l number hours total < tmp
total total < tmp |
       justify |
       tail -1
```

The data used in the *UNIX World* samples were not included in the articles, primarily because their format is virtually unprintable. **/rdb** data are flat ASCII files. Table **hours** is as follows:

```
number  hours   code
------  -----   ----
     1      3      2
     1      4      2
     2      4      1
     2      5      2
     3      5      2
     3      6      1
```

Table **employee** is as follows:

```
number  fname   lname     rate
------  ------- --------  ----
     1  John    Wilson      75
     2  Fred    Jackson     85
     3  Anne    Rowan       75
```

And table **task** is as follows:

```
number  name
------  ------------------
     1  unix/world
     2  \rdb
```

Here is the exact report format required in the *UNIX* articles of July 1986:

```
  number   Employee Name      code    hours     rate      total
       1   John Wilson          2     3.00    75.00     225.00
       1                        2     4.00    75.00     300.00
                  * Employee Total    7.00              525.00

       2   Fred Jackson         1     4.00    85.00     340.00
       2                        2     5.00    85.00     425.00
                  * Employee Total    9.00              765.00

       3   Anne Rowan           2     5.00    75.00     375.00
       3                        1     6.00    75.00     450.00
                  * Employee Total   11.00              825.00

                                  ** Report Total         2115.00
```

And here is the COHERENT shell and **/rdb** program that produces the exact format:

TUTORIALS

```
jointable hours employee |
sorttable code |
jointable -j1 code -j2 number - task |
sorttable number |
column number hours code fname lname rate name total |
compute 'total = hours * rate; name = sprintf("%s %s",fname,lname)' |
column number name code hours rate total > tmp
compute 'if (name == prev) name = ""; prev = name;\
hours = sprintf("%4.2f",hours); rate = sprintf("%6.2f",rate);\
total = sprintf("%7.2f",total)' < tmp |
subtotal -l number hours total |
compute 'if (code ~ / / && code !~ /-/) code = "* Employee Total";\
if (code ~ / / && code !~ /-/) number = ""' > tmp1
rename name "Employee Name" < tmp1 |
justify -r number hours rate total -l "Employee Name" -c code |
sed "/---/d" | sed "s/^/  /" | sed "s/rate/  rate/" |
sed "s/total  /  total/"
TOTAL=`column total < tmp |\
total | compute 'total = sprintf("%10.2f",total)'| headoff`
echo "                                    \
** Report Total                          $TOTAL"
```

This is still only 22 lines. The powers of the COHERENT shell, **awk**, and **sed** programs are used for string and arithmetic processing. The user only needs to know COHERENT tools, not yet another language. If you know COHERENT tools, you understand this; and if you do not, learning them has much greater value than learning yet another special programming language from a single vendor. As COHERENT and MS-DOS/UNIX systems become the standard, you can use your COHERENT skills on almost all computers.

TUTORIALS

Shell Menus

When setting up software applications, it can be important to provide menus to the user. These tell the users what options are available, and make it easy for them to choose the ones they want. As more functions are added to the system, they can also be added to the menus. In this chapter a simple shell program menu is discussed.

Example Shell Menu Program

A simple way to create menus in COHERENT is with a shell script. The advantage of using shell programming is that it is so easy and you can do anything with it. A sample menu program is included with **/rdb**. You can copy it into your own directory and edit it to be one or more menus for your system. For details see the **menu** manual page.

The **menu** program is in two parts. The first half of the menu simply paints the menu selections on the CRT terminal. It is a simple COHERENT **cat** or **echo** command:

```
cat <<SCREEN
$CLEAR                  COHERENT MENU

Number   Name     For
-------  -------  ------------------------------------------------
     0   exit     leave menu or return to higher menu
     1   Menu     goto another local menu (if any)
     2   sh       get unix shell
     3   vi       edit a file
     4   mail     read mail
     5   send     send mail to someone
     6   cal      see your calendar
     7   who      see who is on the system
     8   ls       list the files in this directory
     9   cat      display a file on the screen
    10   rdb      display rdb commands

Please enter a number or name for the action you wish or DEL to exit:

SCREEN
```

The **cat** command shown uses the *here file* feature of the shell: all of the text from the **<<SCREEN** to the line that consists of only **SCREEN** is sent to the standard input of the **cat** command. **cat** sends its output to your terminal. So the text between the two **SCREEN** lines is displayed on your screen.

This lets you edit any menu you wish. The one shown is only one example. Use your text editor to set up any menu format you desire. Whatever you type will be displayed on the user's terminal screen.

case Actions

Whatever the user types is assigned to the shell variable **ANSWER.** (After the first word of the reply, any more words are assigned to the shell variable **COMMENT**).

```
read ANSWER COMMENT

case $ANSWER in

0|exit)      exit 0 ;;
1|Menu)      Menu ;;
2|sh)  sh ;;
3|vi)
       echo 'Which file or files do you wish to edit'
       read ANSWER COMMENT
       vi $ANSWER $COMMENT
       ;;
4|mail)      mail ;;
5|send)
       echo 'Please enter login name of person to send mail to'
       read ANSWER COMMENT
       echo 'Type you letter, and end by typing Ctrl-d'
       mail $ANSWER
       ;;
6|cal) (cd ; calendar) ;;
7|who) who ;;
8|ls)  ls ;;
9|cat)
       echo 'Please enter the name of the file you wish to see'
       read ANSWER COMMENT
       cat $ANSWER
       ;;
10|rdb)      menu.rdb ;;
*)     echo 'Sorry, but that number or name is not recognized.' ;;
esac
```

After the user's answer is read in, the shell **case** statement is used to match the answer to a number of possible cases. If the user types a number 8 or **ls**, then the **ls** command is executed. This sample gives the user the choice of two ways to indicate a menu selection: numbers or short mnemonic string names.

After the **case** pattern, you can type any shell command or call any shell program, or any combination of both. This gives you complete power to do anything, as a result of a user's choice. In the previous examples, you can see how to simply execute a command, invoke another menu, ask for more information and use it in a command, and so on. The ***)** case is selected if the user does not type any of the patterns previously listed. This gives an error message.

To make this system work, two commands are needed. A **clear** command clears the screen so that the form can be written on it and the **cursor** command moves the cursor to each field of the screen.

termput and tput Commands

There are many CRT terminals with different capabilities and commands. Terminal capabilities are described in the **termcap** file, which contain capability descriptions for a number of commonly used terminals. You only need to find out what it calls your terminal and assign that name to the shell variable **TERM**. (Be sure to also **export TERM**). To pick up the command string for a terminal capability **/rdb** provides the **termput** command. It searches for the terminal name you assigned to **TERM** and then for the capability you indicate.

clear Command

The **clear** command clears the screen. To speed up clearing the screen, assign the clear command sequence to the shell variable **CLEAR**. Then you can use **$CLEAR** in **echo** and **cat** <<*HERE* statements. This is ten to a hundred times faster because it does not require that you look up the sequence in a file first:

TUTORIALS

```
CLEAR=‘termput cl‘  # /rdb command
export CLEAR
echo $CLEAR
```

cursor Command

For the **cursor** command to work correctly, you must set up a shell variable called **CURSOR**:

```
CURSOR=‘termput cm‘
CURSOR=‘tput cup‘
export CURSOR
cursor 20 40
```

The **cursor** command moves the cursor to line (row) 20, character (column) 40 on the screen. It uses the sequence in the **CURSOR** variable.

TUTORIALS

Tables and Forms

Simple forms can be used to for data entry and to display data on CRT screens. The two can be mixed with a form in which some fields are filled in and the data from the data base is written into the remaining form fields. **/rdb** comes with several forms systems: **ve**, described earlier, and a shell level forms system, described here. **ve** was designed to make creating and entering data into **/rdb** data bases easy, using a **vi**-style interface. The **screen** program, on the other hand, can be a more general-purpose application development tool that can be customized for any specific application. It's an example of a program generating a program.

To make the shell-level system work, you need the **clear** and **cursor** commands, described in the previous chapter.

Building a Screen Form

You can write a shell script that paints a screen and accepts input. You can clear the screen and use the **cursor** command to move to any location. There you can use the **echo** or **cat** command to display whatever you wish. You can use the **read** command to read in anything the user types and assign it to a shell variable. You can take the users information and look up data in the data base, using the database query commands, and display it anywhere on the screen. The **update.inv** command found in the $RDB/lib directory is an example of this.

The **screen** program uses a screen definition form, which you create with any text editor (just like the **/rdb report** program form):

```
Makeapile, Inc.            Inventory Update            <!date!>

Item Number: <Item>

Item        Cost       Value      Description
----        ----       -----      -----------
<Item>      <Cost>     <Value>    <Description>
```

Amount Onhand: <Amount>

Then you input that form into the **screen** program:

```
       screen < inv.f > inv.s
```

This yields:

```
: paint crt screen
exec 3>&1 1>&2
cat <<SCREEN
${CLEAR}Makeapile, Inc.            Inventory Update

Item Number:

Item        Cost       Value      Description
----        ----       -----      -----------

Amount Onhand:
SCREEN
```

```
: read user input
cursor  0 56 ; date;
cursor  2 13 ; read Item;
cursor  6  0 ; read Item;
cursor  6 12 ; read Cost;
cursor  6 23 ; read Value;
cursor  6 34 ; read Description;
cursor  8 15 ; read Amount;

: output table head
exec 1>&3
echo "Item   Item   Cost   Value   Description   Amount"
echo "----   ----   ----   -----   -----------   ------"

: append row
echo "$Item   $Item   $Cost   $Value   $Description   $Amount"
```

The output is a shell program that will paint the screen, read the user's input, and output a table of the user's responses. This shell program is simple, but you can edit it to do any advanced operation you like.

TUTORIALS

Fast-Access Methods

Fast-access methods allow you to get rows from a table or list file faster than sequentially reading the file. This is very important for larger data bases. Where sequentially reading a huge file would take minutes, the right fast-access method might take only a few seconds to find the records you want. Traditionally, these methods were very important to data-base management systems. For very large data bases, they will continue to be important. But one or a few users on a microprocessor with small to medium-sized tables will seldom need these speedups.

Appropriate Use

It is important to point out that these methods are not always faster and usually require significant time to set up. They are almost never justified unless the file is large (more than 1,000 rows), and the extra time to index or sort the table is worthwhile.

The command **row** finds rows that match complex logical conditions and regular expression (string patterns). But these fast-access methods require that the key value be expressed more specifically. No *greater than, less than,* or string pattern matches: only *equals,* and *partial initial matches* (for example, the first few letters of a key).

We've implemented five fast-access methods: sequential, record, inverted, binary, and hash. **/rdb** has two commands that use these methods: **index** and **search**.

The index and search Commands

We will use a two-row table to show the **index** and **search** commands. Here is a **mailtable**:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

The hash method is often the fastest, so let's use it as an example.

```
index -mh mailtable Name
```

The **-m** means method, and the **h** specifies hash. The **index** command builds a hash secondary index table. You can see it, if you are curious, by listing out the table name followed by the **.h** secondary index extension; for example, **mailtable.h**:

```
  Offset
--------
       0
     156
       0
     104
       0
```

Each of the names in the **Names** column of the **mailtable** was *hashed* into row 4 (104) or 2 (156). The offset in the hash table (the nonzero numbers) is the location (in bytes) in the main file, of the row with that key value.

Searching

The **search** command allows you to search for one or more rows in the table in four different ways. You can input a key two ways, and you can input a table of keys two ways.

Interactive

Now you can search the mailtable by name in an interactive way:

```
search -mh mailtable Name
```

The **search** command first prints out the table head line

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
```

then waits for you to type in a key value. In this case, if you type

```
Ronald McDonald
```

search prints the following from **maillist**:

```
     1   Ronald McDonald   McDonald's      (111) 222-3333
```

After you type the name and hit the return key, the row will print almost immediately, even if the file is huge. You may continue to type keys. When you are finished, type **<ctrl-D>**.

Pipe Key

The **search** command can also be used in a pipe. You can input a table or list of keys through a pipe and the **search** command will output a table or list of rows which match your input keys. This becomes a fast join, with keys coming in and a table of rows coming out. For example, the command

```
echo "Ronald McDonald"| search -mh mailtable Name
```

yields:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
```

File Input

In addition to sending a single key to **search**, you can also send a whole table of keys. **search** will look up each key and output a whole table of matching rows. It is like a high speed join; keys in, records out.

Here we have our name keys in a table called **name**:

```
Name
---------------
Ronald McDonald
Chiquita Banana
```

The following command directs it to the standard input of the **search** command and gets the matching records out.

```
search -mh mailtable < name
```

This yields:

TUTORIALS

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

Note that you do not have to tell the **search** command the name of the column to search on, if you send it a table with the column name, because **search** can pick up the column name from that input table.

File Input by Pipe

You can also put the **search** program in a pipeline and it will send to its right a table of rows that match the table of keys coming from its left. In other words, keys in, matching records out.

As an example, consider again the table **name**:

```
Name
---------------
Ronald McDonald
Chiquita Banana
```

The command

```
cat name | search -mh mailtable
```

writes the following to the standard output:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

The **cat** command simulates any program or pipeline that produces a table of keys, including other **search** commands.

Multi-Rows, Multi-Columns, and Multi-Keys

The **search** command will produce multiple rows, if more than one row matches the key. You can have multi-column keys consisting of more than one column. You can also send more than one key, multi-keys, to **search** and it will output all matching rows.

Methods of Searching

The five different fast access methods each have their own advantages and disadvantages. It is an art and a science to figure out which to use in a given situation, or whether to use them at all.

Sequential

Sequential is the simplest and slowest method. It is hardly a fast access method at all, but is included for completeness. This method simply looks at every record in the table for a match. With a big file, this will take a long time.

When might you use it? It is better than **grep** because a sequential search will look at only a single column for a match, instead of the whole row. Thus it avoids matching strings in the wrong columns of the row.

Also, use the sequential method to time how fast it takes. You will often be surprised that this method is fast enough for your system. If it is fast enough, use it since it requires no indexing or overhead to use. The table can be in any order and can be updated randomly.

Record

One problem with variable-length records is that there is no simple computation of where a row such as number 7 is. Without this method, we would have to sequentially search the table, counting records, until we came to the one we wanted. On the average we would have to search half the file. When we index with this method, the **index** program runs through the whole file and builds a secondary file which contains the offsets to each row. The secondary index table is named by adding a **.r** to the end of the table name. It contains fixed length rows so that the record number can be computed. At that address is the offset of the corresponding record in the data-base table. For example, consider the following **mailtable**:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

The command

```
index -mr mailtable
```

produces index table **mailtable.r**, which appears as follows:

```
   Offset
---------
      104
      156
```

Note that each of the offsets in the secondary index table **mailtable.r** is the byte address of that row in the main **mailtable** table.

Binary

The binary method requires that the table be sorted on the key columns. With this method, the **index** command simply sorts the table. The **search** command can find the desired row by first looking at a row in the middle of the table. It can use the COHERENT system call **seek()** for fast access to any byte in the file. **search** compares the key value of that center row with the key it is looking for. If the row's key is too high, it jumps to the one-quarter point in the file, if too low, to the three-quarter point in the file. Each probe cuts the file in half so that the record can be found quickly.

If there are a thousand records in the table, only ten probes are needed to find the record you wish. One million records require only 20 probes. A billion records need only 30. This is called *log n* search time, where *n* is the number of records in the file and the log to the base of 2 is the number of probes needed to find a record. The sorting takes *n log n* time with the fastest sort routines.

To prepare **mailtable** for this type of search, use the command:

```
index -mb mailtable Name
```

mailtable now looks like:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     2   Chiquita Banana   United Brands   1234
     1   Ronald McDonald   McDonald's      (111) 222-3333
```

Note that the **Name** column is now sorted so that the binary search will work. This is a good method when you have to keep the file in sorted order anyway. Then your sorting pays off twice. It is a painful method, however, if you are adding and deleting records often and have to resort often.

TUTORIALS

Hash

The hash method takes the key and performs a mathematical operation on it that converts it into a single number. Each ASCII character in the key is added together and modulo-ed with the size of the hash table to produce a number that is its location in the hash table. That number is an index into a secondary hash table. At that location should be the offset to the record in the main table. The row at that offset in the main table is checked to see if its key column(s) match the key we are looking for. If it is, we have found our record.

It might not match because more than one key may hash to the same number by accident. If it does not match, the next offset in the hash table is selected and tested. Each offset is tested until a match is found, or until a value of zero is found. This indicates that there is no matching record in the data-base table. So the search fails. No error message is produced, just no record is output.

As an example, consider again our **mailtable**:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

To prepare **mailtable** for a hash-table search, use the command:

```
index -mh mailtable Name
```

This hashes the contents of the **Name** column, and writes the results into file **mailtable.h**:

```
   Offset
---------
        0
      156
        0
      104
        0
```

For example The 104 in the fourth row of the hash table **mailtable.h** is the byte offset of the first record in the data base **mailtable** table. The value '4' (for fourth row) is the result of adding the ASCII values of **Ronald McDonald** together and modulo with 5, the number of rows in the hash table. The size of the hash table is two times the number of rows in the table to be indexed, plus one. Having twice as many hash rows makes it likely that there will be lots of zeros to stop the search for keys. If the hash table is twice as big as the number of keys to hash, then only two probes are needed on average.

Thus, when the **search** command is searching a table via a hash-key index, it does the following:

- Read the field on which the key has been built.
- Sum the key's ASCII values.
- Modulo the sum by the number keys in the hash table (that is, the number of rows in the data table, times two, plus one. This yields the number of the row to read in the hash table.
- Read the appropriate row in the hash table to find the offset of the row in the data table that we want.
- Use that offset to read the row out of the data table.

Hash is usually the fastest method. But it is best for a static file that you are not updating and do not need to keep sorted.

Inverted or Indexed Sequential

The inverted method projects the key columns and the offset of each row and sorts on the key columns to create a secondary index file. Then the **search** command can use a binary search on the key columns to find the right offset into the data-base table.

Consider once again our **mailtable**:

```
Number   Name              Company         Phone
------   ---------------   -------------   --------------
     1   Ronald McDonald   McDonald's      (111) 222-3333
     2   Chiquita Banana   United Brands   1234
```

To prepare an inverted index for on its column **Name**, type:

```
index -mi mailtable Name
```

This yields table **mailtable.i**, which appears as follows:

```
  Offset   Name
---------   -------
      156   Chiquita Banana
      104   Ronald McDonald
```

Note that the **Name** column has been projected and sorted and that the offset column contains the corresponding offset in the main table.

Partial Inital Match

The **-x** option specifies that partial initial match is to be used. This means you can use only the first few letters of a key, and a match will be made on all those records whose key is matched up to that point. When you type:

```
search -mi -x mailtable Name
```

search prints the head line and dash line, then wait for you type the key for which to search. If you type:

```
Ron
```

then **search** prints:

```
     1   Ronald McDonald   McDonald's      (111) 222-3333
```

As you can see, **Ron** matched the first few characters in "Ronald McDonald".

This method of searching helps save you unnecessary keystrokes; but on large tables it will retrieve more rows than you really want.

B-tree

B-tree subroutines are now a standard on computer systems, for those who want to return to software prisons; but of course the easiest to use and most general tree routines are to be found, as usual, in the power of COHERENT itself. What is the hierarchical directory structure but a collection of shell level programs to manipulate a tree? You can use the directory structure itself to implement the first few levels of (tree structured) indices.

For example, in a company data base, you can have a directory for each city; within each city directory, a directory for each department; within each department directory, a file for each employee, each such file having header records and one employee record. Then, to retrieve all the New York sales employees whose names begin with J, one could say:

```
union nyork/sales/J*
```

TUTORIALS

and, to retrieve all sales employees:

```
union */sales/*
```

COHERENT itself has many programs to manipulate trees. So there was no point in duplicating the power of COHERENT.

The tree method is occasionally appropriate, but usually the most expensive in space requirements. Its advantages are in rapid updates, deletes, and inserts of new records in very large data bases. If you have a very large data base that requires lots of updates, it's really quite fast to build the first few levels of indexing into the COHERENT directory structure, but it does involve extra file system overhead.

Analysis

It takes analysis and testing to determine which is the best method for a given situation. Each method has its advantages and disadvantages. Theory only takes one so far. You can time the different methods on your computer to determine the fastest.

To determine the best method to use in a situation, you must both analyze and test your different options. The advantages and disadvantages of each method is discussed earlier. You should also test any strategy you adopt. You might find that a simpler method is fast enough, or the overhead of a fast method outweighs the benefits of speed.

Management

Fast-access methods need to be managed. If the files are being updated, they may have to be reindexed. Reindexing takes time. You may want to schedule it when users are not accessing the data and perhaps when the computer is not being used much. A nightly reindexing may be appropriate.

The COHERENT command **at** lets you schedule big jobs like this, at say 3 A.M., when the system is more likely to be quiet. When you choose an access method, you must plan for any management that is needed. This can also be accomplished by writing a command into the system file **/etc/crontab**.

TUTORIALS

Miscellaneous Commands

/rdb has more than 100 programs to help you develop software applications. Here is a brief discussion of some of them, grouped together by subject.

Record Locking: One at a Time

When your application requires that several people update the same file at the same time, you must lock records. When one user gets a record to change, it is important that no other user pull out the same record and edit it also. If you allow this, the second record written back will clobber the first record, thus destroying the results of the first user's changes.

Many complain that the COHERENT system does not have a record-locking system. But it is trivial to create record locking with COHERENT tools. **/rdb** provides **lock** and **unlock** commands that you can use or modify for this purpose.

Finding What to Lock

To lock a record, **/rdb** must first find its location within the table where it "lives".

To find the location of a record, use the **seek** command. It uses the fast-access method of your choice to find the record and returns the starting and ending byte location, which can be assigned to a shell variable.

For example, consider the command:

```
LOCATION=‘echo 5 | seek -mb -o tmp inventory Item‘
```

A lot is going on here so let's take it step by step.

The **echo** command sent the number '5' to the **seek** command. **seek** looks for an **Item** numbered 5 in the **inventory** file using the binary fast access method (**-mb**), and writes the record into the **tmp** file (**-o tmp**). **seek** sends a string of four numbers to standard out which is assigned to the shell variable **LOCATION**. Echoing **LOCATION** gives us a resulting of the form:

```
207 245 0 9
```

The four numbers are the offsets of the first byte and last byte of the record in the data file, and of the first byte and last byte of the offset in the secondary index file. (In this case it is meaningless, because the binary method does not use a secondary file.)

lock and unlock

Now that we know where a record is, we can use the command **lock** to lock it while we work with it, and the command **unlock** to unlock it once we're done with it.

The **lock** command writes the string from the **LOCATION** shell variable into a file in the **/tmp** directory. It checks to see if that area of the file has been previously locked. The **unlock** command removes the location line from the lock file.

Blanking a Record

For added protection, or as an alternative, you can also blank out the record in the file. Use **seek** to get the location, use **blank** to blank the record, and use **replace** to write the blank record into the file.

For example, the following command blanks out line 5 in table **inventory**:

```
LOCATION=‘echo 5 | seek -mb -o tmp inventory Item‘
blank < tmp | replace -mb inventory $LOCATION
```

When we look at **inventory**, it now appears as follows:

```
Item   Amount   Cost   Value   Description
----   ------   ----   -----   -------------
   1        3     50     150   rubber gloves
   2      100      5     500   test tubes
   3        5     80     400   clamps
   4       23     19     437   plates

   6       89    147   13083   bunsen burners
   7        5    175     875   scales
```

Line 5 has been replaced by a row of blanks. Since the data are no longer there, no other user can grab them for manipulation. Users must be instructed, however, that a blank row means that the record has been locked. If they retrieve a blank record, they must *not* replace it with anything.

Dates: Conversion and Arithmetic

One of the basic tasks a data-base system must perform is manipulating dates. Each date must be translated into a standard format, and users must be given a way to perform arithmetic on dates. **/rdb** includes several programs help you handle dates.

Julian and Gregorian

Gregorian is the kind of dates we are all familiar with: January 1, 1989, 1/1/89, 890101, and so on. Unfortunately, there are two problems with this kind of date: Gregorian dates can not be added or subtracted, and they have several different formats. Fortunately, there is another form of date, called Julian, that solves these problems. A Julian date is a number of days since January 1, 4713 B.C., the date that marked the beginning of the Julian calendar. Because a Julian date is an integer, you can add days to and subtract days from a Julian date and get the correct new date.

/rdb has two commands that are used for this purpose. **julian** converts a date from the standard Gregorian to Julian. It converts a whole column of dates so that they can be operated upon. After the Julian dates have been manipulated, the column can be converted back to Gregorian dates with the **gregorian** command.

For example, suppose you are planning a project, and have written a schedule into table **project**, which appears as follows:

```
Date     Description
------   -----------
890601   Start project
890701   Project crises
890801   Abandon project
890901   Coverup blame
891001   Write off losses
```

Now you find that you need to put off the start date of the project by 45 days. To update your table to reflect this change, do the following:

- First, use the command **julian** to convert the **Date** column to Julian dates:

  ```
  julian Date < project > tmp
  ```

  The output has been written into table **tmp**, which appears as follows:

TUTORIALS

```
Date      Description
-------   -----------
1752623   Start project
1752653   Project crises
1752684   Abandon project
1752715   Coverup blame
1752745   Write off losses
```

Those huge numbers under **Date** are Julian days.

- We can now add 45 days to the column with the **compute** command:

```
compute 'Date += 45' < tmp > tmp1
```

The result is written into table **tmp1**, which appears as follows:

```
Date      Description
-------   -----------
1752668   Start project
1752698   Project crises
1752729   Abandon project
1752760   Coverup blame
1752790   Write off losses
```

Note that all of the Julian dates are 45 days later.

- Now we can convert back to Gregorian:

```
gregorian Date < tmp1 > project
```

The result is written back into our original table, **project**, which now appears as follows:

```
Date     Description
------   -----------
890716   Start project
890815   Project crises
890915   Abandon project
891016   Coverup blame
891115   Write off losses
```

This can also be done in one pipeline:

```
julian Date < project |
> compute 'Date = Date + 45' |
> gregorian Date > tmp
mv tmp project
```

Difference

You can also find the difference between two dates.

For example, a building contractor may have a table called **house**, which contains the start dates and stop dates for each phase of building a home:

```
Start   Stop     Days   Step
-----   ------   ----   ----
890101  890118          Lay concrete
890121  890212          Setup frame
890215  890302          Build walls
890305  890401          Install fixtures
890408  890425          Sell at a loss
```

To compute the number of days for each step, we can use the **julian** and **gregorian** programs on the

house table:

```
julian Start Stop < house |
compute 'Days = Stop - Start' |
gregorian Start Stop > tmp
mv tmp house
```

house now appears as follows:

```
Start   Stop     Days   Step
-----   ------   ----   ----
890101  890118   17     Lay concrete
890121  890212   22     Setup frame
890215  890302   15     Build walls
890305  890401   27     Install fixtures
890408  890425   17     Sell at a loss
```

Formats

There are a number of different formats that people use to express dates. The **julian** and **gregorian** programs also allow you to convert between three different formats.

Computer
: Computer date format is *YYMMDD*. For example, January 31, 1989, translates to "890131". This format is best because it can be sorted, and is the most compact. You can also test to see if one date is greater than another. Most users can quickly get used to it.

US
: In the United States, dates are written in the sequence *MMDDYY*. For example, January 31, 1989 translates to "1/31/89".

European
: The Europeans use the sequence *DDMMYY*. For example, January 31, 1989, translates to *31/01/89*.

Conversions

The **/rdb** commands **julian** and **gregorian** let you convert dates from one format to another. Simply specify the current format to the **julian** command and another format that you want to the **gregorian** command, and the format will change. The default format is **computer** format.

For example, let's change the computer format in the **project** table to US format:

```
julian Date < project | gregorian -u Date
```

project now appears as follows:

```
Date    Description
--------        -----------
 6/01/89        Start project
 7/01/89        Project crises
 8/01/89        Abandon project
 9/01/89        Coverup blame
10/01/89        Write off losses
```

You can see one of the problems of the standard date is that it is eight characters. Since the tabs are also eight characters, the next column has to move over. You can use the **justify** command to pretty things up:

```
julian Date < project | gregorian -u Date | justify Date
```

project now appears as follows:

TUTORIALS

```
Date       Description
--------   ----------------
 6/01/89   Start project
 7/01/89   Project Crises
 8/01/89   Abandon project
 9/01/89   Coverup blame
10/01/89   Write off losses
```

Set-Theory Commands

In addition to the basic relational commands, there are several that come from relational set theory.

Concatenating Tables

The **union** command appends one table to another, making a larger table consisting of all of the rows of each.

For example, consider a set of accounting tables. The first table is called **journalcash**:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ----------------------------------
891222       101   25000            cash from loan
891223       101              5000   cash payment
891224       101             15000   cash payment to CCPSC for parts
```

The second is called **journalloan**:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ----------------------------------
891222   211.1              25000   loan number #378-14 Bank Amerigold
891223   211.2               5000   note payable to Zarkoff Equipment
```

And the third is called **journaladjust**:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ----------------------------------
891224       130   30000            inventory - parts from CCPSC
```

Note that all three tables consist of the same set of columns.

Now, we use the **union** command to consolidate these subsidiary journals to create a general journal:

```
union journalcash journalloan journaladjust > journal
```

Table **journal** appears as follows:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ----------------------------------
891222     101     25000            cash from loan
891223     101               5000   cash payment
891224     101              15000   cash payment to CCPSC for parts
891222     211.1            25000   loan number #378-14 Bank Amerigold
891223     211.2             5000   note payable to Zarkoff Equipment
891224     130     30000            inventory - parts from CCPSC
```

Subtract One Table From Another

The command **difference** lets us subtract one table from another to give the rows that are in the first table but not in the second table.

For example, the following command finds the difference between two of the above-described tables:

```
difference journal journalloan
```

This prints the following on the standard output:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ---------------------------------
891222       101   25000            cash from loan
891223       101              5000   cash payment
891224       101             15000   cash payment to CCPSC for parts
891224       130   30000            inventory - parts from CCPSC
```

Intersect Between Tables

The command **intersect** lets us find the rows that are in both tables, the intersection of the set.

For example, the following command finds the intersection between two of the above tables:

```
intersect journal journaladjust
```

It prints the following on the standard output:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   ---------------------------------
891224       130   30000            inventory - parts from CCPSC
```

TUTORIALS

Combining /rdb with COHERENT

Much of your data-base work can be done with COHERENT commands. Many of the traditional problems of data bases disappear when you have a COHERENT system. If you have only one or a few users, problems like security, distributed data, concurrent access to files, backup, checkpoint and recovery, validation, audit trails and logging, and others, are minimized.

This chapter gives suggestions for combining **/rdb** with COHERENT tools to develop solutions to these problems. This is not to say that there are no longer any problems. There is still a role for programmers and system/analysts. But it is important for these professional people, who got their experience on older systems, to re-think what they know in light of the new tools and the new economics of our changing technology.

Multi-User Concurrent Access to Files

On small systems, you can assign each file to only one person. Then only one person need access it at a time. Any number can append new records to a file at one time with the **enter** command. Also, any number of users can read a table into their **/rdb** and COHERENT commands, since reading does not change the table.

The only problem that arises is when two or more people must edit a table or list file at the same time. Files can be locked in several ways. There is a **lock** and an **unlock** command for this purpose. With the **blank** and **replace** commands, you can blank out a record while you have it out for edit. The **update** commands does this.

Each user can move (**mv**) the file to a new name before editing it. Perhaps they might move it into their own directories, make their changes and move it back. That way other users will not find the file when they try to edit it. This is how the **vilock** command works. Application programs can change the permission of a file so that it is *read only* to others on the system, when our user is updating it.

There are also the **lock** and **unlock** commands, which are shell scripts to show how easy it is to solve this problem with COHERENT tools.

Screen Form Entry

If you want to put forms on the terminal screen for the users to input data into, you can use **ve** or the **screen** program. **screen** takes a form you create in a text editor, as with the **report** command, and creates a shell script that will display the form, read the user's input and output a table. Because it is a shell script, it can be edited to do a lot more including validation, table lookup, and others. **ve** also takes a form, like the report command's form, and creates a table.

Security

You can handle security by physically protecting the computer and the terminal, by using the COHERENT security permissions on files and directories, and by encrypting files. **ve** contains special security features like unwriteable and invisible fields. There are read, write, and execute permissions for the owner of the file, the group, and anyone on the computer. See the COHERENT commands **chmod** and **crypt.**

Backup

You should backup the **/rdb** tables and list files just like regular COHERENT files. You can simply copy a file or a whole directory to a backup media like magnetic tape, cassette tape, or floppy disk, or transmit to another computer for backup. The COHERENT **ustar** and **cpio** commands can be

used for putting an entire directory structure into a single file.

Checkpoint and Recovery

You can always create a copy of a file that you are working on. **/rdb** commands do not alter their input files. They simply create new files or pass their output (pipe) to another program. In a crash, the original file is not harmed. Only the editors modify the existing file and at least one, **vi**, has a crash recovery system.

Validation

ve provides programmable validity checks for data as it is entered. **awk** scripts and other programs can be developed to check the validity of operator input. The **vi** editor has the ability to assign to a single key a complex program. It can be used to validate each line typed. The **/rdb** command **validate** is available to do any data validation from simple to complex. See the sections on validation in the **/rdb** manual and the **ve** chapter.

Audit Trails and Logging

This is very easy because of the **diff** command.

At the end of each day's work the **diff** program can be run like this:

```
diff todaysfile yesterdaysfile > todayslog
```

The file **todayslog** will contain all lines that have changed since yesterday. Save it and you have a log or audit trail. It can also be used for recovery to any date in the past.

TUTORIALS

Other Data-Base Systems

Many data-base management systems are available. Each has its strengths and weaknesses. However, they usually take a traditional approach. Such systems were developed on operating systems that provide only a fraction of the services that COHERENT provides. Therefore, the data-base developers had to write as much functionality into their data-base packages as they could. When these data-base management systems are moved to the COHERENT environment, they require that you leave COHERENT tools and all of their power behind, and go into a software box where you are limited to the functions that the data base developers have supplied.

There are many other data-base packages running on COHERENT systems, but they all take this traditional *big box* approach. One usually finds that functions COHERENT tools could provide are not available inside those data-base system boxes. There is usually no good way to get the data out to COHERENT and back. It was the frustration of knowing that we could do what we needed to do with COHERENT tools, but that our data was locked up in the data base package, that originally prompted the development of **/rdb**.

With **/rdb** you always have access to the full power of the COHERENT system, because **/rdb** commands are COHERENT commands that can be piped together with other COHERENT commands. They extend the power of COHERENT tools rather than waste it. **/rdb** works at the shell level and allows you to use all of the COHERENT utilities and powers. With shell programming you can build applications easily with both COHERENT and **/rdb** commands mixed and piped together.

With other data-base systems, you must learn a whole new language/syntax that is unique to that data-base package. Learning **/rdb** is learning the COHERENT system, which is useful in many applications. Or you can use the COHERENT knowledge you have already acquired. In fact, it is an excellent way to learn COHERENT tools, just as many people learned computers starting with a spreadsheet program.

Resource Use

When conventional data base systems duplicate COHERENT functions, programs become huge. These programs take up tremendous amounts of computer memory and other resources. Most data base packages will severely impact even large expensive computers with only a few users. By doing things the COHERENT way, only the small programs that are needed are brought in from the disk, putting only a light load on the system. This can save hundreds of thousands of dollars in computer hardware, to say nothing of software.

C Programming Unnecessary

The COHERENT system has many programming languages, including the C language. Although many programmers naturally think they must use such programs in their work, it is usually unnecessary. Putting together COHERENT programs will usually accomplish whatever is needed.

Although much can be done with the basic COHERENT system as delivered, some data-base facilities and tools are missing or are hard to create. Therefore, a number of relational data-base management systems have been developed for the COHERENT system. Unfortunately, most come from other operating systems and ignore the power of COHERENT. On other operating systems, virtually nothing exists to help the developer and user. Everything must be programmed from scratch. If the developer or the user is to have a feature, it must be programmed into the data base program.

On COHERENT the reverse is true. Much of what you need is already available in the COHERENT environment. The traditional data base management systems, when moved to COHERENT, discourage the developer and user from getting access to these powerful and familiar COHERENT tools. It is as if these systems create an unfamiliar box that you must go into, leaving the power of COHERENT behind.

/rdb, however, was built for the COHERENT environment. It consists of over 100 COHERENT-like commands that fit together nicely with other COHERENT commands. It has the tools that were found to be needed from real experience of developing software applications. Using COHERENT shell programming, developers can quickly put together applications from both COHERENT and **/rdb** commands. Users face a unified environment. They may not know or care whether the commands they use come from the COHERENT system or **/rdb**. This is the correct way to extend the COHERENT system: keeping the power and genius of the COHERENT system and adding functions to it.

Speed

Most of the research on data bases and their problems assumes that you have huge files, need to find a record (row) very quickly, and have many users. Not many years ago, few could afford to have the power that a microcomputer offers today. Then computers cost millions of dollars and had to be shared by many users. Only large volume applications were cost effective enough to computerize, leading to a tremendous concern for speed and size. Early eight-bit microcomputer systems reinforced this thinking. With weak computers, small memories, and floppy disks, size and speed were still overriding issues.

Today, and more so in the future, we have powerful 16- and 32-bit microcomputers with large memories and big disks, but so inexpensive that they are ubiquitous. Applications will run faster and handle larger data bases. Now we want ease of use and powerful systems. The bulk of data base applications are for relatively small data files of a few hundred to several tens of thousands of records, with a small number of frequently asked queries. If you think of something new to do that was not planned for in the development of the application, you will usually find it very difficult or impossible to do. These traditional data bases often require complicated setups, lots of training to use and are difficult to modify. Worse still, they are huge programs that take up much of the memory and CPU cycles of the computer. Only a few users can slow a big expensive computer down to an unacceptable response. These traditional data base systems have features built into a large program. The COHERENT system has lots of features, but they are in the hundreds of programs stored on the hard disk. COHERENT only brings in the code it needs for a particular function. The average COHERENT utility is about 40 kilobytes. Traditional data base programs are often a quarter of a megabyte and larger.

/rdb takes a much different approach. It is much closer to the ideas of a text editor, spreadsheet and, of course, the COHERENT system. It handles small data bases as well as large. **/rdb** is aimed at ease of use and maximum power. Even so, it is usually very fast. But when more speed is needed, **/rdb** also has not one but five fast-access methods to give a variety of ways to access large files. **/rdb** is as fast or faster than most other data base management system. Furthermore, even **grep** with its sequential search is not so bad. **grep** is a COHERENT program which looks for a string pattern and prints out each line in which it finds a match. Remember that there is computing overhead for the traditional fast access methods. Binary searches require sorting, hash tables must be computed, B-trees must split their pages when they fill, linked lists must maintain their pointers, and so on. As the size of our tables diminishes, so do the advantages of the fast access methods. **bm**, an implementation of the Boyer-Moore linear search algorithm that is quite a bit faster than **grep** but limited to fixed strings (not patterns), is distributed with **/rdb.** in the **$RDB/lib** directory.

Size

TUTORIALS

Many data-base comparisons put great emphasis on the size of the data base that can be handled. This is really a problem for data-base systems that are one large program. **/rdb,** however, is a group of small programs which can be piped together. Its only limits are those of your hardware and COHERENT itself.

A table can be as big as the available free space on your disk, or as big a file as your COHERENT system can handle. Use the COHERENT commands **df** to see how much space is left on the disk, and use the COHERENT command **du** to see how much disk space a given directory uses. For details, see these commands' entries in the Lexicon.

TUTORIALS

Can You Say that in English?

If you are not a mathematician, you may find the literature on relational data bases to be incomprehensible. One problem is the terms used in the literature. Different terms are used by the mathematicians, computer people, and other humans.

We try to use human terms throughout the **/rdb** documentation. We lapse into computer terms either by mistake, or when we think we are talking to programmers and analysts about the more technical details. We ignore the mathematicians, since they seem to ignore us. But we love them. Without them we would not have relational theory. Here is a table that will help each group translate terms used by another group:

Humans	*Hackers*	*Mathematicians*
table	file	relation
column	field	attribute(domain)
row	record	tuple
number of columns	number of fields	degree
number of rows	number of records	cardinality
list of tables	schema	datamodel
user's tables	userview	datasubmodel
simplified	simplified	normalized
no repeated columns	no multi fields	normalized
one concept per table		3rdnormalized
get rows	getrecords	select
get columns	getfields	project
combine tables	concatenate	join(union)
new command	shellscript	view

Data-Base Models

A data-base model is a way of structuring and thinking about data. We use the relational model, but there are several others. There are three major types of data base models that have been implemented in several commercially successful data base management systems: relational, network, and hierarchical. In addition there are at least four more important models discussed in the literature, but for which few implementations exist as yet: entity-relationship, binary, semantic network, and infological [Tsichritzis, 1982]. We will probably see more of them in the future.

The major difference between these models is their structure. As you read through these descriptions, you will probably have a sense of *deja vu*. Each is quite similar to the other with some differences. Remember that information in each model can be converted or transformed into the other models.

Hierarchical

The hierarchical data base has a tree structure. (In computer science, trees have their roots at the top and grow their branches and leaves downward.)

This type of data base was popularized by IBM. Their IMS data-base management system has been widely used in the past. When the users build a data base, this structure must be imposed upon their data. Some searches are speeded up (such as from classes to grades). Others become difficult or impossible, e.g., *What grades did an instructor give?* The emphasis is on speedy responses from older, slower, and usually heavily burdened computers. The speed is achieved by anticipating standard searches and optimizing them. But *ad hoc* queries (ones you just think up as you are working) are not only slow, but are often difficult, even impossible, to do.

Network

The network type of data base has a two-level structure. Both **Total** and **Image/Query** are examples of this approach. This model comes from a CODASYL committee standard. This system is a little easier to build and use than hierarchical, but still requires that the user know the structure and *navigate* through the links. It is difficult, if not impossible to get from one detail file to another.

Relational

Relational data bases have what are called a *flat* file structure. All files are at the same level. With such a structure and a set of commands like **project**, **select**, and **join**, one can get all of the information out of the data base that has been put into it.

E. F. Codd of the IBM San Jose Research Laboratory proposed this relational model in the early 1970s [Codd 1970]. Since then, the relational approach has captured the attention and approval of most of the academic, and now business, researchers in the field. IBM is converting to relational with its new System R data base management system. It uses SQL for a user interface. Almost all of the data-base systems that have been built for COHERENT are relational.

A relational data base management system is considered by most researchers in the field as the best for several reasons. Relational data bases have a solid mathematical base in relational set theory, relational algebra and relational calculus. There are theorems in this relational mathematics that prove that any data put into a relational data base can be extracted. The mathematical base also assures us that the manipulations we perform will have correct results, just as arithmetic assures us that the mathematics functions we perform on the computer have correct results.

The relational structure is the simplest. All information is kept in simple tables. The user does not have to navigate through complex networks (network model) or tree structures (hierarchical model). Relational theory has tables (relations) with rows (tuples) and columns (attributes of domains), which anyone can understand.

It also has functions on tables: **project** (**column**), **select** (**row**), **join** (**jointable**), and so on. These functions both input and output tables, just as the functions in algebra which we are familiar with input and output numbers (scalars), lists of numbers (vectors), or arrays of numbers (matrixes). These commands we've discussed earlier. They are the most frequently used.

There are also several commands from set theory that are not as often used but are important for completeness. There has been a lot written on relational data base systems and theory. See the Bibliography.

Entity-Relationship

The entity-relationship model was put forth by Chen [Chen 1976]. This model sees the universe, or more practically the company or institution, as composed of entities and relations between them. Things, people, departments, and so on, are entities. Entities have relationships between them. A department is part of an enterprise. People work in departments as staff members. Equipment is assigned to departments. Some people are heads of departments. One can draw a diagram of these entities and relationships.

There is a one-to-many relationship. A department can have many staff people and many pieces of equipment. But each employee and piece of equipment is assigned to only one department. If the rules of the organization change, so will this graph.

This model has similarities to the three classic models. The relational model holds entities in tables (also called relations). It provides the **jointable** command to combine tables on keys in which there is a relationship between the entites of one table and another. It can also be seen as hierarchical and network because of the structures. This model trys to capture the overall structure of the enterprise regardless of how it is implemented. It is the *big picture.*

TUTORIALS

Binary

The binary model sees data as a graph in which each node is a simple column or field of data and the arcs that join the nodes represent simple relationships between them. It is based on graph theory. As we move to computer systems with powerful graphic screens, this model might become more common.

Semantic Network

The semantic-network model comes from the field of artificial intelligence (AI). (See Quillian [1968] and Sowa [1983]). AI is the part of computer science that trys to get computers to be *intelligent*. Its subfields are games, expert systems, vision, robotics, natural-language recognition, and others. Researchers came up with this model in trying to structure data for these efforts, trying to understand the human associative memory, and trying to understand natural language.

There are many variations of the semantic network model because many researchers have written about different versions in the technical literature. In the broadest sense any graph model is a semantic network model, including the entity-relationship and the binary model.

The new graphic terminals will make this graph model easier to display and input. Since all information can be represented in this way, it will be a very powerful way to interact with computers.

Infological

An infological model is the user's view of the application. There is a dream that some day a user can simply communicate the structure of the application to the computer and the data base system will be set up automatically — somewhere between a general data-base management system and a specific application. Some elements exist today. The new graphic terminal systems provide powerful tools to communicate with the user and give the user the ability to specify needed applications pictorially. Work in this field will be fun and productive for future users.

PROLOG: Programming in Logic

Finally, the PROLOG language adds logic programming on top of a data base model. PROLOG is discussed more in the next chapter.

The Grand Unified Field Theory of Information

The *Grand Unified Field Theory of Information* is that all of the different structures of information are simply transformations of each other. Natural language sentences and their parses, predicate calculus, tables and relational algebra, graphs, and semantic networks are simply different representations and transformations of information.

Each has its advantages and uses. People speak language and understand graphs. Computers can parse language, store tables, and manipulate them through relational algebra and predicate logic. Sentences can be converted to predicate calculus formulas and solved by comparing with the table data base.

This offers enormous power and possibilities. In the years ahead the written literature of many fields can be scanned by computer, parsed, and inserted into tables. Then written or spoken queries can be parsed and converted into predicate calculus formulas which can be evaluated against the relational table data base. The computer can then speak, write, or draw a graph or a picture of the answer.

TUTORIALS

PROLOG and AI

Artificial-intelligence programs, like most programs, rely heavly on data bases. **/rdb** has several facilities to assist AI, and especially expert system programs.

PROLOG Language and Environment

PROLOG is a programming language that allows you to program in logic. It was developed in Europe, and has been adopted as the language for the Japanese Fifth Generation Computer project. It is used in artificial intelligence work. It provides a powerful new method to make logical inferences from data bases. Instead of being able to find data only, it can reason from that data to answer questions that are not in the data base, but are logically derivable from the data. This is an enormous leap in getting the computer to be smart.

PROLOG is an entire programming language and environment, mainly because it was developed on non-UNIX operating systems where almost nothing was provided. But we are only interested in the logical searching features which are its major contribution. The logic programming of PROLOG would be best placed in a UNIX/COHERENT environment with its many tools, rather than in the primitive PROLOG environment.

If you are interested in working with PROLOG, COHware volume 2 contains, among many other items, the source code for a PROLOG interpreter.

Predicate Calculus

The branch of logic that PROLOG uses is predicate calculus. This is not the calculus you learned in math classes, but refers to the manipulation of formulas, of which the calculus you are familiar with is only one form.

Almost all sentences in a natural language can be converted into predicate calculus statements and manipulated logically. The sentence: **Bill loves Kathy** is written: **loves(Bill, Kathy).** This says that there is a relationship, **love**, and that the first argument, **Bill**, has that relationship with the second argument, **Kathy**. This predicate calculus formula can also be expressed as a table:

```
Subject   Object
-------   ------
Bill      Kathy
```

Facts

PROLOG stores facts in the predicate calculus notation. Since almost any sentence can be expressed, almost any information can be stored in the data base.

PROLOG	*Means*
female(michele)	michele is a female
female(jane)	jane is a female
male(john)	john is a male
male(shawn)	shawn is a male
parent(shawn, jane)	shawn is a parent of jane
parent(michele, jane)	michele is a parent of jane

These facts are entered into PROLOG by simply typing them in or having PROLOG get them from a file with the PROLOG **consult** command.

Questions

Once the above data are entered into the PROLOG data base, you can ask questions. The following gives a sample PROLOG "question time" — the user's questions appear in Roman, and PROLOG's replies in *italics.*

parent(shawn,jane)
yes
female(shawn)
no
mother(michele,jane)
no

The first query is found in the data base and *yes* is returned. The second and third queries are not found in the data base, so *no* is returned. None of this is very exciting because any data-base management system can do this. However, we humans know that the third query is true because we know that if **michele** is the parent of **jane** and that **michele** is **female**, then **michele** must be the mother of **jane**. But no data base can know this rule and logically deduce this conclusion except PROLOG.

Rules

This great innovation of PROLOG comes when we add rules and logical inference to the data base of facts. A rule is stated in this form:

```
mother(X,Y) :- parent(X,Y), female(X).
```

In English this rule says that the relationship **mother** exists between two entities *X* and *Y* if (**:-**) *X* is the **parent** of *Y* and (**,**) *X* is a **female**. This rule is stored by PROLOG. Now we can ask again:

mother(michele,jane)
yes

This time PROLOG gives the right answer. Not because it found the fact in the data base, but because it inferred the fact from facts and rules in the data base. PROLOG failed to find the fact in the data base, so it searched its list of rules for one that started with **mother.** It then assigned **michele** to *X* and **jane** to *Y*. It went to the facts after the if (**:-**) symbol. PROLOG first looked for the fact **parent(michele,jane)**. It found that successfully, so it then searched for the fact **female(michele)**. This was also successful, so it responded with *yes*.

With a simple mechanism, we now have the ability to logically infer facts from other facts and rules!

/rdb Interface to PROLOG

/rdb provides an interface to PROLOG with two commands: **tabletofact** and **tabletorule**. These convert **/rdb** tables into the predicate calculus formulas that PROLOG needs to see. Therefore, we can logically query our data base.

tabletofact

The **tabletofact** command converts a table to predicate calculus. First let's look at our **/rdb** fact tables. First, table **female**:

```
Female
------
michele
beth
sandy
jan
```

Table **male**:

TUTORIALS

```
Male
----
kirk
rod
shawn
durk
```

Table **parent**:

```
Parent   Child
------   -------
michele  rod
kirk     rod
```

And finally, table **isa**:

```
Name    Isa
----    ---
rod     human
human   mammal
mammal  animal
animal  lifeform
```

Now let's convert them all to PROLOG fact format.

```
tabletofact female male parent isa > fact
```

File **fact** appears as follows:

```
female(michele).
female(beth).
female(sandy).
female(jan).
male(kirk).
male(rod).
male(shawn).
male(durk).
parent(michele,rod).
parent(kirk,rod).
isa(rod,human).
isa(human,mammal).
isa(mammal,animal).
isa(animal,lifeform).
```

tabletorule

Rules can also be stored in **/rdb** tables and converted to PROLOG format with the **tabletorule** command. For example, consider table **ruletable**, as follows:

```
True            If
--------------  ----------------------
mother(X,Y)     female(X) , parent(X,Y)
father(X,Y)     male(X) , parent(X,Y)
son(X,Y)        male(X) , parent(Y,X)
isa(X,Y)        isa(X,Z) , isa(Z,Y)
```

The command

```
tabletorule ruletable > rule
```

writes the following into file **rule**:

```
mother(X,Y) :- female(X) , parent(X,Y).
father(X,Y) :- male(X) , parent(X,Y).
son(X,Y) :- male(X) , parent(Y,X).
isa(X,Y) :- isa(X,Z) , isa(Z,Y).
```

Then you can consult these files within PROLOG with the **consult** command.

Now you are ready to ask questions. With this data base you can now ask questions about motherhood and fatherhood, and questions like *is rod a lifeform?*

Problems of PROLOG

Richard Forsyth [Forsyth 1984, page 16] lists many complaints about PROLOG. These include the following:

''PROLOG provides a relational data base for free — a big bonus, but the trouble is that it resides in main memory, and is consequently very greedy on storage.''

He also quotes Feigenbaum and McCorduck [1983]: *''The last thing a knowledge engineer wants is to abdicate control to an 'automatic' theorem-proving process that conducts massive searches without step-by-step control exerted by knowledge in the knowledge base.''*

The reason for all of the problems in PROLOG is that it is a great idea *and* a programming environment. On non-COHERENT systems, when you want to do something, you wind up having to do everything else also. You have to write editors, floating-point handlers (some PROLOGs don't), trace, input/output, and on, and on. In addition, both PROLOG and LISP do everything in memory. This is fine for small prototype and demonstration programs in a university AI research lab, but unworkable for large data bases.

The same is true for LISP. Peter Jackson [Jackson 1986] writes about a classic expert system: *''The difference between MYCIN's score and those of Stanford experts was not significant, but its score is as good as the experts and better than the non-expert physicians. However, MYCIN is not currently used in wards for a number of reasons ... it is written in INTERLISP, is slow and heavy on memory ...''*

This problem is not new to the computer world. It was solved decades ago in commercial computing. There needs to be a shift from memory to secondary storage as AI moves from the research labs to real users. See *Expert Database Systems: Proceedings From the First International Workshop* [Kerschberg 1986].

We also need to shift from PROLOG and LISP to UNIX/COHERENT and a powerful data-base management system like **/rdb**. As radical as this sounds, one of the leading expert system companies converted its system from LISP to COHERENT and C. They got a fifty-fold increase in speed! The other expert-system companies are in various stages of converting to UNIX.

On UNIX/COHERENT, you can simply add a new capability to an excellent environment. PROLOG's environment is quite primitive and its inference system narrow and limited. The PROLOG rule inference and search mechanism should be added to COHERENT as simply more programs. We expect to see all of the AI ideas added to COHERENT in the years ahead. We should see different logics, that is, fuzzy logic, Bayesian logic, multi-valued logic and certainty factors. We should also see many new search strategies. Heuristic searches guided by data in the data base are needed. Then we will see many new expert systems developed with these vastly enhanced tools.

By concentrating on only what is new, developers can do a much better job. They won't have to waste most of their time reinventing environment wheels.

searchtree: Data-Base Tree Searching

TUTORIALS

Most AI programs search data bases that can be visualized as trees or networks. Speeding those searches and controlling them is a very important area of research. Usually, these trees are kept in memory and searched with pointers. This limits the size of the data base that a LISP or PROLOG program can handle. It is not as fast as one might imagine. Memory searches are fast, but the overhead of bringing the data in off the disk, and hogging so much memory that programs must swap frequently, quickly wipes out any theoretical speed advantage.

searchtree is a **/rdb** shell program that shows another way to search a tree. It can use the fast access methods to do a breadth first search of any sized data table. It is a shell program example which can be edited to handle many search situations. Heuristics can be built into the code to speed, or otherwise improve, the performance of the search.

TUTORIALS

/rdb and C

You can call COHERENT and the **/rdb** data-base commands from C programs. This chapter gives you several example programs starting from the most simple to the most complex.

Don't Do It

Before we start, it is strongly recommended that you think twice before writing C or other language programs. The shell, COHERENT tools, and **/rdb** commands are so powerful, fast, and easy to develop, you seldom need to bother with the old third-generation programming languages. If you think of writing a program in C by habit, try to break the habit. Always try to do things in the COHERENT shell and **/rdb** first. Only resort to C when there is a compelling reason, as opposed to a compulsion. We consider resorting to C as a failure of imagination, or lack of knowledge or insight, in most cases. If you think you need a C program, you may really need to know more shell programming tricks. Think of your problems as a more general problem, and check to see if there is a COHERENT or **/rdb** program that will do the job.

Speed is the most common excuse for descending to C coding. Remember that computers are getting faster and cheaper. The RAM disk can greatly speed up COHERENT programs. Also, try using the shell to implement a prototype. You can see if it is too slow before coding. Often our intuitions are wrong in these matters. Also, the users might change their minds and decide on a different way to approach the problem. Your prototype will then have saved you a lot of unnecessary C coding.

If you do descend to C, try to write small programs that can be used in future shell programming. Only write what you need. Read from the standard input and write to the standard output. Make your programs table driven and use **/rdb** data-base table and list formats so that the full power of COHERENT and the data base can manipulate the tables that drive your programs.

system(): Tell Shell to Execute a Program

The easiest way for C programs to access both COHERENT and the **/rdb** data base is with the COHERENT function **system()**. It only needs a character string, or pointer to a character string that contains a shell command. **system()** will execute the command, just as though you typed the command at the terminal or entered it into a shell program.

In the example below are three different ways of setting up the command string:

```
#define COMMAND "ls | wc"

main (argc, argv)
int    argc;
char   *argv [];
{
       system ("echo hello word");
       system (COMMAND);
       system (argv [1]);
}
```

The first call to **system()** hard-codes the command in the code of the function call. The second call uses a previously defined name **COMMAND**. Note the line

```
       #define COMMAND
```

at the beginning of the code. Finally, the first argument on the command line is used as a command string for **system()**:

```
	system (argv [1]);
```

This allows the user to type a command as the first argument and it will be executed.

The COHERENT command **make** is used to compile the program. It uses the **Makefile**, whose contents are as follows:

```
system: system.o
	cc system.o -o system
```

Thus, typing the command

```
	make system
```

prints the following on the screen:

```
	cc -O -c system.c
	cc system.o -o system
```

This shows the steps COHERENT uses to compile and link your program. (For a full description **make**, the programmer's best friend, see its tutorial in your COHERENT manual.)

Now that the **system** program has been compiled, it can be run with a command as an argument:

```
	system date
```

This produces something like:

```
hello word
      7      7     65
Thu Apr  3 02:10:04 PST 1986
```

Note that the first system call produced the **hello world** line. The second call executed the defined command

```
	ls | wc
```

Note the output on the second line probably will differ on your system.

Finally, the date and time were displayed when the **date** argument to the **system** program was executed by the call to **system()**.

execl(): Execute a Call

The next step from simple to complex is the **execl()** function. This function will replace the current process with another program. It is usually called *chaining*. Once called, the calling process dies and is never reentered. The executed program takes over the process table entry of the calling program and all of the open input and output files. Here is an example.

```
#include	<stdio.h>
#define FIRST "First"

char	*p1 = "argument";
char	*p2;
```

```
main (argc, argv)
int    argc;
char   *argv [];
{
       p2 = "is:";
       fflush (stdout);
       execl ("/bin/echo", "echo", FIRST, p1, p2, argv[1], NULL);
}
```

When we run the command

```
       execl ARG1 ARG2
```

we see:

```
       First argument is: ARG1
```

All of the arguments to **execl()** are pointers to strings; you can have as many as you want. The last one must be NULL, as a sentinel or end marker.

In the first argument, the full path to the program to be executed must be given because **execl()** does not search the **PATH** environmental variable. It is not good to hard code paths.

The second argument is the zero'th argument, which is the name of the called program. The rest of the arguments are the normal arguments as you would type on the command line at the shell level. Several possible ways of setting up the arguments are shown. The first regular argument is a defined constant. **p1** and **p2** are each pointers to strings. The first is initialized to a string when it is declared and the second is assigned later in the code. Finally, the first argument passed to the calling program is passed on through the second pointer in the **argv** array of pointers.

execl() Shell Programs

If you try to use **execl()** to execute a shell script file, it will fail. You have to execute a **sh** shell program to read the text of the shell file. Here a shell file called **listargs** is executed. It simply echoes out its arguments separated by newlines:

```
for I in $*
do
       echo $I
done
```

Now, the program **execl.sh.c**, which executes **listargs** via **execl()**:

```
#include     <stdio.h>
#define FIRST "First"

char   *p1 = "argument";
char   *p2;

main (argc, argv)
int    argc;
char   *argv [];
{
       p2 = "is:";
       fflush (stdout);
       execl ("/bin/sh", "sh", "listargs", FIRST, p1, p2, argv[1], NULL);
}
```

When executed with the command

```
       execl.sh arg1
```

you see:

```
First
argument
is:
arg1
```

Note that **sh** is the program executed, and that its first argument is **listargs**. This is like typing:

```
sh listargs
```

fork(): Create a Child Process

Sometimes we want to run another program without killing ourselves. The system call **fork()** creates a child process that is identical to the calling program. The return value of the **fork()** command can be tested to see which process the code is in, parent or child. Therefore, the same program, with the same code, can branch to different statements, depending upon whether it is the parent or child process. The child code can do its thing, including executing another program. The parent process can wait for the child to finish before going on. But it does not have to, because the COHERENT system is a multi-processing environment.

The following program, **fork.c**, gives an example of using **fork()** in a C program:

```
#include     <stdio.h>

#define      FAIL   -1
#define      CHILD  0
#define      PARENT 1

main (argc, argv)
int    argc;
char   *argv [];
{

       switch (fork()) {
       case -1:     /* error */
              fprintf (stderr, "fork system call failed.\n");
              exit (FAIL);

       case 0:             /* child process */
              execl ("/bin/echo", "echo", "I am the child.", NULL);
              exit (CHILD);

       default:     /* parent process */
              if (wait(NULL) != -1) {
                     fflush (stdout);
                     printf ("I am the parent.\n");
              } else {
                     fflush (stdout);
                     fprintf (stderr, "wait system call failed.\n");
              }
       }

       exit (PARENT);
}
```

When this program is compiled and invoked, you see the following on the standard output:

TUTORIALS

```
I am the child.
I am the parent.
```

This example uses the C **switch()** keyword execute **fork()**, because it has three different return values. A **-1** return indicates a failure to fork a new process. Perhaps the number of processes has reached a limit. We want to handle this error condition.

A zero return means that the code is now in a child process, so the code should do the child's thing. In this case, the child process executes an **echo** command with **execl()**. Thus, it changes into another program.

Finally, the parent process gets the child's process ID number, which is some number greater than zero. Here the parent waits for the child to die before printing its message. It flushes the standard-out buffers before printing its own message.

You can see when we execute this program, both processes write to the standard output.

Pipes

A very important use of this mechanism is to use the **pipe()** system call to open a pipe between the two processes so that they can exchange data. This can go either way, from parent to child, or from child to parent.

One-Way Pipe

The first example here is of a one-way pipe. The next section will demonstrate a two-way pipe.

```
#include     <stdio.h>

#define      FAIL   -1
#define      CHILD  0
#define      PARENT 1
#define      MESSAGE      "Hi, kid"

main (argc, argv)
int    argc;
char   *argv [];
{
       int    pd [2];
       char   childbuf [BUFSIZ];

       if (pipe(pd) == FAIL) {
              fprintf (stderr, "pipe system call failed.\n");
              exit (FAIL);
       }

       switch (fork()) {
       case -1:     /* error */
              fprintf (stderr, "fork system call failed.\n");
              exit (FAIL);

       case 0:             /* child process */
              if (close (pd [1]) == FAIL) {
                     fprintf (stderr, "close pipe failed.\n");
                     exit (FAIL);
              }
```

```
                read (pd [0], childbuf, BUFSIZ);
                execl ("/bin/echo", "echo",
                        "child read:", childbuf, NULL);
                exit (CHILD);

        default:        /* parent process */
                if (close (pd [0]) == FAIL) {
                        fprintf (stderr, "close pipe failed.\n");
                        exit (FAIL);
                }

                printf ("parent wrote: %s\n", MESSAGE, sizeof(MESSAGE));
                if (write (pd[1], MESSAGE, sizeof(MESSAGE)) == FAIL) {
                        fprintf (stderr, "pipe write filed\n");
                        exit (FAIL);
                }
        }
        exit (PARENT);
}
```

When compiled and run, this program prints the following on the standard output:

```
parent wrote: Hi, kid
child read: Hi, kid
```

pd is the pipe-descriptor buffer, which holds two integers, one for each pipe direction. The parent writes into **pd[1]** and the child reads from **pd[0]**. Each closes its other pipe with the system call **close()**. **fork()** creates the child process. The parent process writes its message to the pipe with the system call **write()**. It used the C **sizeof** operator to count the characters of the **MESSAGE**. This allows us to edit the message and not have to look through the code to find, count and change its size.

The child process reads the pipe with the system call **read()** and executes the **echo** command to display the message it got from the parent process.

Of course this is not very useful, but it is simple enough to follow. It can be used in any situation in which we want to crank up a background process and periodically send it data to act upon. This mechanism is especially useful in COHERENT because there are so many programs that are available to save us the effort of writing them.

Pipe to Standard Input

In the example above, **write()** was used to read the pipe and the message was passed on to the executed **echo** program as an argument. But we also want to write to the standard input of a COHERENT program. In the example below, the child executes a process that reads from its standard input. The child connected the pipe to its standard input using the system call **dup()**:

```
#include      <stdio.h>

#define       FAIL   -1
#define       CHILD  0
#define       PARENT 1
#define       PM     "Hi, kid"
#define       CM     "Hi, parent"
```

```
main (argc, argv)
int    argc;
char   *argv [];
{
       int    pd [2];
       char   buffer [BUFSIZ];

       if (pipe(pd) == FAIL) {
              fprintf (stderr, "pipe system call failed.\n");
              exit (FAIL);
       }

       switch (fork()) {
       case -1:      /* error */
              fprintf (stderr, "fork system call failed.\n");
              exit (FAIL);

       case 0:              /* child process */
              if (close () == FAIL) {
                     fprintf (stderr, "close pipe failed.\n");
                     exit (FAIL);
              }

              if (dup (pd [0]) != 0) {
                     fprintf (stderr, "dup pipe failed.\n");
                     exit (FAIL);
              }

              if (close (pd [0]) == FAIL || close (pd [1]) == FAIL) {
                     fprintf (stderr, "dup pipe failed.\n");
                     exit (FAIL);
              }

              execl ("/bin/cat", "cat", NULL);
              fprintf (stderr, "execl cat failed.\n");
              exit (FAIL);

       default:      /* parent process */
              if (close (pd [0]) == FAIL) {
                     fprintf (stderr, "close pipe failed.\n");
                     exit (FAIL);
              }

              printf ("parent wrote: %s\n", PM, sizeof(PM));
              if (write (pd[1], PM, sizeof(PM)) == FAIL) {
                     fprintf (stderr, "pipe write filed\n");
                     exit (FAIL);
              }

              if (close (pd [1]) == FAIL) {
                     fprintf (stderr, "close pipe failed.\n");
                     exit (FAIL);
              }
```

```
/*
              read (pd [0], buffer, BUFSIZ);
              printf ("parent read: %s\n", buffer);
*/
       }
       exit (PARENT);
}
```

When compiled and run, this program writes the following to the standard output:

```
parent wrote: Hi, kid
Hi, kid
```

This is similar to the example before, except that the **cat** command reads from its standard input. The parent process then uses the system call **close()** to close the pipe after writing. This sends an end-of-file character to the pipe and the standard input of the **cat** program, so that the program will terminate.

Two-Way Pipe

In addition to sending data one way, we sometimes want to get replies. Two-way pipes make possible co-routines that exchange data. The following gives an example of how to implement a two-way pipe:

```
/*
pipe2 - sends first argument thru pipe to child program,
       and reads the childs replay and sends it to standard-out.
*/

#include     <stdio.h>

/* program return status */
#define      OK     0
#define      FAIL   -1

/* fork return codes distinguished child from parent process */
#define      CHILD  0
#define      PARENT 1

/* pipe descriptor array elements */
#define      READ   0
#define      WRITE  1

/* standard-io file descriptors */
#define STDIN 0
#define      STDOUT 1
#define      STDERR 2

/* program for child to execl */
#define      PATH   "/usr/bin/bc"
#define      PROGRAM      "bc"

/* standard-error handling macro */
#define      error(message) {fprintf (stderr,"%s: %s\n", argv [0], message);\
                     exit (FAIL); }

char   buffer [BUFSIZ];    /* buffer to read into */
```

TUTORIALS

```
main (argc, argv)
int    argc;
char   *argv [];
{
       register char *p;    /* pointer for scanning strings */
       int    pcpipe [2];   /* pipe from parent to child */
       int    cppipe [2];   /* pipe from child to parent */

       if (pipe (pcpipe) == FAIL || pipe (cppipe) == FAIL)
              error("pipe system call failed.");

       switch (fork()) {
       case FAIL:   /* error */
              error("fork system call failed");

       case CHILD:  /* child process */

              /* close stdin to free for pipe connection */
              if (close (STDIN) == FAIL)
                     error("close stdin failed");

              /* connect pipe to stdin of child */
              if (dup (pcpipe [READ]) != STDIN)
                     error("dup stdin pipe failed");

              /* close stdout to free for pipe connection */
              if (close (STDOUT) == FAIL)
                     error("close stdout failed");

              /* connect pipe to stdout of child */
              if (dup (cppipe [WRITE]) != STDOUT)
                     error("dup stdout pipe failed");

              /* close all pipes so child reads from stdin/out */
              if (   close (pcpipe [READ])  == FAIL ||
                     close (pcpipe [WRITE]) == FAIL ||
                     close (cppipe [READ])  == FAIL ||
                     close (cppipe [WRITE]) == FAIL )
                     error("close child pipes failed");

              /* execute basic calculator */
              execl (PATH, PROGRAM, NULL);

              /* if we get here, execl failed */
              error("execl failed");

       default:     /* parent process */

              /* close unused ends of the pipes */
              if (   close (pcpipe [READ])  == FAIL ||
                     close (cppipe [WRITE]) == FAIL )
                     error("parent close pipe failed");
```

```
                /* write command line argument into pipe to child */
                if (write (pcpipe [WRITE], argv [1], strlen(argv [1]))
                        == FAIL)
                        error("parent pipe write argv failed");

                /* bc needs a newline to execute a line */
                if (write (pcpipe [WRITE],"\n", 1) == FAIL)
                        error("parent pipe write nl failed");

                /* close pipe sends end-of-file to kill child */
                if (close (pcpipe [WRITE]) == FAIL)
                        error("parent close pipe failed");

                /* read childs reply */
                if (read (cppipe [READ], buffer, sizeof(buffer))
                        == FAIL)
                        error("parent pipe read failed");

                /* be sure string is terminated by NULL */
                for (p = buffer; *p != '\n'; p++);
                *p = NULL;

                /* output the answer */
                puts (buffer);

                exit (OK);
        }
}
```

When this program is compiled and run with the command

```
pipe2 '2+2'
```

you'll see the following on the standard output:

```
4
```

Likewise, the command

```
pipe2 'scale=2 ; (100/3) * 5'
```

prints:

```
166.65
```

The **bc** command is the COHERENT calculator program. It reads equations from the standard input and writes solutions to the standard output. Our program reads an equation from its first command-line argument, and writess it through one pipe to a child process. The child process connects pipes to its standard input and standard output, then executes **bc**. **bc** reads the equation and sends the answer to the parent through the second pipe. The parent reads the second pipe and sends the solution to its standard output.

This program is not very useful, but is a template. You can use this mechanism to give your C programs access to all of the COHERENT, **/rdb**, and other programs as if they were subroutines. This will save you a lot of programming. If a program exists to do what you want, use it.

Programming Style

TUTORIALS

The program above also introduces a more advanced programming style. Most *magic* numbers have been defined to mnemonic constants to make the code easier to read and to simplify changes. A zero appearing in the code usually means nothing to the reader, but a name helps us understand what it is. In addition, if you have to change a magic number, it is much easier to edit it once at the top of the code than to search through all of the code to find it. It is especially difficult when the magic number is a common number like zero or one. You can not globally change such numbers, because they are used in many different ways. You must carefully search the code for the correct numbers to change.

Notice that the error condition is now handled by a simple macro called **error()**. It saves a lot of typing, makes the code easier to read, and standardizes the way errors are handled and reported.

Finally, every major section of code is commented. It takes a little more time, but saves a lot of time when we are debugging, changing, and maintaining the program.

Fast Access

The final example shows a more practical program. This **fastaccess** program creates the **search** program and sends it a key through the first pipe. This child process finds a row in the **inventory** table and sends it back through the second pipe to the parent. In this example, the parent merely writes the retrieved row to the standard output, but you can modify the parent to do other things to the row.

```
/*
fastaccess - sends first argument thru pipe to child seek program,
       and reads the childs offset replay, seeks the record,
       reads it and sends the record to standard-out.
*/

#include     <stdio.h>

/* program return status */
#define      OK     0
#define      FAIL   -1

/* fork return codes distinguished child from parent process */
#define      CHILD  0
#define      PARENT 1

/* pipe descriptor array elements */
#define      READ   0
#define      WRITE  1

/* standard-io file descriptors */
#define STDIN 0
#define      STDOUT 1
#define      STDERR 2

/* program for child to execl */
#define      PATH   "/usr/rdb/bin/search"
#define      PROGRAM      "search"
#define      METHOD "-mb"
#define      TABLE  "inventory"
#define      KEY    "Item"
```

```
/* standard-error handling macro */
#define      error(message) {fprintf (stderr,"%s: %s\n", argv [0], message);\
                     exit (FAIL); }

char   buffer [BUFSIZ];    /* buffer to read into */

main (argc, argv)
int    argc;
char   *argv [];
{
       register char *p;    /* pointer for scanning strings */
       int    pcpipe [2];  /* pipe from parent to child */
       int    cppipe [2];  /* pipe from child to parent */
       int    file;        /* file descriptor for table file */
       int    from, to;    /* offsets of record in table */
       int    xfrom, xto;  /* offsets in secondary index file */

       if (pipe (pcpipe) == FAIL || pipe (cppipe) == FAIL)
              error("pipe system call failed.");

       switch (fork()) {
       case FAIL:   /* error */
              error("fork system call failed");

       case CHILD:  /* child process */
              /* close stdin to free for pipe connection */
              if (close (STDIN) == FAIL)
                     error("close stdin failed");

              /* connect pipe to stdin of child */
              if (dup (pcpipe [READ]) != STDIN)
                     error("dup stdin pipe failed");

              /* close stdout to free for pipe connection */
              if (close (STDOUT) == FAIL)
                     error("close stdout failed");

              /* connect pipe to stdout of child */
              if (dup (cppipe [WRITE]) != STDOUT)
                     error("dup stdout pipe failed");

              /* close all pipes so child reads from stdin/out */
              if (   close (pcpipe [READ])  == FAIL ||
                     close (pcpipe [WRITE]) == FAIL ||
                     close (cppipe [READ])  == FAIL ||
                     close (cppipe [WRITE]) == FAIL )
                     error("close child pipes failed");

              /* execute program */
              execl (PATH, PROGRAM, METHOD, TABLE, KEY, NULL);

              /* if we get here, execl failed */
              error("execl failed");
```

TUTORIALS

```
        default:     /* parent process */
               /* close unused ends of the pipes */
               if (   close (pcpipe [READ])  == FAIL ||
                      close (cppipe [WRITE]) == FAIL )
                      error("parent close pipe failed");

               /* write command line argument into pipe to child */
               if (write (pcpipe [WRITE], argv [1], strlen(argv [1]))
                           == FAIL)
                      error("parent pipe write argv failed");

               /* needs a newline to execute a line */
               if (write (pcpipe [WRITE],"\n", 1) == FAIL)
                      error("parent pipe write nl failed");

               /* close pipe sends end-of-file to kill child */
               if (close (pcpipe [WRITE]) == FAIL)
                      error("parent close pipe failed");

               /* read childs reply */
               if (read (cppipe [READ], buffer, sizeof(buffer))
                           == FAIL)
                      error("parent pipe read failed");

               /* output the answer */
               puts (buffer);

               exit (OK);
        }
}
```

After you compile this program, the command

```
        fastaccess 2
```

prints the following on the standard output:

```
Item   Amount Cost   Value  Description
----   ------ ----   -----  --------------
   2      100    5     500  test tubes
```

This is similar to the last example. But here, several arguments are given to the **search** program. These are defined with the **#define** preprocessor statements, but they could be passed from the command line arguments of the parent program or created within the program, or read from another file, a technique which is called *table driven* programming.

The best way to program is to write simple code that gets its parameters and data from data-base tables. Then the programs can be easily modified by simply editing the driving data tables, without having to reedit and recompile the source code. Users can modify these programs without calling back the programmer.

tabletostruct: Convert a Table to a C struct

A useful **/rdb** command is **tabletostruct**. It converts an **/rdb** table to a C-language **struct** type and initializes it to the values in the table. You can compile this **struct** into your C program and access any field. This makes possible table-driven code. You can use the data base to enter and update your tables, and recompile the program when you wish. This is the fastest access to tables, but requires compiling, and the inflexibility of not being able to change the programs internally compiled tables while the program is running.

Here we start with a simple table, convert it to **struct**, make it a header file, and compile it into a simple program that prints out each field.

To begin, we create a table called **table**, which appears as follows:

```
A       B       C
-       -       -
1       2       3
```

Now, we use **tabletostruct** to covert **table** to a **struct**:

```
        tabletostruct Table table < table > table.h
```

The output is written into the header file **table.h**, which appears as follows:

```
struct Table
{
char    *A;
char    *B;
char    *C;
}       table [] =
{ "1","2","3" }
;
```

The following program displays the **struct**ed contents of **table.h**:

```
#include      "table.h"

main ()
{
        printf ("A=%s\n", table[0].A);
        printf ("B=%s\n", table[0].B);
        printf ("C=%s\n", table[0].C);
}
```

When compiled and run, the program prints the following onto the standard output:

```
A=1
B=2
C=3
```

Note that the **struct** is an array and that we have to give a subscript for each row we want. We can also put the **tabletostruct** into the **Makefile** to further automate the re-compilation:

```
printtable:   table.h printtable.o
        cc -o printtable printtable.o

table.h:      table
        tabletostruct Table table < table > table.h
```

This says that the printable program depends upon **table.h** being up to date, and **table.h** depends upon **table** being up to date. If you modify **table** since the last compile, **make** will re-execute the **tabletostruct** command.

Our next example begins with a bigger table, called **inventory**. The command

```
        tabletostruct Inventory inventory < inventory
```

writes the following to the standard output:

TUTORIALS

```
struct Inventory
{
char   *Item;
char   *Amount;
char   *Cost;
char   *Value;
char   *Description;
}      inventory [] =
{ "1","     4","  50","  150","rubber gloves" }
,{ "2","   100","   5","  500","test tubes" }
,{ "3","     5","  80","  400","clamps" }
,{ "4","    23","  19","  437","plates" }
,{ "5","    99","  24"," 2376","cleaning cloth" }
,{ "6","    89"," 147","13083","bunsen burners" }
,{ "7","     5"," 175","  875","scales" }
;
```

You can refer to the **Description** of the third item with this code:

```
printf ("Item Name = %s\n", inventory [2].Description);
```

Remember that tables in C start with element 0. You can convert the strings to integers with the function **atoi()**:

```
int    item;        /* current inventory item number */
int    row;         /* current row number starting from 0 */

item = atoi (inventory [row].Item);
```

Read Table into Memory: getfile() and fsize()

To get tables from the data base at run time, you can read the table into memory and set up an array of pointers to reference any field by row and column number. Use the COHERENT system calls **open()** and **stat()** to get the size of the table, and the function **malloc()** to get that much memory. Then use **read()** to copy the whole table into memory and run through it with a loop that sets a two-dimentional array of pointers to point to every field in the array. Then you can reach any field with:

```
char   *field, *p_table [][];
int    row, column;

field = p_table [row][column];
```

To keep from having to hard code the size of your pointer table, first find out how many columns and rows you have by running through the memory counting newline characters. You might turn tabs and newlines into NULs while you go. Then **malloc()** enough memory to hold this array of pointers and run through the memory again to set the pointer array to pointing to fields.

Here are some routines that show examples of how to do this.

```
/*
Copyright (c) 1990, 1991 Schaffer and Wright
getfile - reads a file into memory and returns pointer and size
*/

#include     "rdb.h"
#include     <sys/types.h>
#include     <sys/stat.h>
```

```
char   *filebuffer, *malloc();
int    file, open(), read();
long   fsize ();

char   *getfile (filename, p_size)
char          *filename;
unsigned      *p_size;
{
       *p_size = 0;

       /* open the file */
       if ((file = open (filename, 0)) < 0) {
              fprintf (stderr, "Can't open file %s.\n", filename);
              fflush (stderr);
              perror ("getfile");
              return (NULL);
       }

       /* get the memory to hold the table */
       /* get size adding a byte for a trailing NUL */
       *p_size = (unsigned) (fsize (file) + 1); /* byte for NUL end */

       /* get a buffer in memory for the system */
       if ((filebuffer = malloc ((unsigned) *p_size)) == NULL) {
              fprintf (stderr,
                     "Can't malloc the size of file %s.\n", filename);
              fflush (stderr);
              perror ("getfile");
              return (NULL);
       }

       /* read in the file and reset p_size to number of bytes read */
       if ((*p_size = (unsigned) read (file, filebuffer, *p_size)) < 1) {
              fprintf (stderr, "Can't read file %s.\n", filename);
              fflush (stderr);
              perror ("getfile");
              return (NULL);
       }

       /* write NULL at the end of the buffer */
       *(filebuffer + *p_size) = NULL;

       return (filebuffer);
}
```

getfile() calls **fsize()**, which gets the file size from the operating system via the system call **fstat()**:

```
/*
Copyright (c) 1990, 1991 Schaffer and Wright
fsize  get the size of a file by calling the fstat routine
*/

#include     "rdb.h"
#include     <sys/types.h>
#include     <sys/stat.h>
```

TUTORIALS

```
extern int    Debug;       /* global for debugging */

long   fsize (file)
int    file;
{
       int    fstat ();           /* system call */
       struct stat   statusbuffer; /* info on file */

       /* get status information including size */
       if ((fstat (file, &statusbuffer)) < 0) {
              fprintf (stderr,
                     "fsize: Can't get the size of file %d\n", file);
              perror ("fsize");
              return (FAIL);
       }

       if (Debug)
              fprintf (stderr, "fsize: file size=%d\n",
                     statusbuffer.st_size);

       /* return size */
       return ((long) statusbuffer.st_size);
}
```

/rdb Functions: librdb.a

In the directory, **$RDB/lib** is the archive **librdb.a**, which contains all of the functions called by the **/rdb** programs. You can call them like any function if you link the archive into your program. For example, the command

```
cc -o prog prog.c librdb.a
```

compiles program. **prog.c** into executable **prog**, and links the contents of archive **librdb.a** into the final executable.

If you, or your system administrator, moves the file to directory **/usr/lib** you only need to add **-lrdb** to you compile line:

```
cc -o prog prog.c -lrdb
```

Colroutines

Another file in **$RDB/lib** is named **Colroutines**. "Col" stands for "column," because the functions largely handle the columns of a table. **Colroutines** contains documentation on each of the C functions in **librdb.a**. It is the header of the source code for each routine. It shows the description, function name, arguments and argument types. It should be enough information to use the function, without being able to see the code.

Display Example

The example **display.c** calls the **/rdb** functions:

```
static char  Copyright []="Copyright (c) 1990, 1991 Schaffer and Wright";
/*
display will read a table or list file and send to standard-out
*/
```

```
#define      USAGE  "usage: display < tableorlist\n"
#include "rdb.h"

int    Debug; /* global for debugging */

struct rowstruct row;                    /* row  information */
int    colgeth (), colgetr ();           /* input   functions */
int    colputh (), colputr ();           /* output  functions */
int    colinit (), coldump ();           /* utility functions */

main (argc, argv, envp)
int argc;
char *argv [];
char *envp [];
{
       register     columns;     /* columns returned */
       register     i;            /* index for loops  */

       /* handle command line arguments */
       for (i = 1; i < argc; i++) {
              if (argv [1][0] == '-') {
                     switch (argv [1][1]) {

                     case 'D':
                            if (argv [1][2] != EOS) {
                                   Debug = atoi (&(argv [1][2]));
                            } else {
                                   Debug = TRUE;
                            }
                            break;
                     /* add other options here */

                     default:
                            break;
                     }

              } else {
                     /* get non-option arguments, like files, here */
              }
       }

       /* get table or list headlines */
       if ((columns = colgeth (&row)) == EOF)
              exit (EOF);

       /* output headlines */
       colputh (&row);

       /* read in each row till end-of-file */
       while ((columns = colgetr (&row)) != EOF) {
              /* output each row, or do other row processing */
              colputr (&row);
       }
```

```
        /* return status code becomes shell $? variable for testing */
        exit (OK);
}
```

Compile it with the command:

```
cc display.o -o display librdb.a
```

This assumes that **librdb.a** is in the current directory. If it is not, use its full path name.

When invoked with the command

```
display < inventory
```

display displays the following on the standard output:

```
Item   Amount   Cost   Value   Description
----   ------   ----   -----   --------------
   1        3     50     150   rubber gloves
   2      100      5     500   test tubes
   3        5     80     400   clamps
   4       23     19     437   plates
   5       99     24    2376   cleaning cloth
   6       89    147   13083   bunsen burners
   7        5    175     875   scales
```

Likewise, the command

```
display < maillist
```

displays the following on the standard output:

```
Number   1
Name     Ronald McDonald
Company  McDonald's
Street   123 Mac Attack
City     Menphis
State    TENN
ZIP      30000
Phone    (111) 222-3333

Number   2
Name     Chiquita Banana
Company  United Brands
Street   Uno Avenito De La Revolution
City     San Jose
State    Costa Rica
ZIP      123456789
Phone    1234
```

The first line is the copyright notice. It is declared to be a **static char** so that it will appear in the object module and the binary code.

The next comment is a description of the purpose of the program. The **USAGE** definition gives the syntax of the program. It can be displayed when a syntax error is detected. The debugging system is described in the next section.

The **row** structure is defined in the **rdb.h** header file that can be included in your programs. **row** contains lots of information about the headline and rows of the input table or list file. It is passed to the various **col** functions, which are declared below **row**.

After **main()**, the command line arguments are handled. Only the **Debug** variable is set, but this code can be edited to get all expected options and other arguments.

The **col** functions read in the headline and display it, followed by each row in the table or list.

The **return** command returns an exit status so that the program can be tested with the shell's **if**, **while**, and **for** statements.

You can copy this program to start your own program. You can process each row of the table as it is read in by inserting your code within the **while** loop. Have fun, but remember, shell programming is much easier.

Debugging

The **Debug** variable is an external global variable that can be seen by all of the functions. The advantage of this method of debugging is that it is in the final product so that diagnostics can be run when errors are found by users and service people.

On the command line one can put **-D** to turn on debugging traces. These messages are printed out as the program runs to show the value of certain key variables. To control the quantity of output, one can follow the option with a number. The higher the number, the more output. **-D9** turns on all output, which is a lot to wade through, but shows everything.

In the example below, all diagnostics are dumped. Each function gives its name and the value of variables. **colgeth()** is the column-get-header function which calls **colinit()** to initialize the row buffers.

The COHERENT function **malloc()** is invoked to get memory for head and row data. This is part of the dynamic buffering that allows the programs to get as much memory as they can to handle large heads and rows. In this way the **/rdb** programs are not limited by software, but only by available hardware memory.

Because pointers are often a problem in C, several pointers are printed out. Their values don't mean much, but can be checked to see that they have been set to reasonable numbers. **coldump** lists all of the variables and their values in the row structure. You can also call **coldump()** from your program if you wish. (Of course, you can use the COHERENT debugger **db**, if you prefer.)

colgetr() also displays the columns as they are read, so that you can see what it is seeing. All of this is coming to you through the standard error (aka **stderr**), so that you can redirect it. It is not buffered, so it comes out before the table which is coming through the buffered standard output, and is not sent until the buffer is full or the stream is closed on program termination.

The command

```
display -D9 < inventory
```

prints the following to the standard error:

```
colgeth: row=14976 filein=0
colinit: row=14976
colinit: bufsize=2048
colinit: colsize=2
colinit: p_buffer allocated
colinit: maxcolumns=1024
colinit: p_heads allocated
colinit: p_columns allocated
colinit: p_collengths allocated
coldump:---------------------------------------
       row pointer
row=14976
```

TUTORIALS

```
       file pointer for input file
row->p_filein=14754
       fileout pointer for input fileout
row->p_fileout=0
       points to buffer to write into
row->buffer=16224
       points to (offset) buffer to write into
row->p_buffer=16224
       buffer size
row->bufsize=2048
       points to end of buffer
row->p_endbuffer=18272
       points to each head found in lists
row->p_heads=18274
       to number of head columns found
row->heads=0
       points to each column found
row->p_columns=20324
       number of columns
row->columns=0
       number of colsize
row->colsize=2
       number of maxcolumns
row->maxcolumns=1024
       each column's length
row->collengths=22374
       entire row length
row->rowlength=0
       boolean fixed or variable
row->fixed=0
       boolean list or table
row->list=0
coldump:--------------------------------------
Item
Amount
Cost
Value
Description
colgetr: end-of-row.nrow->rowlength = 37
colgeth: row->heads=5
----
------
----
-----
--------------
colgetr: end-of-row.nrow->rowlength = 37
colgeth: row->columns=5 length=38 fixed=0 list=0
colputh: row->p_fileout=0
   1
     3
  50
  150
rubber gloves
colgetr: end-of-row.nrow->rowlength = 36
```

```
   2
   100
   5
  500
test tubes
colgetr: end-of-row.nrow->rowlength = 33
   3
     5
  80
  400
clamps
colgetr: end-of-row.nrow->rowlength = 29
   4
    23
  19
  437
plates
colgetr: end-of-row.nrow->rowlength = 29
   5
    99
  24
 2376
cleaning cloth
colgetr: end-of-row.nrow->rowlength = 37
   6
    89
 147
13083
bunsen burners
colgetr: end-of-row.nrow->rowlength = 37
   7
     5
 175
  875
scales
colgetr: end-of-row.nrow->rowlength = 29
colgetr: eof
Item   Amount   Cost   Value   Description
----   ------   ----   -----   -------------
   1        3     50     150   rubber gloves
   2      100      5     500   test tubes
   3        5     80     400   clamps
   4       23     19     437   plates
   5       99     24    2376   cleaning cloth
   6       89    147   13083   bunsen burners
   7        5    175     875   scales
```

We did warn you that there was a lot of it.

Fast-Access Example

The program **inverted.c**, found in directory **$RDB/lib**, demonstrates how to use the inverted method from within a C program:

TUTORIALS

```
/* inverted.c is a sample program which illustrates the usage of the
   fast access librdb.a routines (index and search).
   specifically, this program exercises xinverted() and sinverted().
   the search and index programs do essentially the same thing for
   all fast access methods, and provide more command line options.
   The following compile command assumes that the include file rdb.h and
   library archive (distributed in the lib directory of rdb's home dir)
   are in the current directory:  "cc -O inverted.c librdb.a -o xinverted"
   Link the resulting xinverted binary to sinverted.nCall xinverted to
   perform indexing and sinverted to perform searching.
*/

char   Copyright[] = "Copyright (c) 1990 by Schaffer and Wright";

/*
index  index the given file according to the method specified.
       It calls the xinverted (for index) routine.
search search the given file using the method specified.
       It calls the sinverted (for search) routine.
*/

#define      USAGE "usage: %s [-n -x] tableorlist [keycolumn ...] [ < keytable ]\n"

/*
-n = numeric comparison of key and column value
-x = partial match and upper or lower case insensitive
 */

#include     "rdb.h"
int    Debug;              /* global for debugging  */
char   *Program;           /* argv [0] program name */
char   *Function = "main"; /* current function name */

/* error message table(make into include and automatic error handling)*/
#define      NOTABLE             1
#define      EOPTION             2
#define      NOMEM        3
#define      NOFILENAME   4
#define      NOFILE       5
#define      NOPUT        6
#define      NOKEYHEADS   7

/* structures */
struct rowstruct rowfile;
struct rowstruct rowkey;
struct keystruct key;

/* functions called */
int    sinverted ();
int    xinverted ();

int    (*method)();        /* method function pointer */
```

```
/* comparison routines */
int    colcmp ();          /* default: ignore blanks */
int    collook ();         /* -x partial match, case insensitive */
int    numcmp ();          /* -n numeric comparison */
int    strcmp ();          /* -x exact character match */

/* flags */
int    foldcase = FALSE;   /* -x sort fold upper and lower case */

main (argc, argv, envp)

int    argc;
char   *argv [];
char   *envp [];
{

       register     a;     /* index for arguments */
       register     r;     /* row found index */
       char   *p;          /* character pointer */
       int    rows=0;             /* number of rows matched */
       int    search=FALSE;/* search or index */
       int    matches;     /* columns matched */
       FILE   *fopen ();   /* functions used */
       char   *rindex();

       /* get arguments */
       if (Debug) fprintf (stderr, "argc=%d\n", argc);
       if (argc < 2) {
              fprintf (stderr, USAGE, argv [0]);
              exit (NOTABLE);
       }

       /* initialize key structure */
       if (keyinit (&key) == FAIL)
              exit (NOMEM);
       key.singlerow = FALSE;

       /* initialize rowkey structure */
       if (colinit (&rowkey) == FAIL)
              exit (NOMEM);

       /* initialize: called by search or index  */
       /* handle path names, walk to end looking for slash */
       if (p = rindex(argv[0], '/')) ++p;
       else p = argv[0];

       /* if the command name begins with 's' assume search procedures */
       if (*p == 's')
              search = TRUE;      /* search */
       else   search = FALSE;     /* index  */

       if (search)
              method = sinverted;
       else   method = xinverted;
```

TUTORIALS

```
/* default comparison method is ignore blanks */
key.compare = colcmp;

/* loop through arguments looking for flags and file name */
for (a = 1; a < argc; a++) {
       if (Debug)
              fprintf (stderr, "argv [%d]=%s\n", a, argv [a]);

       if (argv [a][0] == '-') {
              switch (argv [a][1]) {

              case 'D':
                     if (isdigit(argv [a][2]))
                            Debug = atoi (&argv [a][2]);
                     else   Debug = TRUE;
                     break;

              case 'n':
                     key.compare = numcmp;
                     break;

              case 'x':
                     foldcase = TRUE;
                     key.compare = collook;
                     break;

              default:
                     fprintf (stderr,
                     "%s: %s option not recognized\n",
                     argv [0], argv [a]);
                     fprintf (stderr, USAGE, argv [0]);
                     exit (EOPTION);
              }

       } else {
              if (key.p_filename == NULL) {
                     key.p_filename = argv [a];
              } else {
                     rowkey.p_heads [rowkey.heads++] = argv [a];
              }
       }
}

/* do we have a file to open */
if (key.p_filename == NULL) {
       fprintf (stderr, "%s: no file name\n", argv [0]);
       fprintf (stderr, USAGE, argv [0]);
       exit (NOFILENAME);
}

/* open the file */
key.file = rowfile.p_filein = fopen (key.p_filename, READ);

rowfile.p_fileout = stdout;
```

```
if (rowfile.p_filein == NULL) {
       fprintf (stderr,
       "%s: Can not open %s\n", argv [0], key.p_filename);

       exit (NOFILE);
}

if (Debug)
       fprintf (stderr, "%s: rowfile.columns=%d\n",
              argv [0], rowfile.columns);

/* get the head line of the file */
if ((rowfile.columns = colgeth (&rowfile)) == EOF)
       exit (EOF);

if (Debug)
       fprintf (stderr, "%s: rowfile.columns=%d\n",
              argv [0], rowfile.columns);

/* put the head line of the file */
if (search) {
       if (colputh (&rowfile) == FAIL)
              exit (NOPUT);
}

/* if the key columns are not in the command line,
       get from the stdin */
rowkey.p_filein = stdin;
rowkey.p_fileout = stdout;

if (rowkey.heads == 0) {
       /* get from key head */
       if ((rowkey.heads = colgeth (&rowkey)) == EOF)
                  exit (EOF);
}

/* get first row if list */
if (rowfile.list)
       if ((rowfile.columns = colgetr (&rowfile)) == EOF)
              exit (EOF);

if (rowkey.list)
       if ((rowkey.columns = colgetr (&rowkey)) == EOF)
              exit (EOF);

/* find key columns in the file columns */
matches = colmatch (rowfile.p_heads, rowfile.heads,
       rowkey.p_heads, rowkey.heads, key.keytocolumn);

if (matches != rowkey.heads) {
       fprintf (stderr,
       "%s: Can not find your keys in column heads.\n",
       argv [0]);
```

TUTORIALS

```
        exit (NOKEYHEADS);
}

/* if index, go and index and then exit */
if (! search) {
        if (Debug) fprintf (stderr,
        "%s: rowfile.p_filein=%d\n", argv [0], rowfile.p_filein);

        /* call method */
        if ((rows = (*method) (&key, &rowfile, &rowkey)) < 0) {
                perror (argv [0]);
                exit (rows);
        }
        exit (OK);
}

/* get first row of keys if table */
if (Debug) {
        fprintf (stderr, "%s: before colgetr (&rowkey)\n",
                                argv [0]);
        colputh (&rowkey);
}

if (rowkey.list == FALSE)
        if ((rowkey.columns = colgetr (&rowkey)) == EOF)
                exit (NOKEYHEADS);

if (Debug) {
        fprintf (stderr,
                "%s: after colgetr (&rowkey)\n", argv [0]);
        colputh (&rowkey);
        colputr (&rowkey);
}

/* loop getting key(s) from stdin */
do {
        if (Debug) {
                fprintf (stderr,
                        "%s: Before method rowkey.columns=%d\n",
                        argv[0], rowkey.columns);
        }

        /* don't look if no keys */
        if (rowkey.columns == 0)
                continue;

        /* call method */
        rows = (*method) (&key, &rowfile, &rowkey);

        if (Debug) fprintf (stderr,
        "%s: After method rows=%d key.rowoffsets [0]=%d\n",
                argv[0], rows, key.rowoffsets [0]);
```

```
            if (rows <= 0) {
                  continue;
            }

            for (r = 0; r < rows; r++) {
                  if (fseek (rowfile.p_filein,
                        key.rowoffsets [r], BEGINNING)
                        == BADSEEK)
                  fprintf (stderr, "search: Bad seek.\n");

                  colgetr (&rowfile);

                  colputr (&rowfile);
            }
      }
      while ((rowkey.columns = colgetr (&rowkey)) != EOF);

      exit (OK);
}
```

Manual Pages

The following gives manual pages for all **/rdb** commands. These include the commands used by the **/act** accounting system, which is included as an example of **/rdb** programming.

accounting terms — Definition

The following defines several of the more commonly used accounting terms. These are included to help you understand the descriptions of **/rdb**'s accounting package.

Financial information that records the day-to-day operation of a business is recorded in an *account*. A simple account is made up of three parts:

1. A title that describes the name of the information being stored in the account;
2. The left side of the account, used to record a debit; and
3. The right side of the account, used to record a credit.

This simple configuration is referred to as a 'T' account because its shape resembles a capital T:

```
                  Title
==================================
    Debit         |      Credit
```

A *ledger* is the combination of all the accounts maintained by a business. Transactions are posted to accounts through a *journal entry*, as in the following example:

```
Cash                |     500.00   |
      Sales         |              |   500.00
```

Entries recorded on the far left are debits; entries indented from the left are credits.

Accounts fall into the following classifications:

Assets Anything that is owned and has a monetary value.

Liabilities
Amounts owed to others.

Capital This is the owner's equity in the business. It is equal to the total assets minus the total liabilities.

Revenue
Proceeds obtained in the course of doing business.

Expenses
Costs incurred in the course of doing business.

Further information on accounting terms may be obtained from any textbook on accounting principles. Your local public library should contain several examples.

See Also

act

Notes

Please note that **/rdb**'s accounting system is meant to serve as an example of data-base programming with **/rdb**. It is not designed to be an exhaustive accounting package, nor is it designed to teach you how to perform accounting. Mark Williams technical support will help you if you have a problem with an **/rdb** command, but not with problems that involve the principles of accounting. *Caveat utilitor*.

act — /rdb Command

List all /act commands

act

/rdb comes with a set of commands that implement a basic accounting system, suitable for running a small business. These commands, called **/act** commands, are kept in directory **$RDB/act**.

The accounting system consists of the following sub-systems:

gl General ledger

inv Inventory

opr Operations, for manufacturing

pay Payroll

pur Accounts payable and purchasing

sales
Accounts payable and sales

The command **act** lists all **/act** commands.

Example

Typing **act** displaying the following on your screen:

```
act          close        income       postap    shorttoaccount
adjust       consolidate  invoice      postar    start
balance      cstate       makecatalog  postpay   tax.calc
bom          fillform     onhand       purchase  trial
calculate    foot         po           sale      vstate
chartdup     getjournal   post         ship      w2
```

Many **act** commands have their own entries in this manual.

See Also

accounting terms

Notes

Please note that **/rdb**'s accounting system is meant to serve as an example of data-base programming with **/rdb**. It is not designed to be an exhaustive accounting package, nor is it designed to teach you how to perform accounting. Mark Williams technical support will help you if you have a problem with an **/rdb** command, but not with problems that involve the principles of accounting. *Caveat utilitor*.

LEXICON

addcol — /rdb Command

Add a column to a table
addcol *newcolumn* **<** *table*

The command **addcol** appends one or more new columns to *table*. The column is initialized to blank spaces, and is given the name *newcolumn*.

Example

The command

```
addcol New < inventory
```

modifies table **inventory** to appear as follows:

```
Item  Amount  Cost    Value   Description     New
----  ------  ----    -----   -------------  -------
   1       3    50      150   rubber gloves
   2     100     5      500   test tubes
   3       5    80      400   clamps
   4      23    19      437   plates
   5      99    24     2376   cleaning cloth
   6      89   147    13083   bunsen burners
   7       5   175      875   scales
```

Note that column **New** has been added to the table.

See Also

jointable, number

adjust — /act/gl Command

Create adjusted trial balance table
adjust

The command **adjust** is part of the accounting/general ledger system that is included with **/rdb**. It produces the adjusted trial balance sheet, called **adjusttrial**, by adding the trial-balance table **trialbalance** with the journal-of-adjustments table **journaladjust**.

The accountant makes adjustments at the end of the accounting period, to reflect changes in inventory, depreciation, depletion, amortization, and so on. These are computed costs of doing business that are assigned to each accounting period.

Example

The following gives an example of table **trialbalance**:

```
Account Debit   Credit  Format  Taxline Short   Name
------- -----   ------  ------  ------- ------- -----------------
1010    11178           B-A     4L01    cash      Cash
3010             13500  I-I     101a    sales     Sales
3080               250  I-I     106     royalty   Gross royalties
3100              3286  I-I     108     income    Other Income
4020     3601           I-COGS  102-A2  merch     Misc Merchandise
4030     2100           I-COGS  102-A3  wages     Salary and Wages
4370        2           I-E     122     parking   Business parking
4380       41           I-E     122     travel    Business travel
4384      114           I-E     122     hotel     Business hotel
```

The following gives an example of table **journaladjust**:

```
Account Debit   Credit  Format  Taxline Name
------- ------- ------- ------- ------- --------------------
1110              20000 B-A     4L06      Beginning Inventory
1110    22130           B-A     4L06      Ending Inventory
1310                0 B-A     4L09a     Accum Depreciation
1330                0 B-A     4L10a     Accum Depletion
1420                0 B-A     4L12a     Accum Amortization
4010      20000         I-COGS  102-A1    Beginning Inventory
4080            22130   I-COGS  102-A4    Ending Inventory
4170          0         I-E     121       Depreciation
4180          0         I-E     118       Depletion
4215          0         I-E     122       Amortization
```

The command **adjust** then produces the table **adjustedtrial** by combining **trialbalance** with **journaladjust**. In this example, **adjustedtrial** appears as follows:

```
Account Debit   Credit  Format  Taxline Short   Name
------- -----   ------  ------  ------- ------- -----------------
1010    11178           B-A     4L01    cash      Cash
1110     2130           B-A     4L06    inv       Inventory
1310        0           B-A     4L09a   adeprec   Accum Depreciation
1330        0           B-A     4L10a   adeplet   Accum Depletion
1420        0           B-A     4L12a   aamort    Accum Amortization
3010            13500   I-I     101a    sales     Sales
3080              250   I-I     106     royalty   Gross royalties
3100             3286   I-I     108     income    Other Income
4010    20000           I-COGS  102-A1  beginv    Begin Inventory
4020     3601           I-COGS  102-A2  merch     Misc Merchandise
4030     2100           I-COGS  102-A3  wages     Salary and Wages
4080            22130   I-COGS  102-A4  endinv    Ending Inventory
4170        0           I-E     121     deprec    Depreciation
4180        0           I-E     118     deplet    Depletion
4215        0           I-E     122     amor      Amortization
4370        2           I-E     122     parking   Business parking
4380       41           I-E     122     travel    Business travel
4384      114           I-E     122     hotel     Business hotel
```

See Also

act, balance

append — /rdb Command

Add a row to a table and update index tables
append [-h -m[bhirs]] *table* **[***keycolumn ...***] <** *tableorrow*

The command **append** appends the row in table *tableorrow* onto the end of *table*. It also updates the appropriate index table so that the next search will find it. For the *record* method of indexing, **append** adds the new record's offset to the end of the index table. For the *inverted* method of indexing, it adds the record and then sorts the table.

Options

append recognizes the following options:

-h *tableorrow* has no **/rdb** head line and dash line. This option is useful when you only have a row to append to a table, but it does not have a head line.

-m[bhirs]
Use a fast-access method. See the manual page for **search** for a description of the fast-access methods and how to invoke them.

LEXICON

Example

Consider the table **inventory**, which is defined as follows:

```
Item     Amount     Cost     Value     Description
----     ------     ----     -----     -----------
   1          3       50       150     rubber gloves
   2        100        5       500     test tubes
   3          5       80       400     clamps
   4         23       19       437     plates
   5         99       24      2376     cleaning cloth
   6         89      147     13083     bunsen burners
   7          5      175       875     scales
```

This table has an index table, called **record**, which appears as follows:

```
  Offset
--------
      76
     113
     147
     177
     207
     245
     283
```

Finally, consider the table **newrecord**, which is as follows:

```
Item     Amount     Cost     Value     Description
----     ------     ----     -----     -----------
   8         35      105         0     pipettes
```

Note that **newrecord** has the same columnar layout as **inventory**.

The following command appends **newrecord** onto the end of **inventory:**

```
append -mr inventory < newrecord
```

After **append** has done its work, **inventory** appears as follows:

```
Item     Amount     Cost     Value     Description
----     ------     ----     -----     --------------
   1          3       50       150     rubber gloves
   2        100        5       500     test tubes
   3          5       80       400     clamps
   4         23       19       437     plates
   5         99       24      2376     cleaning cloth
   6         89      147     13083     bunsen burners
   7          5      175       875     scales
   8         35      105         0     pipettes
```

inventory's index table now appears as follows:

```
  Offset
--------
      76
     113
     147
     177
     207
     245
     283
     313
```

The new "313" entry in the secondary index table **inventory.r** is the offset to record 8 — that is, the number of bytes for the beginning of the file to record 8.

See Also

delete, index, replace, search

ascii — /rdb Command

Return the ASCII value of a character
ascii *character...*

The command **ascii** converts each of the characters in its first argument to a space-delimited series of numbers. You can use this command to find the internal computer representation of a character. Any strings following the first are ignored. If you have spaces in your string, be sure to put quotation marks around them.

Example

The command

```
ascii aA1
```

produces:

```
97 65 49
```

The command

```
ascii ' '
```

produces:

```
32
```

Note that the single space character, enclosed within apostrophes, is converted to its ASCII code in decimal (32).

See Also

chr
COHERENT Lexicon: **ASCII**

LEXICON

backup — Technical Information

When you are using **/rdb** to store and manipulate your data, you must back up your data regularly. If you do not back up your data and your disk should fail for whatever reason, your data will be irretrievably lost.

See the COHERENT manual section 2, *Using the COHERENT System*, for a detailed description of how to back up your data. Also see the Lexicon articles on **ustar** and **cpio** for descriptions of how to use these tools to back up your data.

balance — /act/gl Command

Create balance sheet from adjusted trial balance
balance

The command **balance** is part of the accounting/general-ledger system included with **/rdb**. It reads table **adjustedtrial** and writes a balance sheet into table and writes it into file **balancesheet**.

Example

Consider the table **adjustedtrial**, as follows:

```
Account Debit   Credit  Format  Taxline Short   Name
------- -----   ------  ------  ------- -------  ------------------
1010    11178           B-A     4L01    cash     Cash
1110     2130           B-A     4L06    inv      Inventory
1310        0           B-A     4L09a   adeprec  Accum Depreciation
1330        0           B-A     4L10a   adeplet  Accum Depletion
1420        0           B-A     4L12a   aamort   Accum Amortization
3010             13500  I-I     101a    sales    Sales
3080               250  I-I     106     royalty  Gross royalties
3100              3286  I-I     108     income   Other Income
4010    20000           I-COGS  102-A1  beginv   Begin Inventory
4020     3601           I-COGS  102-A2  merch    Misc Merchandise
4030     2100           I-COGS  102-A3  wages    Salary and Wages
4080             22130  I-COGS  102-A4  endinv   Ending Inventory
4170        0           I-E     121     deprec   Depreciation
4180        0           I-E     118     deplet   Depletion
4215        0           I-E     122     amor     Amortization
4370        2           I-E     122     parking  Business parking
4380       41           I-E     122     travel   Business travel
4384      114           I-E     122     hotel    Business hotel
```

The command **balance** writes the balance sheet into **balancesheet**, which appears as follows:

```
      Makeapile, Inc.
     31 December 1985
       Balancesheet
```

```
11178     Cash
 2130     Inventory
    0       Accumulated Depreciation
    0       Accumulated Depletion
    0       Accumulated Amortization
-----    ------------------------
13308    Total Assets

 13308   Retained Earnings (Profit)
------   -------------------------
 13308   Total Liabilities and Equity
```

See Also

act, adjust

blank — /rdb Command

Replace all data in a record with spaces
blank < *tableorlist*

The command **blank** replaces with spaces every values in each record of *tableorlist*.

The command **update** uses **blank** when it updates a record. **update** temporarily replaces the record it is updating with a blank record. This blank record serves as a record lock; this ensures that other users cannot read or update the record while it is being updated, but it leaves the rest of the table available for use.

Please note that when you pull up a blank record, it means that that record is being updated by someone else. Come back to it later.

Example

To blank out the data fields in table **maillist** and then **see** the result, you could type the following:

```
blank < maillist | see
```

This command produces the following output:

```
Number^I $
Name^I               $
Company^I          $
Street^I              $
City^I       $
State^I    $
ZIP^I     $
Phone^I              $
$
Number^I $
Name^I               $
Company^I             $
Street^I                              $
City^I       $
State^I            $
ZIP^I         $
Phone^I    $
```

Note the spaces between the tab (^I) and the dollar sign '$' that indicates the end of the line. The spaces replaced the characters, including spaces, that were in the original records.

Note too that the column names are untouched, so that this is a perfectly correct file as far as the **/rdb** commands are concerned, it just contains no data.

LEXICON

See Also

see, update

bom — /act/opr Command

Produce bill-of-materials from parts list
bom

The command **bom** is part of the accounting/operations system included with **/rdb**. It creates a bill-of-materials table from table **part**, which holds all of the items that are required to make a product, and the table **saleitem**, which holds all sales items from the sales department. This tells how many items must be manufactured or purchased to meet current sales orders.

bom writes its output into table **bom**.

Example

Assume the table **saleitem** contains the following data:

```
Order Number Code Backord Qty Price    Total Name
----- ------ ---- ------- --- -----  ------- ----
   1      1     rdb      10   1  1500  1500.00 /rdb
   2      1     rdb      10   2  1500  3000.00 /rdb
   3      1     rdb      10   5  1500  7500.00 /rdb
   3      2     act      10  10  1500 15000.00 /act
   4      1     rdb      10   5  1500  7500.00 /rdb
   4      2     rdb      10  19  1500 28500.00 /rdb
   5      1     rdb      10   5  1500  7500.00 /rdb
   5      2     act      10   9  1500 13500.00 /act
```

Assume, too, that the table **part** contains the following data:

```
Code    Subpart Count
------- ------- -------
act     binder  2
act     actdoc  2
act     f1      10
rdb     binder  2
rdb     rdbdoc  2
rdb     f1      10
```

The command **bom** reads these tables to produce the following summary table:

```
Code    Qty
------- -------
actdoc       38
binder      112
f1          560
rdbdoc       74
```

You can see that we have orders for 19 copies of **act**s in the **salesitem** table. Each package has two documents, for a total of 38 **actdoc**s. Note that table **bom** lists 560 floppies ((19 act x 10) + (37 rdb x 10)). We can order these, regardless of which software packages are being put onto them.

See Also

act

calcpay — /act/pay Command

Post payroll to ledgerpay
calcpay

The command **calcpay** is part of the accounting/payroll system that is included with **/rdb**. It posts the payroll-journal table **journalpay** to the payroll-ledger table **ledgerpay**. This ledger groups the pay checks by employee, and is used to create the W2 form.

Example

Assume that table **journalpay** has the following data:

```
Date    Number  Hours   Salary  Rate    Gross   Federal State   Net
----   ------- ------   ------  ----   ------   ------- -----   ---
860518       1     80     1000     0     1000        70    10   920
860518       2     80        0    10      800        42     6   752
860518       3     30        0    10      300         0     0   300
```

Typing **calcpay** copies it to table **calcpay**.

See Also

act, getjournal

calculate — /act/gl Command

Compute each tax form listed
calculate *taxform ...*

The command **calculate** is part of the accounting/general-ledger system that is included with **/rdb**. It is a simple shell script that calculates each tax form listed on its command line.

Example

The following command calculates the IRS 1040 form

```
calculate 1040
```

See Also

act, fillform

Notes

Please note that the tax tables included with **/rdb** are from 1987. They are included solely to serve as examples, and no claim is made as to their accuracy when calculating current taxes. *Caveat utilitor.*

LEXICON

cap — /rdb Command

Convert first letter of each word to upper case
cap < *textfile*

The command **cap** capitalizes the first letter of each word in *textfile*. It writes its results to the standard output.

Example

Assume that table **Inventory** has the following data:

```
Item  Amount  Cost  Value  Description
----  ------  ----  -----  --------------
   1       3    50    150  rubber gloves
   2     100     5    500  test tubes
   3       5    80    400  clamps
   4      23    19    437  plates
   5      99    24   2376  cleaning cloth
   6      89   147  13083  bunsen burners
   7       5   175    875  scales
```

As you can see, the column **Descriptions** has consists of words all in lower case. To capitalize the first letter of each word in **Inventory** file, type:

```
cap < Inventory
```

This writes the following on the screen:

```
Item  Amount     Cost  Value Description
----  ------     ----  ----- ----------------
   1       3       50    150 Rubber Gloves
   2     100        5    500 Test Tubes
   3       5       80    400 Clamps
   4      23       19    437 Plates
   5      99       24   2376 Cleaning Cloth
   6      89      147  13083 Bunsen Burners
   7       5      175    875 Scales
```

See Also

lowercase, uppercase

cashflow — /rdb Command

Compute balance column of cash table
cashflow < *cashtable*

The command **cashflow** helps you manage your cash flow. It reads *cashtable*, which you design, then computes a running balance in a *Balance* column (which must be column 3) from an *Amount* column (which must be column 2).

Cash-flow analysis lets you see if you will always have enough money in hand to cover each day's bills. It makes cash management possible by telling you whether to put off an expense until a check arrives, ask for an advance, get to work and make more money, or control your spending.

Example

The following gives an example *cashtable*, called **cash**:

```
Date     Amount   Balance   Description
------   ------   -------   -----------
890101                512   current balance
890101      -450           rent check
890101      1000           pay check
890115      -300           estimated tax payment
890115      1000           pay check
890115     -1000           living expenses
890120      -900           big purchase

890201      1000           pay check
890201      -450           rent check
890215      1000           pay check
890115     -1000           living expenses
```

The command **cashflow** needs two columns, named **Amount** and **Balance.** (If you use different column names, you must edit the **cashflow** shell script). All other columns are optional. Each row in your table represents a projected income item (positive) or a projected expense (negative). This is your flow of cash. To summarize the cash flow in our example table **cash**, type:

```
cashflow < cash
```

This prints the following on your screen:

```
Date     Amount   Balance   Description
------   ------   -------   -----------
890101      512       512   current balance
890101     -450        62   rent check
890101     1000      1062   pay check
890115     -300       762   estimated tax payment
890115     1000      1762   pay check
890115    -1000       762   living expenses
890120     -900      -138   big purchase

890201     1000       862   pay check
890201     -450       412   rent check
890215     1000      1412   pay check
890115    -1000       412   living expenses
```

That **big purchase** gives us a negative balance in 890120. Now that you know, you can do something about it. One move is to put the big purchase off until the first of February. After you edit table **cash** to reflect this change, it appears as follows:

```
Date     Amount   Balance   Description
------   ------   -------   -----------
890101                512   current balance
890101      -450           rent check
890101      1000           pay check
890115      -300           estimated tax payment
890115      1000           pay check
890115     -1000           living expenses
890201      1000           pay check
890201      -900           big purchase
890201      -450           rent check
890215      1000           pay check
890115     -1000           living expenses
```

Running **cashflow** again will tell us if this strategy works:

LEXICON

```
Date     Amount   Balance   Description
------   ------   -------   -----------
890101      512       512   current balance
890101     -450        62   rent check
890101     1000      1062   pay check
890115     -300       762   estimated tax payment
890115     1000      1762   pay check
890115    -1000       762   living expenses
890201     1000      1762   pay check
890201     -900       862   big purchase
890201     -450       412   rent check
890215     1000      1412   pay check
890115    -1000       412   living expenses
```

As you can see, this works — there's no negative cash flow for any period. You can also try other options.

You can do all of the inputting and editing in the text editor. You can execute commands in **vi** by using the exclamation '!' shell feature; or from MicroEMACS by typing **<ctrl-X>!**. You also might want to write out to a *tmp* file and use **mv** to overwrite your **cash** table so that it is file is up to date. For example, in **vi** you can type:

```
:!cashflow < cashtable > tmp ; mv tmp cashtable
:e!
```

The first line computes the **Balance** column, writes the result into a **tmp** file, and moves **tmp** to be the new **cashtable.** All of this is necessary because in COHERENT you cannot have one file as both input and output without wiping out the file. The second line, **:e!** pulls the new file into the **vi** editor and displays it on your screen.

See Also

COHERENT Lexicon: **ksh, me, sh, vi**

chartdup — /act/gl Command

Check for duplicate names and accounts in chart
chartdup

The command **chartdup** is part of the accounting/general-ledger system that is included with **/rdb**. It checks to see if there are any duplicates among the account numbers or the short names within the chart of accounts. This is the most common error that people make when they update the chart of accounts.

Example

The following gives example output of **chartdup**:

```
Duplicate Account numbers in chart.  See chart.Account
4150    I-E     115     taxes     Taxes
4150    I-E     115     taxes     Taxes
Duplicate Short names in chart.  See chart.Short
4150    I-E     115     taxes     Taxes
4150    I-E     115     taxes     Taxes
```

check.rdb — /rdb Command

Report any rows in which columns do not match head line
check.rdb < *tableorlist*

The command **check.rdb** counts the columns (number of tabs plus one) in the head line of an **/rdb** table or list formatted file. Then it displays information on each row that does not match the number of columns in the first head-line row. You should use it to see if your data entry is correct. If your head-line row is incorrect, then all of your rows will be reported as errors.

check.rdb returns zero when all is correct, and nonzero when it has found an error. You can use this status value (**$?**) in a shell script to take different actions, depending upon the validity of the table or list file.

Examples

Here is a bad table, called **badtable**:

```
Date    Account  Debit   Credit  Description
------  -------  ------  ------  -----------
820102  101      25000           cash from loan
820102  211.1             25000  loan #378-14 Bank Amerigold
820103  150.1             10000  test equipment from Zarkoff
820103  101       5000           cash payment
820103  211.2     5000           note payable to Zarkoff
820104  130      30000           inventory - parts from CCPSC
820104  201.1             15000  accounts payable to CCPSC
820104  101      15000           cash payment to CCPSC for parts
```

It is hard to see anything wrong with it. However, when we run the command

```
check.rdb < badtable
```

we see the following output:

```
check.rdb: Columns (4) do not equal headline columns (5) in row 3 (line 5).
820103 150.1   10000 test equipment from Zarkoff
```

The command

```
see < badtable
```

gives us the following output:

```
Date^IAccount^IDebit^ICredit^IDescription$
----^I ------^I-----^I------^I-----------$
820102^I101^I25000^I^Icash from loan$
820102^I211.1^I^I25000^Iloan number #378-14 Bank Amerigold$
820103^I150.1  10000^I^Itest equipment from Zarkoff$
820103^I101^I^I5000^Icash payment$
820103^I211.2^I^I5000^Inote payable to Zarkoff Equipment$
820104^I130^I30000^I^Iinventory - parts from CCPSC$
820104^I201.1^I^I15000^Iaccounts payable to CCPSC^I$
820104^I101^I15000^I^Icash payment to CCPSC for parts$
```

Note that a tab is missing from row 3 (line 5) of **badtable**, between values 150.1 and 10000. The spaces hide the fact that it is missing.

When you work with a large file, run **check.rdb** first. It will find the tab problems and give you the information you need to find them. Then use a text editor to correct the problem.

You can move down to the first bad line by typing the following **vi** command **:5**. This takes you to line 5. Then to see the line, type the list command **l:**. (This is the letter **el**, not the number one). Or

LEXICON

to do it in one step (move and list), type **:51**. Or you can using the following colon command to set all of the lines in the file to display tabs:

```
:set list
```

To return it to normal display, use the command:

```
:set nolist
```

Edit in whatever changes you need, then go to the next bad line. You can change a space to a tab with the following command:

```
:s/ /^I/
```

where the **^I** is the key on your keyboard that produces a tab. If you want to list three lines above and below the bad line, type:

```
:-3,+3l
```

If you use **ve** to enter your data, you will have fewer problems like this. Data from foreign sources should be checked carefully before you use it with **/rdb**.

See Also

see, ve
COHERENT Lexicon: **elvis, me, vi**

Notes

This command is named **check.rdb** rather than **check**, as in other implementations of **/rdb**, to avoid clashing with the COHERENT command **check**, which does something very different.

chr — /rdb Command

Display the character corresponding to a number
chr *number ...*

chr converts each of the integers on its command line to its corresponding ASCII character. You can use it to send special characters to the screen. This gives you a good way to produce special characters that are hard to type or are interpreted by the shell.

Example

Here we convert several characters.

```
chr 97 65 49
aA1

chr 7
[beep]

echo ‘chr 7’Wake up
[beep]Wake up
```

The **[beep]** is the sound. It does not print on your terminal.

See Also

ascii
COHERENT Lexicon: **ASCII**

clear.rdb — /rdb Command

Clear the terminal's screen
clear.rdb

The command **clear.rdb** clears your terminal screen of all characters and leaves the cursor in the upper left corner. It is useful for menus and forms that look better on a clear screen.

clear.rdb uses the **termcap** file of terminal capabilities to find the string of special characters that are needed to clear your screen. To work correctly, the **TERM** environmental variable must be set correctly to the name of your terminal as listed in **/etc/termcap**.

The COHERENT command **clear.rdb** also clears the screen.

The **/rdb** command **termput** can also clear the screen, as follows:

```
termput cl
```

Speed

clear.rdb is quite fast, but there is a much faster way. Set a shell variable **CLEAR** to the output of the command **clear.rdb** like this:

```
export CLEAR=‘clear‘
```

Then you can use:

```
echo "$CLEAR"
```

in your shell programs for a fast clear. Put it into your **.profile** so that it will always be available. Use quotation marks around the shell variables in case they have special characters in them.

See Also

termput
COHERENT Lexicon: **clear, export, .profile, sh, TERM, termcap**

Notes

This command is named **clear.rdb** rather than **clear**, as in other implementations of **/rdb**, to avoid clashing with the COHERENT command **clear**.

close — /act/gl Command

Close accounting period creating journal for next
close

The command **close** is part of the accounting/general-ledger system that is included with **/rdb**. It closes an accounting period, such as a month, quarter, or year, and creates the next period's journal, the table **journalnext**. When you start the next period, carry the **journalnext** forward by **mv**ing it to file **journallast**.

Example

The command **close** creates a file whose output resembles the following:

```
Account Date    Debit   Credit  Ref     Description
------- ------- -----   ------  ------- --------------
1010    850101  11178           jl      brought forward
1110    850101  22130           jl      brought forward
1310    850101      0           jl      brought forward
1330    850101      0           jl      brought forward
1420    850101      0           jl      brought forward
4997    850101           33308  jl      brought forward
```

LEXICON

See Also

act

column — /rdb Command

display columns of a table in any order
column [*Column ...* **] <** *tableorlist*

The command **column** reads *tableorlist* and writes a new table (or list) that consists of each *Column*, in the order you list them on the command line.

If you give a column name that is not one of the columns for *tableorlist*, **column** creates a new column with that name, in that location, and leaves it empty. This lets you to create new columns. If you want to compute a column that does not exist, use this **column** facility to create it, then use the **compute** command to compute it. However, if you misspell a column name, you will get an empty column.

If you name no *Column* on the command line, **column** writes all of *tableorlist* to the standard output without change.

Example

Assume that you have an inventory table, as follows:

```
Item   Amount    Cost   Value   Description
----   ------   -----   -----   -----------
   1        3    5.00       0   rubber gloves
   2      100    0.50       0   test tubes
   3        5    8.00       0   clamps
   4       23    1.98       0   plates
   5       99    2.45       0   cleaning cloth
   6       89   14.75       0   bunsen burners
   7        5     175       0   scales
```

The command:

```
column Cost Amount Description < inventory
```

produces the following output:

```
 Cost   Amount   Description
-----   ------   -----------
 5.00        3   rubber gloves
 0.50      100   test tubes
 8.00        5   clamps
 1.98       23   plates
 2.45       99   cleaning cloth
14.75       89   bunsen burners
  175        5   scales
```

Note that the columns **Cost** and **Amount** have been reversed. We only have the columns we asked for and in the order we requested.

column can also manipulate list-formatted files. For example, the mailing list called *maillist* looks like this:

```
Number   1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

If you want only the names and numbers, type:

```
column Name Phone < maillist
```

This produces the following output:

```
Name    Ronald McDonald
Phone   (111) 222-3333

Name    Chiquita  Banana
Phone   1234
```

See Also

project, row
COHERENT Lexicon: **awk**

commands — /rdb Command

Describe /rdb commands
commands

The command **commands** displays a table of **/rdb** commands, with a one-line description and the syntax of each command.

See Also

act, helpme, rdb, whatis, whatwill

compresss — /rdb Command

Squeeze out all leading and trailing blanks
compresss [-b] < *tableorlist*

The command **compresss** removes all leading and trailing blanks from every column or field in *tableorlist*. You an use it to save disk space.

Please note that the name **compresss** is not a typographical error. The command is spelled in this way to avoid collision with the COHERENT command **compress**, which behaves very differently from **compresss**.

compresss is almost the opposite of the command **justify**. Your files are not as pretty when compressed as they are when justified, but they are smaller — usually one third the size or less. Therefore, they take less disk space and are much faster to process; after all, most of the time used to execute data-base programs consists of moving data from the disk to the memory and,

sometimes, back to the disk.

Multiple blanks that separate words are left alone.

Options

compresss recognizes the following option:

-b Drop blank lines. Don't use this option with list files!

Example

For example, consider table **oldjournal**, which is as follows:

```
Date     Account   Debit   Credit   Description
------   -------   -----   ------   -----------
890102   101       25000            cash from loan
890102   211.1              25000   loan number #378-14 Bank Amerigold
890103   150.1     10000            test equipment from Zarkoff
890103   101        5000            cash payment
890103   211.2               5000   note payable to Zarkoff Equipment
890104   130       30000            inventory - parts from CCPSC
890104   201.1              15000   accounts payable to CCPSC
890104   101       15000            cash payment to CCPSC for parts
```

After we run the command

```
compresss <old.journal | see
```

old.journal has no extra spaces. We prove this by piping the output to the **/rdb** command **see**, which displays the following output:

```
Date^IAccount^IDebit^ICredit^IDescription$
----^I-------^I-----^I------^I-----------$
890102^I101^I25000^I^Icash from loan$
890102^I211.1^I^I25000^Iloan number #378-14 Bank Amerigold$
890103^I150.1^I10000^I^Itest equipment from Zarkoff$
890103^I101^I^I5000^Icash payment$
890103^I211.2^I^I5000^Inote payable to Zarkoff Equipment$
890104^I130^I30000^I^Iinventory - parts from CCPSC$
890104^I201.1^I^I15000^Iaccounts payable to CCPSC$
890104^I101^I^I15000^Icash payment to CCPSC for parts$
```

Note that **see** turns all of the tabs into **^I** and puts a dollar sign '$' at the end of each row to show us that there are no spaces at the end.

See Also

justify, rmblank

Notes

Do not confuse this command with the COHERENT system's command **compress**, which produces binary output that cannot be read by any **/rdb** command.

compute — /rdb Command

Calculate columns of a table
compute '*column* **=** *expression* **[; ...]' <** *tableorlist*

The command **compute** lets you to do arithmetic on columns. A column can be computed as a function of other columns, of itself, and of constants.

compute uses the COHERENT command **awk** to perform its work, as does the command **row**. **awk** is a a powerful, interpreted programming language in which you can write programs to perform

complex transformations on text. The advantage of **compute,** over **awk** is that **compute** knows about the names of columns in a table; this permits you to use column names instead of column positional numbers. (In **awk** the second column is named **$2**.) To grasp the full power of **compute** and **row**, read the COHERENT system's **awk** tutorial. If you must perform very complex manipulations of tables, use **awk**.

If you must create a new empty column for **compute** to put values into, use the command **column**. This command creates an empty column for any column it cannot find in the header line.

Example

This example computes a column from other columns:

```
compute 'Value = sprintf("%5.2f", Cost * Amount)' < inventory
```

```
Item   Amount    Cost    Value   Description
----   ------   -----   ------   --------------
   1        3    5.00    15.00   rubber gloves
   2      100    0.50    50.00   test tubes
   3        5    8.00    40.00   clamps
   4       23    1.98    45.54   plates
   5       99    2.45   242.55   cleaning cloth
   6       89   14.75  1312.75   bunsen burners
   7        5  175.00   875.00   scales
```

Here are more examples of what you can do with compute. You can change the format of **column** by using a C-style **sprintf**:

```
compute 'column = sprintf ("%8.2f", column)' < tableorlist
```

You can set **column** equal to the length of the string in **column2**:

```
compute 'column = length (column2)' < tableorlist
```

You can also Set **column** equal to the natural logarithm of **column2**:

```
compute 'column = sprintf("%f", log (column2))' < tableorlist
```

Or, you can increment **column2** whenever **column** is greater than **number**:

```
compute 'if (column > number) column2 = \
     sprintf("%d", column2+1)' < tableorlist
```

Expressions

As you can see, you can write rather complex expressions for **compute**. By *expression*, we mean many kinds of equations. For example, a variable or a column is an expression. So is variable + variable. So are rather complex equations. The following symbols indicate some of the more complex expressions:

column
Name of a column, exactly as it appears at the top of the column.

= Make the column on the left equal the expression on the right.

+ Add

- Subtract

* Multiply

/ Divide.

LEXICON

% Modulo (zero if left value is equally divisible by right).

; Statement separator needed if more than one equation

log()
Natural logarithm of value or column within parentheses.

length()
String length of string in column.

You can also use C language-like statements:

if (*expression***)** *equation* **[else** *equation* **];**
For example:

```
if (Value > 10000) Status = "Special"; else Status = "Normal";
```

while (*expression***)** *equation***;**
For example:

```
while (column => 0) column -= 1 ;
```

for (*expression***;** *condition***;** *expression***)** *equation***;**
For example:

```
for (i = 0; i < column1; i++) array [i] = column2 + i ;
```

You can also use the **awk** format *pattern* { *action* }. For example:

```
compute 'length > 80 { print NR, "Line too long." }' < table
```

awk recognizes *length* as a function that returns the length of the row and **NR** as the line number. Note that the line numbers of a file starts with the head line, while row numbers of a table starts after the dash line (line 3).

What Is a Column Name?

compute looks up, in the column header of the input file, each word that it finds in its command-line program. If there is a match, **compute** converts the column name to its position (first, second, third, etc.) depending upon the relative location of the column. These column numbers are needed by **awk**.

compute defines a column name to be a string of the following characters:

1. Upper and lower letters, numbers, and the underscore '_'.

2. Any string that is enclosed by apostrophes or quotation marks.

Therefore, if you want special characters in your column names, put quotation marks around them when you use them. For example:

```
compute ' "Item#" = NR ' < inventory
```

Note the quotation marks around the column head **Item#** because of the number character '#'. Also, note that we used quotation marks, because the whole program was enclosed in apostrophes. If you use one kind of mark around the whole program, use the other kind of mark around the column names that have special characters in them.

Reserved Words to Avoid in Column Names

Some words are understood by **awk**. Therefore, they should not be used in column names. A simple way to avoid a conflict is to start your column names with a capital letter. **awk** does not confuse **Print** with **print**. Here is a list of **awk**'s built-in functions.

Reserved Words *Description*

BEGIN.....Pattern that matches before first input record
END .....Pattern that matches after last input record
break .....Get out of for or while loop
continue.....Go to next iteration of for or while loop
else .....Used in "if then else" expression
exit .....Leave program entirely as if end of input
exp.....Raise number to a power
for.....for (expression ; condition ; expression) statement
getline .....Get next input line
if .....if (condition) statement [else statement]
in .....for (variable in array) statement
index .....index (string1, string2)
int.....Truncate argument to integer
length.....Return current line length, or length of argument
log.....Return log (to base 2) of argument
next .....Skip to next record and reexecute all commands
print .....Output variables
printf.....printf ("format", variable, ...)
split .....split (string, arrayname, separator)
sprintf .....sprintf ("format", variable, ...)
sqrt.....Return square root of argument
substr .....substr (string, start, number)
while.....while (condition) statement

Using Shell Variables in Programs

Often you want to use a shell variable in a **compute** command. It is easy, if you understand what is going on. For example, the following commands:

```
DATE=860101
compute "Date = $DATE" < journal
```

produce the following output:

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  ---------------------------------
860101  101     25000           cash from loan
860101  211.1            25000  loan number #378-14 Bank Amerigold
860101  150.1   10000           test equipment from Zarkoff
860101  101               5000  cash payment
860101  211.2             5000  note payable to Zarkoff Equipment
860101  130     30000           inventory - parts from CCPSC
860101  201.1            15000  accounts payable to CCPSC
860101  101              15000  cash payment to CCPSC for parts
```

We set a shell variable named **DATE** to a date. Then we used the variable in the **compute** command line program. But note that we had to use quotation marks (") instead of our usual apostrophes ('). To the shell, apostrophes protect absolutely. With apostrophes, the shell would not see the dollar sign '$' in front of **DATE** and would not convert it to its value. The quotation mark protects the enclosed from the shell, also, except for shell variables and command substitution (`cmd`). With quotation marks, the shell will still replace variables (which start with a dollar sign character).

There is one further complication. The previous worked fine because the value of date was a number. But if we want to use strings in **awk**, they must have quotation marks (not apostrophes) around them. For example:

```
COMPANY='Makeapile, Inc.'
compute "Company = \"$COMPANY\"" < maillist
```

LEXICON

produces the following:

```
Number        1
Name          Ronald McDonald
Company       Makeapile, Inc.
Street        123 Mac Attack
City          Memphis
State         TENN
ZIP           30000
Phone         (111) 222-3333

Number        2
Name          Chikeeta Banana
Company       Makeapile, Inc.
Street        Uno Avenito De La Revolution
City          San Jose
State         El Salvadore
ZIP           123456789
Phone         1234
```

Here we set a shell variable **COMPANY** to a company name. We want to update the mail list because these characters have been hired by a new company. Note what we had to do. We needed quotation marks around the whole program to let the shell substitute the value of the shell variable. But we also needed quotation marks around the string "Makeapile, Inc." for **awk**. So we have to protect the innermost quotation marks with backslashes.

There is another way, which opens up a little window in the program for the shell to insert strings. But it fails when there are spaces in the window string. So use the method previously discussed.

Problem With awk Floating-Point Format

With some implementations of **awk**, the value column is computed to much greater precision than you are likely to want. For example, version produces this table:

```
Item#   Amount   Cost    Value  Description
------- ------- ------- ------- ---------------
      1       3    5.00 15      rubber gloves
      2     100    0.50 50      test tubes
      3       5    8.00 40      clamps
      4      23    1.98 45.539997100830078125   plates
      5      99    2.45 242.5499725341796875    cleaning cloth
      6      89   14.75 1312.75 bunsen burners
      7       5  175.00 875     scales
```

To control the precision of the floating-point values in a column, you can use the **sprintf** function from the C programming language that has been implemented in **awk**. For example:

```
compute 'Value = sprintf ("%7.2f", ( Cost * Amount ))' < inventory
```

These commands will produce the first table under Examples, above.

See Also

COHERENT Lexicon: **awk, ksh, sh**

computedate — /rdb Command

Add given number of days to a date
computedate *date days*

The command **computedate** adds days to a date, and displays the new date. *date* must be the format *YYMMDD*; *days* can be negative or positive. **computedate** displays the new date in the same format. This is the best format for entering dates in a column because they sort correctly and take

only six characters.

Internally, **computedate** converts the date to **julian**, adds days, then converts the date back to **gregorian**. Suppose we want to know the date 45 days from March 31, 1989. We can type:

```
computedate 890331 45
```

This command returns the string:

```
890515
```

That is May 15, 1989.

Another trick is to combine the **todaysdate** and the **computedate** programs. **todaysdate** gives the date of today in the preferred format of *YYMMDD*. For example, the following command shows us the date 45 days from today:

```
computedate 'todaysdate' 45
```

Data Validation

You can use **computedate** to validate a date. If you add zero days to a date and get a different date, then the original date was not valid. For example, let's test the 32nd day of January 1989:

```
computedate 890132 0
```

This returns the string **890201** or February 1, 1989. As we expected, we received a different date. We can test this with the following shell program:

```
if test $DATE = 'computedate $DATE 0'
then
     echo OK
else
     echo Invalid DATE
fi
```

See Also

gregorian, julian, todaysdate

consolidate — /act/gl Command

Combine all subsidiary journals to general journal
consolidate *nextdate*

The command **consolidate** is part of the accounting/general-ledger system that is included with **/rdb**. It assembles all of the subsidiary journals, such as **journalincome**, **journalexpense**, and **journaldeposit**, into the general journal named **journal**. It also converts from single-entry to double-entry bookkeeping by totaling **Debits** and **Credits** and inserting counterbalancing adjustments to the cash account. It needs the **nextdate** for these accounts.

Example

The command

```
consolidate $NEXTDATE
```

writes into file **journal** data that resemble the following:

```
Date    Account Debit   Credit  Ref     Description
------  ------  ------- ------- ------  --------------------
890101  4370          2         r       parking at clients
890103  4380         15         c115    shuttle bus to airport
890104  4380         13         v       meal
890105  4380         13         c116    meal
890106  4384        114         r       hotel
890118  3100               3286 c101    consulting
890125  3080                250 c1459   book fees
891231  1010               5858         double-entry
891231  1010      17036                  double-entry
891231  3010              13500 deposit from sales/deposit
891231  4020       3601         check   from pur/check
891231  4030       2100         payroll from pay/journalpay
```

Note the double-entry adjustments, and the totals of journals from sales, purchasing, and pay departments. The **Date** came from the **NEXTDATE** shell variable, which you must set in your **.profile.**

See Also

getjournal

cpdir.rdb — /rdb Command

Copy one directory tree to another directory
cpdir.rdb *fromdirectory todirectory*

The command **cpdir.rdb** copies a directory tree — that is, all directories and their files — from *fromdirectory* to *todirectory*.

cpdir.rdb uses the tar command in a special way. It is a shell program so you can read it and see how it does it. It performs almost exactly the same work as the COHERENT command **cpdir**.

Example

You might use this to back up a large directory:

```
cpdir.rdb /usr/rdb /usr/rdb.backup
```

The following moves a directory to a preferred place:

```
cpdir.rdb /usr/he.left/goodstuff /usr/project/goodstuff
```

See Also

COHERENT Lexicon: **cpdir, tar**

Notes

This command is named **cpdir.rdb** rather than **cpdir**, as in other implementations of **/rdb**, to avoid clashing with the COHERENT command **cpdir**.

cstate — /act/sales

Produce customer statement
cstate

The command **cstate** is part of the accounting/sales system that is included with **/rdb**. It generates the customer statement from the tables **salesorder** and **deposit**. It shows what the customer has ordered and paid, and the net. The result is printed as a form to be sent to the customer as a bill.

Example

The following gives an example of the output of **cstate**:

```
Please enter customer number (or all or Return to exit): 3
                Customer Statement
                ------------------
Makeapile, Inc., 123 Bigbucks Blvd., Dallas, TX 12345, 1-800-SOF-WARE

Mailing Address for Customer Number  3

Ms Ute  Unix Ph.D. President

UniUniUniUni Software Sellers
12345 Nixuni Street
Union, New Jersey  11111  USA

Now Due and Payable is your balance of: 107550.00

Details of Your Orders and Payments

Cust    Date       Amount       Ref
----    ------  ---------       ----
   3    850902   -6300.00         17
   3    850902  -14175.00         16
   3    850902  -22050.00         15
   3    850902  -23625.00         12
   3    850902  -37800.00         13
   3    850903  -12600.00         23
   3    860105    9000.00       3347
----    ------  ---------       ----
                -107550.00

Please enter customer number (or all or Return to exit):
```

See Also

act, vstate

cursor — /rdb Command

Move the cursor to the row and column requested
cursor *row col* **[** *CURSOR* **]**

The command **cursor** move the cursor to screen position *row* and *column*. You can **cursor** and the **/rdb** command **clear.rdb** to perform complex screen handling from within a shell script. You can paint your screen with cursor moves and **echo** commands; then use the **read** command to read the user's input; and finally use various COHERENT and **/rdb** commands to examine and confirm the input. You can also clear fields of the screen by moving the cursor and writing blanks.

You must give the row and column you want. **cursor** also needs the cursor movement (**cm**) entry from file **/etc/termcap**. This can be given as an argument, but the easiest way is to set up the shell variable called **CURSOR** then **cursor** will find it in its environment automatically. To set up the **CURSOR** variable, do the following:

```
CURSOR='termput cm'
```

Another way is to use your text editor to find the entry in **/etc/termcap** and copy its **cm** entry for your terminal to your **.profile**:

```
CURSOR=^[[%p1%d;%p2%dH
```

This is the entry for the **pc** terminal. The uparrow bracket (^[) is the ESC character (**<ctrl-[>**). The function *tgoto* function from the **termcap** library converts it into the proper escape string.

LEXICON

Example

The following moves the cursor to the middle of an 80-by-24 screen:

```
cursor 11 38
```

Speed

cursor is fast, but there is a much faster way. Set a shell variable like **L22** to move to the 22nd line:

```
L22=‘cursor 23 0‘
```

Then you can use:

```
echo "$L22$MESSAGE\c"
```

in your shell programs for a fast cursor movement and to print a message. Note the quotation marks around the messages in case there are special characters. Also note the *\c*, which tells **echo** not to print a carriage return.

An even better trick is to use the shell feature that tests to see if a variable has been set and only gives it a value when it needs one:

```
echo "${L22:=‘cursor 22 0‘}$MESSAGE\c"
```

All of this magic causes the shell to check if **L22** has a value. If not, it executes the **cursor** command and assigns the value to **L22.** Then, if this line is called in a loop, the next time the value will have been set and will be about 20 times faster.

See Also

termput
COHERENT Lexicon: **.profile, termcap**

dash line — /rdb Definition

The *dash line* is the second line in an **/rdb** table. It underscores each entry in the table's head line. Each entry in the dash line must be separated by a tab character.

Example

The table **inventory** appears as follows:

```
Item  Amount  Cost    Value   Description     New
----  ------  ----    -----   --------------  -------
   1       3    50      150   rubber gloves
   2     100     5      500   test tubes
   3       5    80      400   clamps
   4      23    19      437   plates
   5      99    24     2376   cleaning cloth
   6      89   147    13083   bunsen burners
   7       5   175      875   scales
```

The dash line is:

```
----  ------  ----    -----   --------------  -------
```

Note that there is one entry in the dash line for each entry in the head line.

See Also

head line, listtotable, tabletolist

datatype — /rdb Command

Display the data type of each column selected
datatype [-l] [*column ...* **]** **<** *table*

The command **datatype** displays the type of data contained in each *column*. If you do not name a *column* on the command line, **datatype** calculates and displays all of *table*'s columns.

Options

The **-l** option tells **datatype** to print the entire contents of *table*, in addition to information about each *column*.

Code

-1 Character strings with nonnumeric data.

0 Integer number.

1,2... Floating-point number (number of digits to the right of the decimal).

LEXICON

Example

The following shows a sample output of **datatype**:

```
Int     Char    Float2  Float4
---     ----    ------  ------
1       a       1.2     1.1234
745     zzzz    8.54    12.3
---     ----    ------  ------
  0       -1         2       4
```

Note that the *Int* column has only integers in it (code 0), the *Char* column has characters in it (code -1), and the two *Float* columns have different precisions (positive numbers codes). Note that the float precision is the maximum.

See Also

ascii, chr, dbdict

dBASE crossreference — Technical Information

dBASE was developed and marketed by Ashton-Tate; this company was purchased recently by Borland International.

The following table gives the **/rdb** and COHERENT equivalents for most common dBASE commands. The slug in **bold** characters gives the dBASE command; the following description gives the equivalent in **/rdb** and COHERENT commands:

ACCEPT Comment TO Variable
echo Comment ; read Variable

APPEND BLANK
echo >> Fileout

APPEND FROM Filename FOR Condition
row 'Condition' < Filename >> Fileout

BROWSE
more, scat, update, or any text editor

CANCEL
DEL (any character set with **stty**)

CHANGE Range FIELD Fieldname FOR Condition
compute 'Range && Condition {Fieldname = Value}' < Filein

CLEAR
CLEAR GETS
clear

CONTINUE
continue [shell statement]

COPY TO Filename FIELD Fieldnames FOR Condition
row 'Condition' < Filein | column Fieldnames > Filename

COUNT FOR Condition TO Variable FOR Condition
row 'Condition' < Filein | tail +3 | wc -l

CREATE Filename
> Filename ; vi Filename ; ve Filename ; cmd > Filename

DELETE RECORD Number
(sed $NUM+1}q Filename ; tail +$NUM+3}) > tmp; mv tmp Fieldname

DELETE NEXT Number
DELETE ALL
sed 2q Filename > tmp ; mv tmp Filename

DELETE NEXT Number FOR Condition
DELETE FILE Fieldname
> Filename

DISPLAY Range FOR Bedingung Field OFF
row 'Range && Dedingung' < Filename | column Fieldnames

DISPLAY STRUCTURE
for I in * do sed 2q; done

DISPLAY MEMORY
df ; du

DISPLAY FILES ON Disk LIKE Datatype
ls *Datatype ; ls Disk/*Datatype

) Program
Program

CASE Condition
Condition) [shell case statement]

OTHERWISE
*) [shell case statement]

DO WHILE Condition CR ENDDO
while Condition CR do done

EDIT
vi, ve, update, ex, ed, me, other editors and commands

ECT
ENDDO
do [in shell while, until, for statements]

ERASE
clear

FIND Text
/Text (in vi, ve, update)

INDEX ON Fieldname TO Filename
index Filename Fieldname ...

USE Filename INDEX Keyfield
search Filename Keyfield

INPUT Text To Variable
Variable=Text

INSERT
sed ${NUM}q Filename; echo $RECORD; tail +$NUM Filename

JOIN TO Filename FOR Condition FIELDS Fieldnames
jointable Filename1 Filename2
row 'Condition' < Filename2 | jointable Filename1 -
column Fieldnames < Filename2 | jointable Filename1 -

LEXICON

Column,Row SAY Comment GET Variable PICTURE
tput Column ; tput Row ; echo Comment ; READ Variable

GO
GOTO
GO RECORD n
ve nG

GO Fieldname
ve /

GO TOP
ve H

GO BOTTOM
ve L

IF Condition Statement2 ELSE Statement2 ENDIF
if Condition then Statement1 else Statement2 fi

LOCATE FOR Condition
row 'Condition' < Filename

LOOP
while, until, for [shell statements]

MODIFY STRUCTURE
column New Fieldnames < Filename

NOTE
REMARK
: Old Comment
New Comment

MOVE Old-Filename TO New-Filename
mv Old-Filename New-Filename

REPLACE Range Fieldname WITH Expression FOR Condition
compute 'Fieldname = Expression' < Filename
compute 'Range && Condition {Fieldname = Expression}' < Filename

REPORT Form FOR Condition TO PRINT
report Form < Filename
row 'Condition' < Filename | report Form | print

PACK
compress, uncompress [COHERENT]

QUIT
<ctrl-D>

READ
read Variable

RECALL Condition
row 'Condition' < Filename

RELEASE Variable
Variable=

RESET

RESTORE FROM Filename
RETURN
return [shell]

SAVE TO Filename
SELECT
row, column, etc.

SET
set [shell]

SKIP
SORT ON Fieldname TO Filename
sorttable Fieldname > Filename

STORE Expression TO Variable
Variable=‘Expression‘

SUM Fieldname TO Variable FOR Condition
Variable=‘row ’Condition’ < Filename | column Fieldname‘

TOTAL ON Keyfield TO Filename FIELDS Fieldnames FOR Condition
row ’Condition’ | subtotal Keyfield Fieldnames > Filename

UPDATE FROM Filename ON Keyfield
update, vi, ve, replace, append, delete, etc.

USE Filename
cmd < Filename

WAIT
wait [shell]

@ Row,Column SAY expression
cursor Row Column; echo EXPRESSION
ve screen and validation file

All data-base systems have some method for importing text files. Sometimes you’ll be able to use the header information that comes with **/rdb** tables, and sometimes you’ll have to specify this information to the target system anyway.

dbdict — /rdb Command

Print a data-base dictionary
dbdict < *table*

The command **dbdict** prints a table of two columns: The first gives the names of each field in *table*, and the second gives the field numbers.

Example

For example, we might have a table named *journal* that looks like this:

LEXICON

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  ---------------------------------
861222  101     25000           cash from loan
861222  211.1            25000  loan number #378-14 Bank Amerigold
861223  150.1   10000           test equipment from Zarkoff
861223  101               5000  cash payment
861223  211.2             5000  note payable to Zarkoff Equipment
861224  130     30000           inventory - parts from CCPSC
861224  201.1            15000  accounts payable to CCPSC
861224  101              15000  cash payment to CCPSC for parts
```

The command

```
dbdict < journal
```

produces the following:

```
field   name
-----   ----
1       Date
2       Account
3       Debit
4       Credit
5       Description
```

delete — /rdb Command

Blank record and update index file
delete [-m[bhirs]] *table keycolumn ...* **<** *keytable*

The command **delete** writes a blank record on the row in *table* whose key matches the row in the *keytable*. It also deletes the offset row in the appropriate fast-access offset table, or, in the case of the **hash** method, replaces the deleted offset with -1.

Options **-m[bhirs]** lets you select a type of fast-accessing to locate the row to delete. For information on the type of fast accessing that each option invokes, see the manual entry for **index**.

Example

Consider table **inventory**, which appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

The **record** index looks like this:

```
  Offset
--------
      76
     113
     147
     177
     207
     245
     283
```

The following command deletes the first record **Item**1:

```
echo 1 | delete -mr inventory Item
```

inventory now appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

inventory.r now appears as follows:

```
  Offset
--------
     113
     147
     177
     207
     245
     283
```

Note that we have blanked out the record and removed the offset.

See Also

append, index, replace

difference — /rdb Command

Output table of rows that are in only one table
difference *table1 table2*

The command **difference** uses the COHERENT commands **sort** and **uniq** to subtract logically *table2* from *table1*. Its output consists of the rows that are in *table1* but not in *table2*. The two input tables must have the same columns (*union compatible* in the technical data-base literature), and the rows must be exactly identical — including every space, tab, and nonprinting character. If two rows look the same to you but not to **difference**, try the **see** command to discover any blank spaces and nonprinting characters.

Example

In this example, consider the table **journal**, which appears as follows:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820107    14.00 meal    v       meal with jones
820113   101.62 car     v       car repairs
820114    81.80 insur   c       car insurance allstate
820114    93.00 car     c       car registration dmv
820119    81.72 vitamin c       sundown vitamins
820121    20.83 meal    v       meal with scott
820121  2500.00 keogh   c       keogh payment
820125    99.00 dues    v       dues to uni-ops unix conference
```

And consider the table **carexpense**, which appears as follows:

LEXICON

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820113   101.62 car     v       car repairs
820114    81.80 insur   c       car insurance allstate
820114    93.00 car     c       car registration dmv
```

The command

```
difference journal carexpense
```

logically "subtracts" **carexpense** from **journal**, to produce the following output:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820107    14.00 meal    v       meal with jones
820119    81.72 vitamin c       sundown vitamins
820121    20.83 meal    v       meal with scott
820121  2500.00 keogh   c       keogh payment
820125    99.00 dues    v       dues to uni-ops unix conference
```

See Also

COHERENT Lexicon: **diff, diff3, sort, uniq**

display — /rdb Command

Write table or list file to standard output
display < *tableorlist*

The command **display** copies *tableorlist* from the standard input to the standard output. This is not a very useful program, but is a template for the C interface programs. If you have its source code, you have an example of how to process tables and lists and can modify it for your own programs.

display is slower than the COHERENT command **cat** because it does more processing. **display** knows about table and list formats, and reads and writes one row at a time.

See Also

COHERENT Lexicon: **cat, more**

domain — /rdb Command

Display invalid values in a column
domain *domaintable* **[***string ...***] [<** *one-column-table***]**

The command **domain** displays all strings or all rows in *one-column-table* that do not match any entry in the *domaintable*.

domain is useful for validating that each item in a column is legitimate. First, build a file that consists of a list of all legal values. In relational theory, these values are called the *domain* of the column. (You can also use the command **validate** to validate numbers and short lists of valid strings.)

To check the domain of a column in a multicolumn table, simply use the command **column** to project the column you want.

The *domaintable* must be sorted because it be searched by the command **search** using the binary fast-access method (**-mb**). **domain** is a shell script, so you can edit it to change its search method,

Example

You might have an order file for cars like this:

```
Qty     Model   Colors  Options
------- ------- ------- -------
1       sedan   black   3
1       sedan   green   1
1       sedan   farble  4
3       sedan   red     5
2       convert white   1
1       sedan   purple  2
1       sedan   yellow  4
3       convert blue    2
```

You would also need a sorted file of possible car colors:

```
Colors
------
black
blue
carmel
green
purple
red
silver
white
```

Now we can validate the **Color** column and see all unacceptable colors:

```
column Colors < orders | domain colors
```

This produces the following output:

```
Colors
------
farble
yellow
```

domain complains about **farble** and **yellow** because neither color is in our domain list.

You can invoke **orders** in the text editor and edit it, or add those colors to the approved list. In **vi** use the **/pattern** command to find all of the *patterns* in the orders file. Another approach is to type something like this:

```
:g/farble/s//purple/g
```

This **vi** command will find all instances of **farble** and change them to **purple**.

You can also validate strings written on the command line. For example, the command

```
domain colors red blue purple farble yellow green
```

also prints the output:

```
farble
yellow
```

These command-line stings should be quoted if they have any special characters in them. They are sought by the command **grep**. If they contain a dollar sign '$', it should be protected with three backslashes: this expression is scanned both by the shell and by **grep**, and it takes three backslashes to get one backslash through to **grep**.

See Also

column, search, validate, ve
COHERENT Lexicon: **egrep, grep, sort, uniq**

LEXICON

Example

enter — /rdb Command

Add rows to a table or list file without an editor
enter [-*limit***]** *tableorlist*

The command **enter** is a simple way to append one or more rows to the end of *tableorlist*. It displays each column name and waits for you to type each item.

enter is a substitute for the form-editor **ve**. It is easier to use, but very limited.

Note that this is one of the few **/rdb** commands that does not use the less-than symbol '<' to input a table, but requires the name of a table to which it is to append data.

If you make an error in entering a record, you can return to the last line by typing a caret '^'; **enter** then prompts you with the previous line. If you return two lines, keep typing carets until you are prompted for the line you wish to reenter.

Options

The option **-***limit* lets you limit the number of rows to be entered. For example, **-1** prompts for one row and then exits. Thus, it can be used in a shell script.

Rules

To set up and enter a file in list format instead of table format, there are several rules and suggestions.

First, let's discuss terminology. A list is like a table that is turned sideways. For example, a wide table looks like this:

```
Number  Name    Company Street  City    State   ZIP
Phone
------  ----    ------- ------  ----    -----   ---
-----
1       Ronald McDonald McDonald's      123 Mac Attack
Memphis TENN    30000   (111) 222-3333
```

Note that the row is so long that it wrapped around to the next line. Also, the column names and column data do not line up. *Names*, *Companies*, and *Street Addresses* are usually too long to fit nicely into such a table. Therefore, we need a list format like this:

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333
```

These two formats (table and list) are made interchangeable by using two programs: **listtotable** and **tabletolist**. You can enter and keep a file in list format. When you want to use several commands, you can convert it to table format and pipe it into the regular commands for faster execution. For example, the command

```
listtotable < maillist | column Name Phone | justify
```

produces the following output:

LEXICON

```
Name            Phone
--------------- --------------
Ronald McDonald (111) 222-3333
```

It is best, therefore, to remember that **Name** and **Phone** are column names. The information that follows them in the list is the first row of a table (that can be created by the command **listtotable**).

1. First use an editor to create the list file. Type each column name followed by one, and only one, tab. Also enter the information for the first row (record).

2. Be sure to put a newline (blank line) as the first line of the file.

3. Be sure to put a blank line at the end of each row (record). In the previous example, it should appear after the row **Phone**. This tells the program that it has reached the end of the row.

You can then save the file and use the **enter** program. Type:

```
enter maillist
```

```
Name    _
```

enter responds with the first column name and a tab, and waits for you to enter whatever you wish for **Name**. The underscore '_' shows where the cursor of your terminal will be waiting for you to input data.

4. Type in your data for the item. Use the return key to end the information for this item. If you type more than 80 characters or so, your terminal will wrap around. You can continue to type and wrap around until you reach your editor's line limit or press **<Return>**. Then **enter** gives you the name of the next column head, a tab, and waits for your entry.

5. When **enter** has given you the first column name of a new record and is waiting for a response, exit from it by typing **<ctrl-D>**. **enter** then returns you to the COHERENT shell.

For list-formatted files, it is better to use the **enter** command than an editor. If you do your entry with a text editor, you must include all of the column names, in the same order and spelled correctly.

Example

To add a line to file **maillist**, type the following:

```
enter maillist
```

The editing session may go as follows:

```
Number  2
Name    Chiquita  Banana
Company Standard Brands
Street  ^
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234

Name    <ctrl-D>
```

Note we made an error entering **Company** and did not realize it until we were at **Street.** We typed a caret '^' that reprompted us for **Company**, which we then reentered. **enter** then went on to prompt us for **Company** again.

The entered file now appears as follows:

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

Note that the last row (actually eight lines and the bottom newline) is the one we entered. Just press **<Return>** if you have no data to enter after a prompt.

Now for an example of **enter**ing a table. If you want to add a line to **inventory**, type:

```
enter inventory

Item#   Amount   Cost    Value  Description
------- ------- ------- ------- ---------------
```

Then start typing your rows. Remember to insert tabs between columns:

```
8       123     5.98    0       widget
9       29      15.50   0       another widget
<ctrl-D>
```

The inventory file now looks like this:

```
Item#   Amount   Cost    Value  Description
------- ------- ------- ------- ---------------
1       3       5.00    0       rubber gloves
2       100     0.50    0       test tubes
3       5       8.00    0       clamps
4       23      1.98    0       plates
5       99      2.45    0       cleaning cloth
6       89      14.75   0       bunsen burners
7       5       175     0       scales
8       123     5.98    0       widget
9       29      15.50   0       another widget
```

Note that the last two lines are the ones we entered.

See Also

listtotable, tabletolist, ve

LEXICON

explode — /rdb Command

Produce table of subparts and their count for a part
explode *Part Amount* **[** *table* **]**

The command **explode** takes a part number or name and looks it up in a part table to find all of its subparts. It then finds the subparts of those subparts, and so forth, until it can seek no farther. **explode** also multiplies the number of parts times the subpart counts at each interaction, so the final subpart table lists all of the subparts of the original part and the total number of subparts needed to make the part.

You can use **explode** to generate a list of all items needed to make a product. The cost of these subparts, times the count, gives the value that can be totaled to arrive at the total cost of the product.

explode assumes that the **Part**, **Subpart**, and **Count** columns are the first three columns of the table, and that the third column is actually named **Count**.

Example

If you had a **parttable,** like this:

```
Part    Subpart Count
----    ------- ------
10001   10010   3
10001   10020   4
10010   10100   2
10020   10100   5
```

and you wanted to find the subpart list and count for two part 10001s, you would type:

```
     explode 10001 2
```

The result would be:

```
Part    Amount
----    ------
10001        2
10010        6
10020        8
10100       52
```

Note that both 10010 and 10020 have 10100 as a subpart, but **explode** uses **subtotal** to combine the 10100 subparts into one row. The table is also sorted.

See Also

bom, subtotal

fd — /rdb Command

Test for functional dependency of columns
fd *determinecolumn dependcolumn* **<** *table*

The command **fd** reads *table* and tests whether its *determinecolumn* determines *dependcolumn*. A column in a table functionally depends upon another column when there is only one value in the second column for each value in the first. This helps you to determine if a table is normalized. (See the section on **Normalization**, below, for details on what this means.)

Example

First let's look at our inventory table:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

Note that the first column, **Item,** is the key column for the table. It functionally determines all of the other columns. Each value in column **Item** is unique, so each value determines, or is the key, to its row and all of the column values on it.

However, column **Cost** does not functionally depend upon **Amount** because there are two '5' values in **Amount** and each has a different value for **Cost** associated with it (on the same row or line). The common sense (technically called the "semantics") of the table is that knowing the number of items we have in inventory does not tell us how much they cost.

Now let's see if **fd** can find the dependencies. The command

```
fd Item Cost < inventory
```

returns the message **true**. This is correct, because **Item** is the key column. However, the command

```
fd Amount Cost < inventory
```

returns the message **false**. This, too, is correct.

Normalization

A key column should aways functionally determine all of the other columns in the table. But nonkey columns should not depend upon each other.

If there are too few data in the table, a pseudo-dependency may be discovered. Be sure your table has lots of data before you test it.

LEXICON

To normalize your tables, you must check for functional dependencies. When functional dependency is found with non-key columns, the columns should be projected and uniqued to form a second table. Then the functionally dependent column, but not the determining column, should be removed from the first table. For example:

```
column column1 column2 < table | uniq > newtable
column 'sed 1q table | sed s/column2//' < table > tmp
mv tmp table
```

In the second table, the column that determines the other column is the key for that table. If you need to recreate the original first table, you can join the two new tables by using the command **jointable**, as follows:

```
jointable -j column1 table newtable > oldtable
```

Normalization, or simplification, is an important process. Normalized files are smaller and are easier to maintain.

See Also

column, jointable, paste.rdb
COHERENT Lexicon: **cut, false, mv, paste, sed, true**

filesize — /rdb Command

Return the number of characters in a file
filesize *file ...*

The command **filesize** displays the number of bytes in each *file*. It is much faster than the COHERENT command **wc** because it uses the system call **stat()** to get the number from the i-node table, rather than counting the bytes in the file.

If you ask **filesize** to size only one file, it sends one number to the standard output. This number can be assigned to a shell variable. With more than one *file*, it produces a table of sizes.

Example

The following command returns the size of file **filesize.1**, which holds the "raw" form of what you are reading:

```
filesize filesize.1
```

This displays one value on the standard output, e.g.:

```
532
```

The following command asks for the size of files **ascii.1** and **filesize.1**:

```
filesize ascii.1 filesize.1
```

This form of the command returns a tabular output, e.g.:

```
  Offset        Filename
--------        --------
     464        ascii.1
     532        filesize.1
```

See Also

COHERENT Lexicon: **ls, stat(), wc**

fillform — /act/gl Command

Fill a tax from with adjusted trial balance data
fillform *form*

The command **fillform** is part of the accounting/general-ledger system included with **/rdb**. It fills in the tax form *form*, which must be one of **1120** (corporate), **1120S** (S corporation), **C** (schedule C), or partnership tax form. It writes its output into a file with the name *form*.

Example

The following example fills in the S corporation form:

```
fillform 1120S
```

When you post-process the output with the command

```
sed 10q 1120S
```

you see output that resembles the following:

```
Taxline Amount  Description
------- ------  -------------------------------------
101a     13500  Sales
101b         0  Returns and Allowances
101c     13500  Net Sales
102          0  Cost of Goods Sold
102-A1   20000  Beginning Inventory
102-A2    3601  Misc Merchandise
102-A3    2100  Salaries and Wages
102-A4       0  Other Costs
```

See Also

calculate
COHERENT Lexicon: **sed**

Notes

Please note the tax tables and forms included with **/rdb** are from 1987. These are examples, and no claim is made for their accuracy in computing this year's taxes. *Caveat utilitor*.

fixtotable — /rdb Command

Converts fixed length format to /rdb table format
fixtotable *column***[=** *n1*,*n2***] ... <** *fixtdb*

The command **fixtotable** converts a data base from fixed-length format to variable length **/rdb** format. It does so by selecting *n2* bytes of data from table *fixtdb*, beginning at the *n1* position for each *column* specified. It inserts tabs between columns and calls the **/rdb** command **compress** to remove leading and trailing blanks. It generates header records from the column names. Column names that do not specify start position (*n1*) and field width (*n2*) are considered empty columns, and appear as such in the output. Blank records are removed.

Example

For example, let's look at a file named **journal**:

```
1       10        20        30
----+----|----+----|----+----|
820102101     25000
820102211.1          25000
820103150.1   10000
820103101              5000
820103211.2            5000
820104130     30000
```

The first six bytes are the date field; the next six bytes, starting at position 7 are the account number field; the next seven bytes, starting at position 13 are the credit field; and the last seven bytes, starting at position 20 are the debit field.

The following command yanks the account number and the credit and debit fields from **journal**, makes a new table with a new field for the description:

```
fixtotable Account=7,6 Debit=13,7 Credit=20,7 Description < journal
```

This produces the following table:

```
Account Debit   Credit  Description
------- -----   ------  -----------
101     25000
211.1           25000
150.1   10000
101     5000
211.2           5000
130     30000
```

See Also

compress, tabletofix

foot — /act/gl Command

Foot or subtotal Debits and Credit of ledger
foot

The command **foot** is part of the accounting/general-ledger system included with **/rdb**. It subtotals the accounts in table **ledger**.

Example

The command **foot** produces a table that resembles the following:

```
Account Date    Debit   Credit  Ref     Short   Description
------- ------  -----   ------  ------- ------- ------------------
1010    851231            1758          cash
1010    851231   5137                   cash
------- ------  -----   ------  ------- ------- ------------------
1010             5137     1758

3010                      1601  deposit sales
------- ------  -----   ------  ------- ------- ------------------
3010                0     1601

3080    850125             250  c1459   royalty book fees
------- ------  -----   ------  ------- ------- ------------------
3080                0      250

3100    850118            3286  c101    income  sales
------- ------  -----   ------  ------- ------- ------------------
3100                0     3286
```

```
4020             1601           check   merch
------- ------  -----   ------  ------- ------- ------------------
4020             1601        0

4370    850101      2           r       parking parking at clients
------- ------  -----   ------  ------- ------- ------------------
4370                2        0

4380    850103     15           c115    travel  shuttle bus
4380    850104     13           v       travel  meal
4380    850105     13           c116    travel  meal
4380    850106    114           r       travel  hotel
------- ------  -----   ------  ------- ------- ------------------
4380              155        0
```

See Also

act

getjournal — /act/gl Command

Copy journals for sales, purchasing, pay systems
getjournal *journalname*

The command **getjournal** is part of the accounting/general-ledger system included with **/rdb**. It copies, totals, and adjusts journals from other parts of the business system. These subsidiary journals are created automatically by the operations of sales, purchasing, and other departments. It also gets the inventory value for the table **journaladjust**.

Example

The command

```
getjournal journaldeposit
```

reads the table **journaldeposit** and prints on the standard output a table that resembles the following:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------
891231  13500.00        sales   deposit from sales/deposit
```

See Also

act, consolidate

gregorian — /rdb Command

Convert column of dates for arithmetic and format change
gregorian [-ceu] [*Column ...* **] <** *tableorlist*

The command **gregorian** converts Julian dates in one or more **Column**s to Gregorian date. When used with the command **julian**, you can do the following:

1. Add days to a date.
2. Subtract days from a date.
3. Subtract two dates to find the number of days between them.
4. Change the format of a date.

A Gregorian date is a string that presents the date of the current year, month, and day; for example "911231". A Julian date, however, is the number of days since the beginning of the Julian calendar (January 1, 4713 BC). This system is used by astronomers, computer programmers, and other people who wish to store dates in a form that can be easily translated from one system of dating to another, without having to worry about leap years, dates of the births of religious figures, etc. Because Julian dates are integers, you can subtract one from another to find the number of days between two dates.

To perform arithmetic on dates, first use the command **julian** to convert the dates from Gregorian form to Julian. Then, use the command **compute** to add or subtract the dates (or add a constant to a date). Finally, use **gregorian** to convert the sum or the difference back to Gregorian form.

Options

gregorian recognizes the following options:

-c Output *computer* date format, i.e., *YYMMDD*. **/rdb** uses this format by default for its Gregorian dates because it can be sorted and selected in correct numerical sequence.

-e Output *European* date format, i.e., *DD/MM/YY*.

-u Output *U.S.* date format, i.e., *MM/DD/YY*.

Example

The first example shows how to add dates. Suppose we have a file called **journal**, as follows:

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  -----------------------------------
861222  101     25000           cash from loan
861222  211.1            25000  loan number #378-14 Bank Amerigold
861223  150.1   10000           test equipment from Zarkoff
861223  101               5000  cash payment
861223  211.2             5000  note payable to Zarkoff Equipment
861224  130     30000           inventory - parts from CCPSC
861224  201.1            15000  accounts payable to CCPSC
861224  101              15000  cash payment to CCPSC for parts
```

Now, suppose that we want to add 45 days to each of the dates. First, see what **julian** does. Typing

```
julian Date < journal
```

prints the following on the standard output:

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  -----------------------------------
1752827 101     25000           cash from loan
1752827 211.1            25000  loan number #378-14 Bank Amerigold
1752828 150.1   10000           test equipment from Zarkoff
1752828 101               5000  cash payment
1752828 211.2             5000  note payable to Zarkoff Equipment
1752829 130     30000           inventory - parts from CCPSC
1752829 201.1            15000  accounts payable to CCPSC
1752829 101              15000  cash payment to CCPSC for parts
```

The **Date** column now holds a Julian date. Note we gave no option because computer format is the default.

The next command adds 45 days to each entry in the **Date** column:

```
julian Date < journal | compute 'Date += 45' | gregorian Date
```

This command prints the following on the standard output:

LEXICON

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  ---------------------------------
870205  101     25000           cash from loan
870205  211.1            25000  loan number #378-14 Bank Amerigold
870206  150.1   10000           test equipment from Zarkoff
870206  101              5000   cash payment
870206  211.2            5000   note payable to Zarkoff Equipment
870207  130     30000           inventory - parts from CCPSC
870207  201.1            15000  accounts payable to CCPSC
870207  101              15000  cash payment to CCPSC for parts
```

As you can see, every date is now 45 days later than they originally were.

The next example converts one format of Gregorian date to another. To convert the **Date** column in our example table from computer format to U.S. format, type the following:

```
julian Date < journal | gregorian -u Date | justify Date
```

The result is as follows:

```
Date            Account Debit   Credit  Description
--------        ------- -----   ------  ---------------------------------
12/22/86        101     25000           cash from loan
12/22/86        211.1            25000  loan number #378-14 Bank Amerigold
12/23/86        150.1   10000           test equipment from Zarkoff
12/23/86        101              5000   cash payment
12/23/86        211.2            5000   note payable‘to Zarkoff Equipment
12/24/86        130     30000           inventory - parts from CCPSC
12/24/86        201.1            15000  accounts payable to CCPSC
12/24/86        101              15000  cash payment to CCPSC for parts
```

Because the U.S. format is two characters wider than the computer format, we included the command **justify** in our command line to line things up.

See Also

compute, computedate, julian
COHERENT Lexicon: **awk, date**

hashkey — /rdb Command

Return the hash offset for key strings
hashkey *rows keystring ...*

The command **hashkey** adds the ASCII value of each character in the *keystring*s (except spaces or tabs), and returns the modulo of *row*s. This is for computing the hash offset into hash-index table for the hash fast-access method. It is used by the command **append** to update the hash-index table.

Example

Here we convert several characters. The command

```
hashkey 11 ABCD
```

returns **2**. (That is, ASCII values 65 plus 66 plus 67 plus 68 yields 266; which, when divided by 11 yields a remainder of 2). The command

```
hashkey 29 abcdefg
```

returns **4**.

See Also

append ascii, chr
COHERENT Lexicon: **ASCII**

head line — /rdb Definition

The *head line* is the first line in an **/rdb** table. It names each column in the table. Each entry in the head line must be separated by a tab character.

Example

The table **inventory** appears as follows:

```
Item  Amount  Cost    Value   Description     New
----  ------  ----    -----   -------------  -------
   1       3    50      150   rubber gloves
   2     100     5      500   test tubes
   3       5    80      400   clamps
   4      23    19      437   plates
   5      99    24     2376   cleaning cloth
   6      89   147    13083   bunsen burners
   7       5   175      875   scales
```

The first line

```
Item  Amount  Cost    Value   Description     New
```

LEXICON

is this table's head line.

See Also

dash line, listtotable, tabletolist

headoff — /rdb Command

Remove an /rdb head from both table and list files
headoff < *tableorlist*

The command **headoff** "beheads" *tableorlist*; that is, it strips off its header. Tables have two-line column heads; list files have only one line that consists of a single newline character. The COHERENT command **tail +3** does the same thing for a table; and **tail +2** for a list.

The command **union** uses **headoff** to assemble files. Only the first file keeps its head.

Example

To behead the table **inventory**, type:

```
headoff < inventory
```

This writes something like the following on the standard output:

```
1        3    50      150   rubber gloves
2      100     5      500   test tubes
3        5    80      400   clamps
4       23    19      437   plates
5       99    24     2376   cleaning cloth
6       89   147    13083   bunsen burners
7        5   175      875   scales
```

See Also

headon, insertdash, union
COHERENT Lexicon: **head, tail**

headon — /rdb Command

Add an /rdb header to a table
headon < *table*

The command **headon** appends an **/rdb** header onto *table*. Tables have a two-line column head. When you have a table without an **/rdb** header, perhaps created by another program, you can add a head to it. You can also do this in a pipe and send the new table to other **/rdb** commands for processing.

Example

If you want to treat the file **/etc/passwd** as a table for **/rdb** command processing, use **headon**, as follows:

```
tr ':' '	' < /etc/passwd | headon | justify
```

Note the use of the COHERENT command **tr** to translate a colon ':' into a tab character; and the use of the **/rdb** command **justify** to align the columns. This produces something like the following:

```
1       2                3       4     5               6
------  -------------   ---     ---    --------------- -------------
root    dVAYxSsJy10Fg     0       3     Super User      /
daemon                    1      12     background      /
bin     rVOhF49RpZXEY     2       2     binary programs /bin
uucp                      5       1     UUCP            /usr/lib/uucp
lp                       71       2     Line Printer    /usr/spool/lp
guest                   100     100     Guest User      /usr/guest
rod                       0       0     Rod Manis       /usr/rod
```

Be sure the first line of the table has no blank fields or **headon** counts the wrong number of fields.

See Also

headoff, insertdash, justify, rmblank
COHERENT Lexicon: **passwd, tr**

helpme — /rdb Command

List the help commands available
helpme

The command **helpme** lists all other **/rdb** help commands.

Example

Typing **helpme** prints the following on the standard output:

```
Command                 Description
------------------      ----------------------------------------
"menu"                  a menu with some commands
"rdb"                   a list of the available commands
"commands"              description and syntax of all the commands
"whatis command"        description and syntax of command
"whatwill feature"      info on commands with that feature
"man command"           the manual page for that command
```

howmany — /rdb Command

Display the number of commands in a directory
howmany

The command **howmany** displays the number of the commands it finds in directory **$RDB/bin**.

See Also

rdb, whatis, whatwill

LEXICON

index — /rdb Command

Set up table for search

index [-m[bhirs]] [-x] [-hsl [2> *location***]]** *tableorlist* **[***keycol ...***] [<** *keytable* **]**

The command **index** sets up a table for the program **search**. The indexing it performs increases the speed with which **search** can find a row in a table or list file. This is very important for large files.

Exactly what **index** does depends upon the method of indexing you choose. For some methods, it builds another table, called a *secondary index file*, that the specified method requires. It is possible to index one file by all five methods. Some methods can index all of a table's columns.

Please note that these fast-access methods are static. If the table or list is changed — by a text editor, for example — it may have to be reindexed. Applications should manipulate the table or list files and their secondary index to update them together. Forms packages should dynamically maintain these files.

There is a trick to allow you to index any or all of the columns in a file. Simply use the COHERENT command **ln** to link the file to as many different names as you have columns to index. Then index the table through each of its link aliases. For details, see the COHERENT Lexicon's entry for **ln**.

Options

The following options control the method of indexing that **index** produces:

-m Index using the default method, which is sequential. See **-ms**, below.

-mb Binary indexing. **index** simply sorts the file on the specified key column. No secondary file is created. Because a file can only be sorted on one column, only one column can be indexed using this method. The command **sorttable** gives the same result as this option.

-mh Hash indexing. **index** builds a secondary-index file, named **tableorlist.h**, hashes all of the key-column's values, and writes them and their offsets to the secondary-index file. Please note that because the COHERENT system places a limit of 14 characters on a file name, the name of *tableorlist* must not exceed 12 characters, or **index** will not be able to name correctly the secondary-index file.

-mi Inverted or indexed indexing. **index** builds a secondary-index file of keys and offsets, then sorts on the keys. The secondary file is named *tableorlist***.i**. The trick to index more than one column, previously discussed, alsos work for the inverted method.

If you plan on using the incomplete-match feature of search (which works only with the inverted method), you must specify the **-x** flag, so the secondary index is sorted properly.

-mr Record-number indexing. **index** builds a secondary file named *tableorlist***.r**, which is one fixed-length column of row offsets.

-ms Sequential or linear indexing. **index** does nothing in this case, because **search** simply looks through the whole file.

The following options control details of **index**'s output:

-h No head line on the output table.

-l Write to the standard error the starting and stopping location of each row, and the offset entry in the secondary-index file. These numbers are used by the commands **replace**, **lock**, and **unlock**. The command **seek** uses this option to find the locations of rows. The locations are sent to standard-error device, which under the Bourne and Korn shells is file descriptor 2; thus, you can redirect them into a file by using the shell operator **2>**. You can store these data until they are needed by your shell programs.

This option produces four integers. If there is no secondary-index file, the start location is zero.

-s Only one row is found. This speeds up some of the methods because they do not have to continue searching after they find the first match. The sequential method will, on average, run twice as fast; whereas the hash, binary, and inverted methods will run slightly faster. The record-access method will not change, as it only finds one match anyway.

Methods and Suffixes

The following gives suffixes that **index** appends to names of tables, in order to name its secondary-index files.

Method	*Suffix*	*Example*
-b	None	...
-h	.h	inventory.h
-i	.i	inventory.i
-r	.r	inventory.r
-s	None	...

Example

Let's look at the first ten lines of an accounts chart, named **chart**:

```
Account Name
------- ---------------------
100     Assets
101     Cash
111     Accounts Receivable
111.1   Allowance for Bad Debt
115     Notes Receivable
120     Deposits
130     Parts Inventory
150     Equipment
```

Access to the data in **chart** can be sped up by indexing it and using the command **search** to find the row you want. To index **chart** using the binary (sorted) search method, type:

```
index -mb chart Account
```

index quietly returns when it finishes. You are now ready to use **search** to find an **Account** number in the **chart**

See Also

append, delete, lock, replace, search, unlock, update
COHERENT Lexicon: **ln**

LEXICON

insertdash — /rdb Command

Insert dash line as second line in table
insertdash < *table-without-dashline*

The command **insertdash** writes a line of dashes (hyphens) as the second line of the table. It makes the dash line by copying the head line and turning every character in the first into a hyphen except the tabs.

This command is useful for files created by the system or non-COHERENT programs that do not have the dash line. However, you will also need to convert field separators into tabs. You can do this with the COHERENT commands **tr** or **sed**.

Example

insertdash assumes a table that resembles the following:

```
Item    Amount  Cost    Value   Description
   1         3    50      150   rubber gloves
```

After running **insertdash** on it, this table looks like this:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
```

See Also

dashline, headoff, headon
COHERENT Lexicon: **sed, tr**

intersect — /rdb Command

Write table of rows that are in both input tables
intersect *table1 table2*

The command **intersect** is a shell script that uses the COHERENT commands **sort** and **uniq** to perform a logical AND on two tables. It produces a new table that consists only of the rows that are in both of the input tables. The two input tables must have the same number of columns (called *union compatible*), and rows must match exactly to be considered the same.

Example

Consider the table **journal**, which appears as follows:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820107    14.00 meal    v       meal with jones
820114    81.80 insur   c       car insurance allstate
820119    81.72 vitamin c       sundown vitamins
820121    20.83 meal    v       meal with scott
820121  2500.00 keogh   c       keogh payment
820125    99.00 dues    v       dues to uni-ops unix conference
```

And consider the table **carexpense**, which appears as follows:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820113   101.62 car     v       car repairs
820114    81.80 insur   c       car insurance allstate
820114    93.00 car     c       car registration dmv
```

To see if any rows in **journal** are also in **carexpense**, type the command:

```
intersect journal carexpense
```

In this example, you would see the following on the standard output:

```
Date    Amount  Account Ref     Description
------- ------- ------- ------- --------------------------------
820114    81.80 insur   c       car insurance allstate
```

As you can see, the insurance expense was already posted to the general journal.

See Also

difference, jointable, union
COHERENT Lexicon: **sort, uniq**

invoice — /act/sales Command

Print invoice for a sale order
invoice

The command **invoice** is part of the accounting/sales system that is included with **/rdb**. It prompts for the number of a sales order and then prints an invoice for that order.

Example

The following walks you through an example session with **invoice**:

```
Please enter order number (or Return to exit): 1
                        Invoice
                        -------
Makeapile, Inc., 123 Bigbucks Blvd., Dallas, TX 12345, 1-800-SOF-WARE

Order Number    1
Date Ordered    850723
Date Shipped    0

Mailing Address for Customer Number  2
Customer Number 2
Mr. Thomas  Boomer  President

Ye Olde Thermonuclear Bombe Shoppe
54321 Blooy Road
Livermore, California    USA

Order   Number  Code    Backord Qty     Price   Total   Name
------- ------- ------- ------- ------- ------- ------- ----
1       1       rdb          10 1          1500 1500.00 /rdb

                                       Gross   1500.00
                                       Tax       75.00
                                       Total   1575.00

Please enter order number (or Return to exit):
```

See Also

act, cstate

LEXICON

jointable — /rdb Command

Join two tables into one where keys match
jointable [-a[1|2]] [-j[1|2] *column*]**... -n -]** *table1* **[***table2***]**

The command **jointable** joins *table1* to *table2* on *column*, which is common to both tables (called the *key column*). If no *column* is named, **jointable** assumes that the first column in each table is the key column.

Each table must be sorted on the key column. **jointable** writes a new table that has one row for the rows in the tables whose values in the key column match. The columns of the new table are all the columns in *table1* followed by all the columns in *table2*— with the sole exception that the key column is not repeated.

This is called a *natural join*. An *equi-join* repeats the key column.

The COHERENT system's command **join** will also join two tables in the manner described above, but it does not know about **/rdb** head lines.

String vs. Numeric Sort

The commonest problem seen with **jointable** is when users attempt to join tables that have been sorted *numerically* on the key columns (that is, the columns that the tables are going to be joined on). They use **sorttable** with the **-n** option (for numeric) to sort a table, then call **jointable. jointable**, however, expects to see a *string sort* fails to join the tables correctly.

The following compares a string sort with a numeric sort:

String	*Numeric*
1	1
10	2
2	10
20	20

Can you see why? Numeric is the way we count and is obviously correct. But string sort looks at the first character, then the second. A space sorts ahead of '0', so '1' comes before "10". More importantly, '1' comes before '2' regardless of the characters to the right. That is why "10" comes before "2" followed by a space. This problem shows up as empty output from a **jointable**, when you know some of the rows should be joined.

Right-justification makes numeric and string sorts the same. The leading spaces help the string sort to sort numbers correctly.

To make it easy for users to avoid this problem, **jointable** has the option **-n**, for symmetry with the COHERENT command **sort**.

If you want to join two tables through a numeric key column, you must either call the **sorttable** without its **-n** numeric sort option, so that it defaults to string sort; or if you use the **-n** option for **sorttable**, be sure to use it also with **jointable**.

Options

jointable recognizes the following options:

-a*n* All or one of the files (1 or 2) is output, whether it matches or not. In addition to the normal output, **jointable** produces a line for each unpairable line in file *n*, where *n* is 1 or 2. For example, the command

```
jointable -a1 ledger chart
```

forces **jointable** to print every row in table **ledger**, regardless of whether any value in **ledger**'s **Account** column matches any value in **chart**'s **Account** column.

-j*n column*
Join on the *column* column in table number *n*. If *n* is missing, use the *column* column in each table. For example, the command

```
jointable -j1 Account -j2 account ledger chart
```

joins tables **ledger** and **chart** using the **Account** column in **ledger** and the **account** column in **chart**. If no columns are specified with this option, **jointable** joins the tables by the first column in each. Thus, this option lets you join on any column defined in either file, even when they have different names and locations.

-n Numeric join. This means that table1 and table2 were sorted with the numeric option **-n**. Remember that if you use **-n** with the command **sorttable**, you must also use it with **jointable**.

- Read the standard input in place of a table. For example, the command

```
sorttable < journal | jointable - chart
```

means to sort table **journal**, then join it with table **chart** so that **journal** is on the left of the output table and **chart** is on the right.

Note that this option is used only in place of *table1*. If *table2* is absent from the command line, **jointable** automatically reads the standard input for that table's data.

Note that most of these options are about the same as for the COHERENT command **join**.

Example

Consider table **chart**, as follows:

```
Account Name
------- ----
100     Assets
101     Cash
111     Accounts Receivable
111.1   Allowance for Bad Debt
115     Notes Receivable
120     Deposits
130     Parts Inventory
 ...
```

and table **ledger**, as follows:

LEXICON

```
Account Date    Debit   Credit
------- ----    -----   ------
101     820102          25000
101     820103           5000
101     820104  15000
130     820104  30000
150.1   820103  10000
201.1   820104  15000
211.1   820102          25000
211.2   820103           5000
```

Note that each is sorted in string fashion on its first column. To join them, use the command:

```
jointable ledger chart
```

This joins the tables on the first column in each, and writes the following to the standard output:

```
Account Date    Debit   Credit  Name
------- ----    -----   ------  ----
101     820102          25000   Cash
101     820103           5000   Cash
101     820104  15000           Cash
130     820104  30000           Parts Inventory
150.1   820103  10000           Test equipment
211.1   820102          25000   Notes Payable - BA
211.2   820103           5000   Notes Payable - Z Equip
```

As you can see, column **Name** has been added to the columns of the **ledger** table, and that column **Account** is not repeated. Also note that **jointable** wrote only the rows whose values in column **Account** match. If a table has several rows with repeated key values (e.g., the three **101**s in the **Account** column), the joined table's rows are also repeated (see the three **Cash**s in column **Name**).

If you have a table named **journal** that is sorted on column **Account**, as follows

```
Date    Account Debit   Credit
----    ------- -----   ------
820104  101             15000
820103  101              5000
820102  101     25000
820104  130     30000
820103  150.1   10000
820104  201.1           15000
820102  211.1           25000
820103  211.2            5000
```

you could join it with table **chart** by typing this command:

```
jointable -j Account journal chart
```

The following gives another way for the same result (because **chart**'s join column defaults to column 1):

```
jointable -j1 Account journal chart
```

Both commands produce the following output:

```
Date    Account Debit   Credit  Name
------  ------- -----   ------  ----
820104  101             15000   Cash
820103  101              5000   Cash
820102  101     25000           Cash
820104  130     30000           Parts Inventory
820103  150.1   10000           Test equipment
820102  211.1           25000   Notes Payable - Bank of Amerigold
820103  211.2            5000   Notes Payable - Z Equipment
```

The **-a** option lets you see all of one table whether or not there is a match. For example, the command

```
jointable -a1 ledger chart
```

writes the following:

```
Account Date    Debit   Credit  Name
------- ----    -----   ------  ----
101     820102          15000   Cash
101     820103          5000    Cash
101     820104  25000           Cash
130     820104  30000           Parts Inventory
150.1   820103  10000           Test equipment
201.1   820104  15000
211.1   820102  25000           Notes Payable - Bank of Amerigold
211.2   820103  5000            Notes Payable - Z Equipment
```

Note that column **Account**'s value **201.1** in the ledger shows up here, even though it is not in **chart**. This shows us that **201.1** is an error. All account numbers should be in the chart of accounts. We should either add **201.1** to the chart of accounts, or find the correct number in the chart of accounts and replace **201.1** in the **journal** with the correct account number.

If you pipe a file into a **jointable**, you can control which table it will be (first or second) by using the dash option. For example:

```
sorttable Account < journal | jointable -j Account - chart
```

This produces the following output:

```
Date    Account Debit   Credit  Name
----    ------- -----   ------  ----
820103  101             5000    Cash
820104  101             15000   Cash
820102  101     25000           Cash
820104  130     30000           Parts Inventory
820103  150.1   10000           Test equipment
820102  211.1           25000   Notes Payable - Bank of Amerigold
820103  211.2           5000    Notes Payable - Z Equipment
```

Note the **-** before **chart**. It says that the standard input file will be file 1 or on the left side of the table.

See Also

column, sorttable
COHERENT Lexicon: **join**

LEXICON

julian — /rdb Command

Convert column of dates for arithmetic and format change
julian [-ceu] [*Column ...* **] <** *tableorlist*

The command **julian** converts Gregorian dates in one or more columns to Julian format. For details on the differences between Gregorian and Julian dates, and how to use these commands in scripts, see the manual entry for the command **gregorian**.

See Also

computedate, gregorian, todaydate

justify — /rdb Command

Left or right justify the columns of a table
justify [-elrct'*c*'**] [** *column ...* **] <** *table*

The command **justify** right-justifies the columns that contain only numbers, and left-justifies the columns that contain any nonnumeric characters. It also lines up floating-point numbers on their decimal points.

justify improves the appearance of a table. This makes it easier for you to read and edit. It also makes a column of variable-length strings into one length, by padding each string with spaces on the right. Note that the extra spaces inserted to improve a table's appearance could easily double or triple the size of the table. This increased size not only takes up disk space, but slows processing of *table*. You must decide whether appearance or size is more important.

Options

justify recognizes the following options:

-e Expand the column to the next tab stop. Ordinarily, a column is only as wide as the widest data or column name. This improves the appear, but increases the size of the file.

The following options override the default justifications (left-justify character strings, right-justify numbers):

-l (This is the letter "el", not the number one.) Left-justify all named columns.

-c Center all named columns.

-r Right justify all named columns.

These options are sticky, meaning that if you follow an **l**, **c**, or r option with several column names, each is treated according to the option. For example, **-l Description Account** left justifies both **Account** and **Description**, regardless of the data each contains.

-t´*c*´ Use the character *c* as the column separator instead of tab. *c* can be any ASCII character. Note, however, that once you process a table with this option, you can no longer use **/rdb** commands on it, because all **/rdb** commands require tabs as column separators. See the command **trim** for a discussion of how you can protect special characters from the shell.

Example

One use of **justify** is to align columns. Variable-length character columns (like names and descriptions) destroy the alignment of columns. For example, notice how the variable-length names cause the tabs to break at different points in the following table:

```
Name    Credit
----    -------
Anderson  23.49
Ho  145.98
Johnson 1.15
```

One solution is to move the variable length column to the last column:

```
     column Credit Name < credit
```

The table now appears as follows:

```
Credit  Name
------- ---------------
23.49   Anderson
145.98  Ho
1.15    Johnson
```

Another solution is to use **justify** to right-pad the **Name** column, and (while we're at it) align the floating point number in the **Credit**:

```
Name       Credit
--------   ------
Anderson    23.49
Ho         145.98
Johnson      1.15
```

Remember that the file is now larger. All of those extra blank spaces can double or triple the size of your file.

It is easier to enter the initial data into a table without the padding characters. Then you can use **justify** to pretty up the table.

When using the command **listtotable** to convert a list-format file to a table, you may wish to use **justify** to make the resulting table readable.

See Also

column, compresss, listtotable, trim

LEXICON

label — /rdb Command

Print mailing labels from a mailing list
label < *list*

The command **label** prints mailing labels from a mailing list, in list format. This command is obsolete because the command **report** can make labels more easily; but it is is retained as an example of shell programming.

label is a shell script so you can change it easily to handle different lists and labels.

Example

If you had a file called **maillist** that looked like this:

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

The command

```
label < maillist
```

generates mailing labels that look like this:

```
Ronald McDonald
McDonald's
123 Mac Attack
Memphis, TENN 30000


Chiquita  Banana
United Brands
Uno Avenido de la Reforma
San Jose, Costa Rica 123456789
```

See Also

letter, report

length — /rdb Command

Return the length of its argument or input file
length [*string***] [<** *file***]**

The command **length** outputs the number of characters in its first argument. If there is no argument, it reads the standard input.

Because **length** counts only its first argument, you must put quotation marks around all of the words on the command line to make them one argument.

length uses **wc -c** to do the counting.

Example

Here we find the length of a string and a file.

```
length aA1
```

This returns **3**.

The next command

```
length 'word1 word2'
```

returns **11** — that is, the length of **word1** plus **word2** plus the space between them.

The final example measures the number of characters in file **length.1**:

```
length < length.1
```

Note that you need must quote strings if they have spaces or tabs in them.

See Also

COHERENT Lexicon: **wc**

letter — /rdb Command

Print form letters from a mailing list
letter *letter.1 ...* **<** *maillist*

The command **letter** prints form letters from a standard letter and a mailing list. It is obsolete, because the command **report** is so much more powerful and easier to use; but it is retained for compatibility and as an example of shell programming.

letter is a shell script, so you can be easily changed it to handle different lists and letters. To modify it, copy **letter** from **$RDB/bin/letter** to your local directory, then use your text editor to edit it. Then you can modify it with a text editor to handle any special features of your mailing list and letters.

Example

If you have a file called **maillist** that looks like this:

```
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

Assume, too, that the two parts of your letter are in two files. **letter.1** contains:

```
We have a new product that we want to sell to everyone at
```

And **letter.2** contains:

```
Please have everyone send us an order.  We will be very grateful.

Sincerely,

Mr. Marketing
Makeapile, Inc.
```

To assemble the letter files and the mailing list into one set of mail-merged letters, type the following command:

```
letter letter.1 letter.2 < maillist
```

This produces:

```
Ronald McDonald
McDonald's
123 Mac Attack
Memphis, TENN 30000

Dear Ronald McDonald:

We have a new product  that  we  want  to  sell  to  everyone  at
McDonald's.   Please  have everyone send us an order.  We will be
very grateful.

Sincerely,

Mr. Marketing
Makeapile, Inc.
```

< *... spaces to bottom of letter ...* >

```
Chiquita  Banana
United Brands
Uno Avenido de la Reforma
San Jose, Costa Rica 123456789

Dear Chiquita  Banana:

We have a new product that we want to sell to everyone at  United
Brands.   Please have everyone send us an order.  We will be very
grateful.

Sincerely,
Mr. Marketing
Makeapile, Inc.
```

< ... spaces to bottom of letter ... >

The following sample letter program prints out the mailing name and address, and the salutation **Dear** *Name*. It then **cat**s **letter.1**, echos the company name, and finally cats **letter.2**. **letter** writes its output to temporary file **TMP**, then **nroff**'s the **TMP** file to the standard output. **nroff** reformats the file (left and right justified) for each company name's length.

By studying the **letter** shell script and the COHERENT shell's documentation **(sh** or **ksh)**, you can assemble fancy, complex form letters.

See Also

label, report
COHERENT Lexicon: **cat, ksh, nroff, troff, sh**

like — /rdb Command

Find names that sound like another name
like *similar-name file* **[** *name-column* **]**

The command **like** converts a name to Knuth's soundex code and looks it up in the **.x** secondary-index file of a name. To work, you must have first run the command **soundex**, to create the soundex file.

Example

See **soundex** for examples.

See Also

soundex

listtosh — /rdb Command

Convert list format to shell variable
listtosh < *list*

The command **listtosh** converts a list-format file (*column***<tab>***data*) to a shell variable format (**VARIABLE='***value***'**). You can use this to move a record into a shell variable.

Example

Consider file **maillist**:

LEXICON

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Menphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita Banana
Company United Brands
Street  Uno Avenito De La Revolution
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

The command

```
eval 'row 'Number == 2' < maillist | listtosh'
```

transforms the mailing list into the following format:

```
Number='2'
Name='Chiquita Banana'
Company='United Brands'
Street='Uno Avenito De La Revolution'
City='San Jose'
State='Costa Rica'
ZIP='123456789'
Phone='1234'
```

Once this is done, test the transformation by typing:

```
echo "$Name's phone number is $Phone."
```

This writes the following on the standard output:

```
Chiquita Banana's phone number is 1234.
```

See Also

tabletolist

listtotable — /rdb Command

Convert from list to table format
listtotable [-el] < *listfile*

listtotable converts a file in list format to table format. Some **/rdb** programs must see files in table format, so the conversion is necessary.

In addition, table-formatted files are processed faster than list-formatted files. Any time you send a list-formatted file through a pipe of two or more programs, even if they can handle list format, convert them to table format so that the programs in the pipe will run two or three times faster on average.

The reason that table-formatted files run so much faster is that a list file is often two or three times larger than a table because of the repeated column names. Therefore, when handling list-formatted files, the programs have to process two or three times as many characters.

Options

listtotable recognizes the following options:

-e Expand each column to next tab stop. Ordinarily, table columns heads are only as wide as list column heads.

-l No head line in the list file. This option makes it possible to convert and sometimes use the older list-formatted files.

Earlier versions of **/rdb** did not require the leading newline in the list-formatted file; however, this requirement was added by later editions of **/rdb** to make it easy for its commands to distinguish a list file from a table.

List and Table-Format Rules

List format looks like this:

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
 ...
```

Remember there are three rules for constructing a list-format file:

1. The first character in the file is a newline. Be sure that there are no unseen spaces, tabs, or nonprinting characters on the first line, or things will get quite messed up.

2. Each line consists of two columns separated by a tab. The first column is gives the column's name, and the second column gives its values.

3. A newline character follows each *row* (record).

And there are three rules for contructing a table, as follows:

1. One tab between each column in a row, and a newline at the end.

2. Name of the column at the top of the column, in the first line of the file.

3. A dash line as the second line in the file.

Example

The command

```
listtotable < maillist
```

converts list file **maillist** into table format. The result appears as follows:

LEXICON

```
Number  Name    Company Street  City    State   ZIP     Phone
------- ------- ------- ------- ------- ------- ------- -------
1       Ronald McDonald McDonald's      123 Mac Attack  Memphis
TENN    30000   (111) 222-3333
2       Chiquita  Banana        United Brands   Uno Avenito De La
Revolution      San Jose        Costa Rica      123456789       1234
```

When you look at this messy table, you can see the need for list format. The information in each column is often too wide for the screen and wraps around to the next line.

See Also

tabletolist

Notes

If you use this command with a table instead of a list file, you will see a mess.

lock — /rdb Command

Lock a record or field of a file
lock *filename processid from to indexfrom indexto*

The command **lock** locks *filename* by writing a row into the lock file **/tmp/L***filename*, which contains one line for each locked record or field. Its command line contains the following arguments:

processid
: The process identifier of the program that locks *filename*.

from The beginning point of the bytes to be locked.

to The end point of the bytes to be locked.

indexbegin
: The beginning point of the index entry for the data being locked.

indexend
: The end point of the index entry for the data being locked.

When a process attempts to lock a string of bytes that is already locked, **lock** returns an error condition. You can test for this by using the COHERENT command **test** to test the shell variable **$?**.

Example

Let's use **lock** on table **inventory**, which appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

First, use **seek** to locate the record and return the offset and size, as follows:

```
LOCATION='echo 5 | seek -mb inventory Item'
echo $LOCATION
207 245 0 8
```

This means that the record is offset 207 bytes into file **inventory**, and that it is 38 bytes long, ending at byte 245. The 0 and 8 mean that there is no secondary index file.

Now we can call **lock** to lock the record:

```
lock inventory $$ $LOCATION
```

Note that we use the shell variable **$LOCATION** to supply the arguments *from* and *to*. We also use the built-in shell variable **$$** to write the process identifier on the command line.

The lock **/tmp/Linventory** has the following contents:

```
207 245 0 8
```

See Also

seek, unlock
COHERENT Lexicon: **test**

lowercase — /rdb Command

Transform text to lower case
lowercase [*string ...***] [<** *file***]**

The command **lowercase** converts to lower case every letter in *string* or *file*. If there are no arguments, it reads and converts the standard input. It uses **tr**, the COHERENT character-translation command, to do the converting.

Example

The command:

```
lowercase BIG WORDS
```

prints

```
big words
```

on the standard output.

If the file **CAPITALS** contains the string

```
THESE ARE ALL CAPS.
```

then the command

```
lowercase < CAPITALS
```

prints on the standard output:

```
these are all caps.
```

See Also

cap, uppercase
COHERENT Lexicon: **tr**

LEXICON

maximum — /rdb Command

Display the maximum value in a column
maximum [-l] [*column ...* **] <** *table*

The command **maximum** displays the maximum value of each *column*. If no *column* appears on the command line, **maximum** returns the maximum value in every column in *table*.

The **-l** option writes the table to the standard output, as well as the minimum value for each named column.

Example

For an example of this command, see the entry for **total**.

See Also

mean, minimum

mean — /rdb Command

Display the mean of a column
mean [-l] [*column ...* **] <** *table*

mean displays the mean value of each *column*. If no *column* appears on the command line, **mean** returns the mean value for each column in *table*.

mean assign the value of zero to a blank line. This changes the value of the mean. If that is not what you want, use the command **rmblanks** or **compress -b** to remove blank lines. **mean** also treats blank values as zero

The option **-l** prints the contents of the table, as well as the mean value for each named column.

Example

For an example of this command, see the entry for **total**.

See Also

maximum, minimum

menu — /rdb Command

Root menu with some COHERENT commands
menu

The command **menu** displays a table of commands that you can execute by simply typing a number or a name. Menus are useful for inexperienced users. Experienced users can set up a menu very easily.

menu is a shell script that simply echos the **menu** table of choices, then waits for the user to type a number or name. Then **menu** uses a shell **case** statement to execute a command or series of

commands for each choice. To set up your own menus, simply copy the **menu** shell command from the **$RDB/bin** into your directory, then edit it. You should change its name to **Menu** or **Menu.something**, to keep from confusing **/rdb**.

There is a sample local **menu,** named **Menu** in directory **$RDB/demo**.

Example

The following gives an example menu:

```
                      COHERENT MENU
Number  Name    For
------- ------- ------------------------------------------------------
  0     exit    leave menu or return to higher menu
  1     Menu    goto another local menu (if any)
  2     sh      get COHERENT shell
  3     vi      edit a file
  4     mail    read mail
  5     send    send mail to someone
  6     cal     see your calendar
  7     who     see who is on the system
  8     ls      list the files in this directory
  9     cat     display a file on the screen
 10     rdb     display rdb commands

Enter a number or name for the action you wish or DEL to exit:
```

menu waits for you to type your choice of number or name. Numbers are quicker to type, but names are more easily remembered, and help you learn how to do things at the shell level.

Setting Up Your Own Menus

You can automate your operations with menus. If you have naive users who might have trouble learning or remembering the COHERENT or **/rdb** commands, you can set up menus for them. Also, when you have an operation in which a few commands are repeatedly executed, you can put them into a menu for simple choice.

You can put your menus in the directories in which they will be used or in your **$HOME/bin** or other **bin** directory.

To start, go to the directory of your choice and copy the **$RDB menu** to your directory:

```
cp $RDB/bin/menu Menu
```

or

```
cp $RDB/demo/Menu .
```

We suggest the convention of using the name **Menu** with the capital 'M' so it appears at the top of your list of files when you do an **lc** command.

You can then name submenus; for example, **Menu.***other* where *other* is something that suggests the kinds of commands the menu displays.

Now edit the **Menu** file. As you study the code, you will find that it is very easy to create your own directory.

Menu is in two parts. The first part is the menu table:

LEXICON

```
     cat << SCREEN
$CLEAR                          COHERENT MENU
Number  Name    For
------- ------- ------------------------
  0     exit    leave menu or return to higher menu
  1     Menu    goto another local menu (if any)
  2     sh      get unix shell
  3     vi      edit a file
  4     mail    read mail
  5     send    send mail to someone
  6     cal     see your calendar
  7     who     see who is on the system
  8     ls      list the files in this directory
  9     cat     display a file on the screen
 10     rdb     display rdb commands

Please enter a number or name for the action you wish or DEL to exit:

SCREEN
```

The command **cat** simply sends to the screen everything after the command and before the line that begins with **SCREEN**, including the line that begins **Please enter a number**.

The shell replaces **$CLEAR** on the first line with a sequence of characters that clears the terminal. **$CLEAR** must have been set by the following command in the **menu** or in the **.profile** file of the user:

```
CLEAR='clear'
```

The second part of **Menu** is a large **case** statement that you can edit to do what you want. Here is a sample:

```
        read ANSWER COMMENT

        case $ANSWER in

        0|exit) exit 0 ;;
        1|Menu) Menu ;;
        2|sh)   sh ;;
        3|vi)
                echo 'Which file or files do you wish to edit'
                read ANSWER COMMENT
                vi $ANSWER $COMMENT
                ;;

        4|mail) mail ;;
        5|send)
                echo 'Please enter login name of person to send mail to'
                read ANSWER COMMENT
                echo 'Type you letter, and end by typing Ctrl-d'
                mail $ANSWER
                ;;

        6|cal)  (cd ; calendar) ;;
        7|who)  who ;;
        8|ls)   ls ;;
        9|cat)
                echo 'Please enter the name of the file you wish to see'
                read ANSWER COMMENT
                cat $ANSWER
                ;;
```

```
10|rdb) menu.rdb ;;
*)      echo 'Sorry, but that number or name is not recognized.' ;;
esac
```

The command **read** waits for the user to enter his choice. The first word typed is assigned to the shell variable **ANSWER**. If the user types any more words, they are assigned to variable **COMMENT**, which might be used in your commands.

Next is the **case** statement that examines **$ANSWER** to decide which command to execute. Note that the number and name are separated by a vertical-bar character '|' to mean **or**:

```
0|exit) exit 0 ;;
```

This line is selected if type either 0 or **exit**. The command **exit** is then executed. The double semicolons ";;" are required to show the end of the commands for this **case** statement.

Each **case** statement can hold any command or series of commands that you could type at your terminal. **esac** at the bottom is **case** spelled backwards; it marks the end of the whole **case** statement.

The expression ***)** catches any pattern that does not match earlier patterns. In this case, a message is displayed. The pound sign '#' marks the beginning of a comment.

You have great power to do things as a result of the user selecting a choice. Of course, you need to match the name and number in your menu table with the patterns in the **case** statement; it is easy to edit the first part and forget to edit the second.

See Also

COHERENT Lexicon: **case, cat, clear, echo, esac, ksh, read, sh**

minimum — /rdb Command

Display the minimum of each column selected
minimum [-l] [*column ...* **]** **<** *table*

The command **minimum** returns the lowest value of each *column* in *table*. If no column is named on the command line, **minimum** returns the minimum value for every column in *table*. Blank lines are given the value of zero.

Option **-l** returns the contents of *table*, as well as the minimum value of each *column*.

Example

For an example of using the command in a script, see the **total**

See Also

maximum, mean, total

LEXICON

not — /rdb Command

Logical not, to reverse return status of command
not *command ...*

The command **not** converts a return status of zero to 255, and any nonzero return status to zero. Thus, true becomes false and vice versa. This is used to test a command for not true or false.

Example

This example shows how **not** reverses the return code of the COHERENT command **true**:

```
true ; echo $?
```

This command returns zero. On the other hand, the command

```
not true ; echo $?
```

returns 255.

See Also

COHERENT Lexicon: **false, true**

number — /rdb Command

Insert a column-row number into a table or list
Number < *tableorlist*

The command **number** creates appends a new column to the beginning of a table, or as the first field in a list. The newly created column contains the number of each row in the table. Unlike the related command **Number**, **number** writes the word "number" in lower-case letters as the header on the column.

Example

For example, consider the following table, called **inventory**:

```
Item#        Amount        Cost       Value     Description
-------     -------     -------     -------     ---------------
      1          3        5.00       15.00     rubber gloves
      2        100        0.50       50.00     test tubes
      3          5        8.00       40.00     clamps
      4         23        1.98       45.54     plates
      5         99        2.45      242.55     cleaning cloth
      6         89       14.75     1312.75     bunsen burners
      7          5      175.00      875.00     scales
```

Typing the command

```
number < inventory
```

yields the following result:

```
 number      Item#     Amount       Cost      Value     Description
-------    -------    -------    -------    -------    ---------------
      1         1          3       5.00       15.00     rubber gloves
      2         2        100       0.50       50.00     test tubes
      3         3          5       8.00       40.00     clamps
      4         4         23       1.98       45.54     plates
      5         5         99       2.45      242.55     cleaning cloth
      6         6         89      14.75     1312.75     bunsen burners
      7         7          5     175.00      875.00     scales
```

Note that the first column is the new number column.

See Also

Number, addcol, column, jointable, rename

Notes

To change the name of a column, use the command **rename**. To move the row-number column to a different position within its table, use the command **column**. If you want to change the name of the number column to something else, use the command **rename**.

Number — /rdb Command

Insert a column-row number into a table or list
Number < *tableorlist*

Number creates appends a new column to the beginning of a table, or as the first field in a list. The newly created column contains the number of each row in the table. Unlike the related command **number**, **Number** capitalizes the column heading — that is, the name of the command sets the name of the column.

See Also

number

Notes

To change the name of a column, use the command **rename**. To move the row-number column to a different position within its table, use the command **column**. If you want to change the name of the number column to something else, use the command **rename**.

pad — /rdb Command

Add extra spaces at end of last column
pad [-*number***] <** *tableorlist*

The command **pad** adds extra spaces to the end of the last column of each record in *tableorlist*. *number* gives the number of spaces to add to the last column. The default is 70.

pad is useful for preparing up a file for updating by the command **update**. **update** writes an edited record back into the file in the same place it was taken out. If you add characters while editing the record, it will not fit.

The command **replace** looks for trailing and leading spaces to trim, to fit the record back. If it does not find enough room, it appends the record to the end of the file. Padding gives each record a fixed number of extra spaces to be trimmed as needed.

The traditional solution is to use fixed-length records and fields. To do this, you can use the command **justify** to pad the file; but this often increase the file by two or three times. That makes the file much larger, and increases the time needed to process it. **pad** lets you control how much larger each record will be.

Example

To pad file *maillist*, type:

```
pad < maillist > tmp
```

To **see** the results, pipe the results to the command **see**:

```
pad < maillist | see
```

The following appears on the standard output:

```
Number^J1$
Name^JRonald McDonald$
Company^JMcDonald's$
Street^J123 Mac Attack$
City^JMenphis$
State^JTENN$
ZIP^J30000$
Phone^J(111) 222-3333                                                $
$
Number^J2$
Name^JChachita Banana$
Company^JUnited Brands$
Street^JUno Avenito De La Revolution$
City^JSan Jose$
State^JEl Salvadore$
ZIP^J123456789$
Phone^J1234                                                          $
$
```

Note the dollar signs, which mark the end of the line, are far to the right of the last column, *Phone*. Those are the 70 extra spaces that have been added; 70 being the default number of spaces. On such a small record, that is probably far too many.

Technical

Note that column *Phone* already had a value in it. If a value is already in the last column, **pad** overwrites the spaces. If the value is longer than the number of spaces, **pad** writes the value but adds no spaces.

We recommend that you have a last column named *Comment* or *Remarks* to act as a catch-all for both comments and padding.

See Also

justify, replace, see, update

padstring — /rdb Command

Return string with blanks to fill a field
padstring [-]*length string*

The command **padstring** adds blanks to the end of *string* to make it the requested *length*. This command useful for screen and report writing, where you want to line up everything.

The option *-length* means to left-justify the string in the field, that is, write the padding blanks to the end of *string*. The option *length*, without the hyphen, right justifies the string — that is, it adds the spaces to the beginning of *string*.

Example

The first example

```
padstring 10 string
```

displays the string:

```
      string
```

Note that the string is right-justified; that is, the spaces were added to the left of **string**.

Now we can use this in a **report** form file to make fixed-length fields. An ordinary form has the files to the right of a field, moving all around. For example, the command

LEXICON

```
report form < mailtable
```

writes the output:

```
Name  : Ronald McDonald             Company: McDonald's
Street: 123 Mac Attack
City  : Menphis             State  : TENN       ZIP: 30000
Phone : (111) 222-3333

Name  : Chiquita Banana             Company: United Brands
Street: Uno Avenito De La Revolution
City  : San Jose             State  : Costa Rica       ZIP: 123456789
Phone : 1234
```

This result is rather messy. However, if we edit the file **form** to include the **padstring** command, then we will get a fixed form, as follows:

```
Name  : <!padstring -17 '<Name>'!> Company: <Company>
Street: <Street>
City  : <!padstring -17 '<City>'!> State  : <State>       ZIP: <ZIP>
Phone : <Phone>
```

Now, running **report** on this file yields:

```
Name  : Ronald McDonald   Company: McDonald's
Street: 123 Mac Attack
City  : Menphis           State  : TENN       ZIP: 30000
Phone : (111) 222-3333

Name  : Chiquita Banana   Company: United Brands
Street: Uno Avenito De La Revolution
City  : San Jose          State  : Costa Rica       ZIP: 123456789
Phone : 1234
```

Note that **State** has not been done yet, and therefore **ZIP** moves back and forth; but **Company** and **State** stay lined up correctly.

Note that if **padstring** is too long a name, use the COHERENT link command **ln** to link it to a shorter name.

See Also

report, screen
COHERENT Lexicon: **ln**

paste.rdb — /rdb Command

Paste together two or more tables
paste.rdb *table1 table2 ...*

The command **paste.rdb** displays *table1*, *table2*, and so on, side by side. This is almost identical to the COHERENT command **paste**.

The **/rdb** version of **paste.rdb** uses the COHERENT command **pr** to list the tables side by side:

```
pr -m -t -s table1 table2 [...]
```

paste.rdb differs from the command **jointable** in that **jointable** looks for matching values in key columns and writes only the rows in which it finds a match. **paste.rdb** does not care what it is pasting together. You can get some interesting garbage with **paste.rdb**.

You can also do a kind of visual **diff** in which you can see an old and new file lined up side by side. You could do this with two list files, but be careful, because the output will generally not be acceptable by other **/rdb** programs.

Example

The following commands break the table **inventory**:

```
column Item Value < inventory > tmp1
column Cost Amount Description < inventory > tmp2
```

This yields table **tmp1**

```
Item    Value
-----   -----
    1     150
    2     500
    3     400
    4     437
    5    2376
    6   13083
    7     875
```

tmp2:

```
Cost    Amount  Description
----    ------  --------------
  50         3  rubber gloves
   5       100  test tubes
  80         5  clamps
  19        23  plates
  24        99  cleaning cloth
 147        89  bunsen burners
 175         5  scales
```

Now, you can use **paste.rdb** to put them back together:

```
paste.rdb tmp1 tmp2
```

This writes the following onto the standard output:

```
Item    Value   Cost    Amount  Description
-----   -----   ----    ------  --------------
1       150     50      3       rubber gloves
2       500     5       100     test tubes
3       400     80      5       clamps
4       437     19      23      plates
5       2376    24      99      cleaning cloth
6       13083   147     89      bunsen burners
7       875     175     5       scales
```

This is not an efficient way use **column**; one command could do the job, instead of two. **paste.rdb**, however, it is a good way to put tables together if **jointable** is inappropriate. If you had wanted to use **jointable** here, you should **project** column **Item** into both tables.

See Also

column, jointable
COHERENT Lexicon: **diff, paste, pr**

Notes

This command is named **paste.rdb** rather than **paste**, as in other implementations of **/rdb**, to avoid clashing with the COHERENT command **paste**.

LEXICON

path — /rdb Command

Find the full path of a command
path [*command ...* **]**

The command **path** finds the directory path to a command, if it is in one of the directories in your shell's **PATH** environment variable.

A path means two things in COHERENT:

1. It means the list of nested directories within which a file is located. For example, file **FOO** may have the path of **/usr/local/src/**; this means that FOO is kept in directory **src**, which in turn is in directory **local**, which in turn is in directory **usr**, which in turn is in directory **/**.
2. It means a list of directories that the shell searches to find an executable program that matches the command typed in. Each user has his path stored in the environmental variable **PATH**; to see your path, type the command:

```
echo $PATH
```

path searches the directories named in your **PATH** to find the path of *command*. If **path** cannot find *command* or if *command* is not executable, it returns nothing.

Example

If you type:

```
path column
```

If this command returns

```
/usr/rdb/bin
```

then **column** lives in directory: **/usr/rdb/bin**.

Please note that for **path** to work, you must insert into your **PATH** the directory that holds the **/rdb** commands.

See Also

COHERENT Lexicon: **PATH, .profile**

precision — /rdb Command

Display the precision of a column
precision [-l] [*column ...* **] <** *table*

The command **precision** returns the maximum number of digits to the right of the decimal point in each *column*. If no *column* is named on the command line, **precision** returns information on every numeric column in *table*.

The option **-l** tells **precision** to return the entire *table*, in addition to the precision of each *column*.

Example

For example of this command, see the entry for **total**.

project — /rdb Command

Write selected projects (same as column)
project [*column ...* **] <** *tableorlist*

The command **project** is another name for the command **column**.

See Also

column

prompt — /rdb Command

Echo a string on the standard output
prompt *string ...*

The command **prompt** echoes a string onto the standard output. It resembles the COHERENT command **echo**, but it does not write a newline character after the echoed string. Thus, you can use **prompt** to write a prompt onto the screen.

Example

The command

```
prompt "Please enter your name: "
```

writes the following on the standard output:

```
Please enter your name: _
```

The underscore character marks where this command leaves the cursor.

See Also

chr
COHERENT Lexicon: **echo**

Notes

prompt looks for an environmental variable called **ECHONOCR**. If the variable is missing or not equal to **-n**, then **prompt** calles the COHERENT command **echo** with its **-n** option, which drops the newline. If **ECHONOCR** is set to anything other than **-n**, then **prompt** calls **\c** option, which is used by some UNIX systems. When using **/rdb** under the COHERENT system, it is best just to not set **ECHONOCR**.

LEXICON

rdb — /rdb Command

List all /rdb programs in directory $RDB/bin
rdb

The command **rdb** prints on the standard output all commands in directory **$RDB/bin**. This command is a shell script, which you should edit if its output is not to your liking.

See Also

act, helpme, whatis, whatwill

record — /rdb Command

Find and output a record from a table
record *number* **<** *table*

The command **record** searches through *table* for the record with the serial number *number*. If it finds the record, it prints it and the table's header on the standard output.

Example

To retrieve the third record from table **inventory**, type:

```
record 3 < inventory
```

You will see something like the following:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   3         5    80      400   clamps
```

The following shell script uses **record** and **read** to find specific values in table **inventory**. The command **read** uses the head line of **inventory** to assign values to each column name.

```
: describe - gives the description for an item in inventory
USAGE='usage: describe Item < table'

record $1 < inventory |
(       read HEAD
        read DASH
        read $HEAD
        echo "Item $Item is $Description"
)
```

When you type

```
describe 1 < inventory
```

you see output in the following format:

```
Item 1 is rubber gloves
```

See Also

row, search
COHERENT Lexicon: **read**

rename — /rdb Command

Rename a column
rename [*oldcolumn newcolumn ...* **] <** *tableorlist*

The command **rename** changes the name of *oldcolumn* to *newcolumn* within *tableorlist*. You can name on the command line as many *oldcolumn*/*newcolumn* pairs as you wish.

Example

Consider the table **inventory**:

```
Item#   Amount   Cost     Value Description
-----   ------  -----   ------- --------------
    1        3   5.00     15.00 rubber gloves
    2      100   0.50     50.00 test tubes
    3        5   8.00     40.00 clamps
    4       23   1.98     45.54 plates
    5       99   2.45    242.55 cleaning cloth
    6       89  14.75   1312.75 bunsen burners
    7        5  175.00   875.00 scales
```

The command

```
rename Amount Qty Description Name < inventory
```

renames **Amount** to **Qty**, and **Description** to **Name**. The table now appears as follows:

```
Item#      Qty    Cost    Value Name
-----      ---  ------  ------- --------------
    1        3    5.00    15.00 rubber gloves
    2      100    0.50    50.00 test tubes
    3        5    8.00    40.00 clamps
    4       23    1.98    45.54 plates
    5       99    2.45   242.55 cleaning cloth
    6       89   14.75  1312.75 bunsen burners
    7        5  175.00   875.00 scales
```

If you have a list-formatted file, you can also rename the column names. For example, consider the list **maillist**:

```
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

LEXICON

The command

```
rename Number Num ZIP Zip < maillist
```

changes **Number** to **Num**, and **ZIP** to **Zip**. The list now appears as follows:

```
Num     1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
Zip     30000
Phone   (111) 222-3333

Num     2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
Zip     123456789
Phone   1234
```

See Also

column

replace — /rdb Command

Insert a record into a file at specified location
replace [-a] [-m[bhirs] *tableorlist from to* **[***xfrom xto***] <** *intableorlist*

The command **replace** replaces the row at offsets *from* through *to* in *tableorlist* with the contents of *intableorlist*. **replace** will not insert a record larger than *to* minus *from*. It adds spaces to the *last* column of records within *tableorlist* that are smaller than the input record.

xfrom and *xto* give the beginning and end of the offsets of the index entry within *tableorlist*'s associated fast-access table, should there be one.

To obtain painlessly the offsets of the record and its index entry, use the command **seek**. This is demonstrated in the following example.

The command **update** remembers the location and size of the record it takes out of a file and passes it to **replace** so that only a correctly sized file goes back into that file.

Options

replace recognizes the following options:

-a Do not append the record in *intableorlist* to the end of *tableorlist*. The default to append if the record is too big to fit back into the location from which it was taken. This option allows you to prevent appending. If you want a binary file to stay sorted, you will either have to resort it or not allow appending.

-m[bhirs]
Fast access methods. See the manual page for **index** for a description of what each of these options means.

Example

If table **recordtable** contains one record, you use can insert that record into table **bigfile** with the following command:

```
replace bigfile 1421 1487 0 8 < recordtable
```

Note that the contents of **recordtable** replace rows 1421 through 1487 within **bigfile**.

For the second example, consider again table **inventory**, as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

First, let's use the command **seek** to extract a record, and save its offset in the shell variable **LOCATION**:

```
LOCATION=‘echo 5 | seek -mr -o tmpfile inventory Item‘
```

$LOCATION is initialized to the following

```
207 245 54 62
```

and **tmpfile** contains the following:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   5        99    24     2376   cleaning cloth
```

Here we use the record fast-access method.

Now let's change 99 to 199 within **tmpfile**, so we can see a change:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   5       199    24     2376   cleaning cloth
```

Now we can replace the edited record in **inventory**:

```
replace -mr inventory $LOCATION < tmpfile
```

inventory now appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5       199    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

You can see that row 5 has been updated, because it now has 199 in it.

See Also

seek, update

LEXICON

report — /rdb Command

Write reports using a form and a table
report *form* **<** *tableorlist*

The command **report** produces reports, form letters, mailing labels, invoices, and so forth, for each row in *tableorlist*, following the template within file *form*.

You must use a text editor to fill out *form* to determine how the form will appear. Insert column names from *tableorlist* between angle brackets (< >) where you want the data to appear within the report.

You can even call shell commands from within a form, pass them data from *tableorlist*, print their output within your formatted form. This is a powerful extension to COHERENT shell programming.

report mostly replaces the older commands **label** and **letter**. These commands are still part of **/rdb**, in part for compatibility with older versions, and in part as examples of shell programming.

Form Syntax

The following describes the format of a form:

regular text
: **form** prints any characters in your form that do not match the symbols shown below.

<column-name>
: Angle brackets indicate that data from *column-name* in the current row of *tableorlist* should be inserted in place of the column name.

<!command!>
: Angle brackets and exclamation points indicate that the shell is to execute *command*. Any output from *command* appears at this point in the report.

<!cmd <col>!>
: This is an insertion within a command. The data from column *col* in the current row of *tableorlist* are passed to *cmd*. This passes the contents of *col* as an argument to *command*. The output of *command* then replaces this entry within the form.

If **report** does not recognize a column name, it prints *<column-name>* as if it were normal text. This allows you to use the angle brackets as literal characters within your form, if you wish. It also helps you debug your form: if the data from the table are not replacing *<column-name>*, then *column-name* does not exactly match the column name in the table. Check for spelling errors, upper- and lower-case discrepancies, embedded blanks, and nonprinting characters.

Example

Consider table **maillist**, as follows:

```
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333
```

```
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
```

You could use an editor to develop a form like the following:

```
<Name>
<Company>
<Street>
< City >, <State>  <ZIP>

Hi <  Name >:

The date and time is <!date!>

We are also sending this letter to:

<! column Name City < mailtable | row 'Name != "<Name>"' | justify !>
Bye <! echo <Name> !>
```

Note that we have used all three types of inserts: column name, command, and command insert. Now with these two files ready, you only need type

```
report form < maillist
```

to see the following output:

```
Ronald McDonald
McDonald's
123 Mac Attack
Memphis, TENN  30000

Hi Ronald McDonald:

The date and time is Wed Aug 19 01:58:19 PST 1983

We are also sending this letter to:

Name            City
--------------- -------
Chiquita Banana San Jose

Bye Ronald McDonald
```

LEXICON

```
Chiquita  Banana
United Brands
Uno Avenido de la Reforma
San Jose, Costa Rica  123456789

Hi Chiquita  Banana:

The date and time is Wed Aug 19 01:58:40 PST 1983

We are also sending this letter to:

Name            City
--------------- -------
Ronald McDonald Memphis

Bye Chiquita  Banana
```

Some users have noted that the shell escape places an extra line in the output. If you don't want the extra line say something like:

```
<! echo -n 'date' !>
```

See Also

label, letter
COHERENT Lexicon: **echo**

reportwriter — /rdb Command

Sample program to write standard reports
reportwriter

The command **reportwriter** is a sample shell program that writes a sample standard report. It uses the command **splittable** to divide a table into page-sized tables. It shows several tricks for using COHERENT shell programming and **/rdb** tools to put together reports.

Example

Here is what **reportwriter** produces (with the middle of each page replaced with an ellipsis to keep the listing short):

```
       Prices of Computers that ran UNIX(TM) in 1983
           From Urban Software of New York City
                 Report Date: 9 Feb 1985

Price   Company                          City
-----   -------                          ----
        Advanced Micro Devices           Santa Clara
        Alcyon Corporation               San Diego
        American Telephone & Telegraph
        BASIS Microcomputer GmbH         D-4400 Muenster
        Corvus Systems                   San Jose
        David Computers Inc.             Kitchener
        Digital Computers Ltd.           Tokyo 102
        Heurikon Corp.                   Madison
 ...
        Western Digital Irvine
        Western Electric Corp.
  4.2   Venturcom/IBM PC                 Cambridge
 ...
 13.9   Victory Computer Systems, Inc.   San Jose

        Page 1
```

See Also

splittable

rmblank — /rdb Command

Remove blank rows from a table
rmblank < *table*

The command **rmblank** deletes from *table* all input rows that consist of only white space (spaces and tabs). The related programs **delete** and **replace** blank out records. They can be used in transaction processing where the files are too large to process and several users are updating them. At a quiet time, the files can be compacted and reindexed.

Be sure to reindex any file that is processed by either **rmblank** or **compress.**

Example

First we have table **inventory**, as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

Then we delete record 4

```
echo 4 | delete inventory Item
```

and **inventory** now appears as follows:

LEXICON

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

The command **see** shows the contents of **inventory** actually to be as follows:

```
Item^IAmount^ICost^IValue^IDescription   $
----^I------^I----^I-----^I-------------$
   1^I     3^I  50^I  150^Irubber gloves$
   2^I   100^I   5^I  500^Itest tubes$
   3^I     5^I  80^I  400^Iclamps$
    ^I      ^I    ^I     ^I      $
   5^I    99^I  24^I 2376^Icleaning cloth$
   6^I    89^I 147^I13083^Ibunsen burners$
   7^I     5^I 175^I  875^Iscales$
```

Now the command

```
rmblank < inventory
```

removes lines that consist only of blanks. After we run this command, **inventory** appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

Remember to reindex a file after you run **rmblank** on it.

See Also

compress, delete, index, replace, rmblank

rmcore — /rdb Command

Remove all core files
rmcore

The command **rmcore** uses the COHERENT system's command **find** to search for files named **core** in the current directory and all subordinate directories. As its name implies, it removes all such files that it finds.

Run this command first when you begin to fill up your disk. **rmcore** starts at the directory you are in and goes down the file tree. If you are superuser, you can issue this command at the root directory '/' and clean the whole system.

See Also

COHERENT Lexicon: **core, find, rm**

row — /rdb Command

Make a new table where rows match logical condition
row '*column condition column-or-value ..*' **<** *table*

The command **row** creates a new table from *table*. The new table consists of only those rows that meet *condition*.

row uses a script written in the **awk** language, which gives you great power for specifying the logical condition.

Several conditional expressions can be combined with the logical AND (&&) and OR (||) operators. Because the logical conditions are represented by special characters, they must be enclosed by apostrophes, to protect them from being interpreted by the shell. If you have special characters within a column name, enclose the column name in quotation marks.

Example

First a simple example. Consider table **inventory**

```
Item    Onhand  Cost    Value   Description
----    ------  ----    -----   -----------
1       3       5       15      rubber gloves
2       100     1       100     test tubes
3       5       8       40      clamps
4       23      2       51      plates
5       99      2       97      cleaning cloth
6       89      18      1602    bunsen burners
7       5       175     875     scales
```

To make a table of all of the parts that cost more that $5, type:

```
row 'Cost > 5' < inventory
```

The new table appears as follows:

```
Item    Onhand  Cost    Value   Description
----    ------  ----    -----   -----------
3       5       8       40      clamps
6       89      18      1602    bunsen burners
7       5       175     875     scales
```

Now a more complex example. If you want a table of all the burners and of all of the items of which we have fewer than ten in stock, type:

```
row 'Description ~ /burner/ || Onhand < 10' < inventory
```

The new table appears as follows:

```
Item    Onhand  Cost    Value   Description
----    ------  ----    -----   -----------
1       3       5       15      rubber gloves
3       5       8       40      clamps
6       89      18      1602    bunsen burners
7       5       175     875     scales
```

The string **~ /burner/** searches for the string "burner" anywhere in the column. The double pipe (||) means logical OR.

Column Names

The commands **row**, **compute**, and **validate** recognize column names as upper- and lower-case letters, digits, and the underscore '_'. The regular expression is [A-Za-z0-9_]. If you have any other characters in your column names, put apostrophes or quotation marks around them.

LEXICON

row compares the recognized strings with the table's head line. Only if it finds a match does it convert the string to the **$1, $2...** column positions that **awk** needs.

awk's Reserved Words

awk is the pattern-scanning language that does all of the work for the commands **row**, **compute**, and **validate**.

awk recognizes the following words as having special significance. If you name a columns with any of these reserved words and then try to use them in a query, the **/rdb** commands that are built around **awk** turn them into column positions and **awk** does not see them. It is best to avoid these names as column heads or simply capitalize the first letter of each of your column names.

This list also indicates the many functions you have with **awk**. However, do not rely on the following descriptions. The tutorial section of your COHERENT manual contains a summary of the **awk** language.

Please note that this list of **awk** functions and reserved words comes from UNIX System V. Other implementations of **awk** may not contain all of these functions.

Reserved Word	*Description*
BEGIN	Pattern that matches before first input record
END	Pattern that matches after last input record
break	Exit the nearest **for** or **while** loop
continue	Go to next iteration of **for** or **while** loop
else	Used in **if then else** expression
exit	Leave program entirely, as if at end of input
exp	Raise number to a power
for	**for (***expression* **;** *condition***;** *expression***)**
getline	Get next input line
if	**if (***condition***)** *statement* **[else** *statement***]**
in	**for (***variable* **in** *array***)** *statement*
index	**index (** *string1***,** *string2* **)**
int	Truncate argument to integer
length	Return current line length, or length of argument
log	Return log (to base 2) of argument
next	Skip to next record and reexecute all commands
print	Output variables
printf	**printf ("***format***",** *variable***, ...)**
split	**split (***string***,** *arrayname***,** *separator***)**
sprintf	**sprintf ("***format***",** *variable, ...* **)**
sqrt	Return square root of argument
substr	**substr (** *string***,** *start***,** *number* **)**
while	**while (** *condition* **)** *statement*

Although **row** is built around the **awk** language, **row** has the advantage of knowing about the names of the columns. Therefore, you can use column names instead of column positional numbers. To grasp the full power of **row** and **compute**, read the COHERENT manual's tutorial on **awk**.

See Also

compute, row, select, validate
COHERENT Tutorials: **Introduction to the awk Language**

sale — /act/sales Command

Enter sale order, and item, update customer
sale

The command **sale** is part of the accounting/sales system included with **/rdb**. It enters a sales order into table **saleorder** and items into table **saleitem**. It also permits an order-taker to add a new customer to the customer-information table **customer**, or to update information on an existing customer.

Example

sale is interactive, to allow you to confirm each action. First, it asks for customer number. If you give it a number, it looks the number up in table **customer**. If it finds a customer with that number, it displays information on that customer for you to confirm. If you do not have the customer's number, you can search with the slash string option or enter the question mark to enter **customer**. There, you can search for the customer or add a new customer.

The following gives an example order-entry session:

```
MakeaPile, Inc.                 saleorder
Enter Customer Number (? for new, /string search, Return to exit) 1
Vendor Number 1
Mr. Luke  Skywalker   CEO

Rebel Enterprises
123 Lea Street
Space Port City, Far Side  123456789  Tatooy

Code    Qty
------- -------
rdb 1
Order   Number  Code    Backord Qty     Price   Total   Name
------- ------- ------- ------- ------- ------- ------- -------
1       1       rdb          10 1          1500 1500.00 /rdb
Number  Cust    Date    Shipped Gross   Tax     Total
------  ----    ------- ------- ------- ------- -------
1       2       850723  0       1500.00   75.00 1575.00

Is this correct? (y, n) y
```

After you have confirmed the order, table **sale** modifies table **salesorder**, as follows:

```
Number  Cust    Date    Shipped Gross   Tax     Total
------  ----    ------- ------- ------- ------- -------
1       2       850723  860525  1500.00   75.00 1575.00
2       2       850724  860525  3000.00  150.00 3150.00
3       3       850902  860525  22500.00        1125.00 23625.00
4       3       850902  0       36000.00        1800.00 37800.00
5       3       850902  0       21000.00        1050.00 22050.00
```

LEXICON

It also modifies table **salesitem**, as follows:

```
Order   Number  Code    Backord Qty     Price     Total    Name
------- ------- ------- ------- ------- -------   -------  ----
1       1       rdb          10 1          1500   1500.00  /rdb
2       1       rdb          10 2          1500   3000.00  /rdb
3       1       rdb          10 5          1500   7500.00  /rdb
3       2       act          10 10         1500  15000.00  /act
4       1       rdb          10 5          1500   7500.00  /rdb
4       2       rdb          10 19         1500  28500.00  /rdb
5       1       rdb          10 5          1500   7500.00  /rdb
5       2       act          10 9          1500  13500.00  /act
```

Once you enter the correct customer number, **sale** gives you a table of items to enter. Type in **Code** and **Qty**, and the program looks up the other information in the inventory. Verify that the information is correct and make sure that you have enough in inventory to satisfy the order. Continue entering items until you are through. Pressing **<Return>** indicates that you are finished; **sale** then calculates the totals for the order, including tax.

This program uses the commands **cursor** and **termput** to maneuver about the screen.

See Also

cursor, invoice, postar, termput

schema — /rdb Command

Print a table's schema
schema *tableorlist ...*

The command **schema** prints the schema, or a data-base dictionary, for each *tableorlist*.

Other data-base systems require that you define a schema prior to creating a table. In **/rdb**, you can derive a schema from an existing table or tables.

Example

The following command combines the schemas for tables **inventory** and **maillist**. The example pipes the concatenated schema through the command **justify** to "pretty up" the output:

```
schema inventory maillist | justify
```

This produces the following:

```
Table           Field   Name
---------       -----   -----------
inventory       1       Item
inventory       2       Amount
inventory       3       Cost
inventory       4       Value
inventory       5       Description
maillist        1       Number
maillist        2       Name
maillist        3       Company
maillist        4       Street
maillist        5       City
maillist        6       State
maillist        7       ZIP
maillist        8       Phone
```

Because **schema** is a shell script, you can modify it to give widths of columns, data types, and so on.

See Also

datatype, precision, width

screen — /rdb Command

Convert a form into a screen-input shell program
screen < *form* > *shellprogram*

The command **screen** reads *form* and writes a COHERENT shell program that paints the screen and reads a user's input. Because the output of **screen** is a shell script, you can edit it to validate data, read data and write it onto the screen, and so on. *form* must be in the same format as described in the manual entry for the command **report**.

screen writes a simple shell script. The script works "as is" to read data typed onto the screen and to write a standard table. It uses the **/rdb** command **cursor** to move to each input field on the screen.

A script output by **screen** can be used in many ways. You can use it as a form, to enter data into a new table. You can edit the script to have it append its output into an existing table, to pass its data via a pipe to another command, or to read data from a table and display them on the screen. Commands can be edited into the screen-form script. You can add calls to the **/rdb** command **validate**, to confirm that the user has added data that are at least sensible. You can even embed the script within another shell script, to perform screen entry for it.

Setup Steps

To create a screen-entry form, do the following:

1. Use your text editor to write the screen-form file.

2. Run **screen** to read the form file. Redirect its output into the file in which you wish to store the form's executable script.

3. Edit the shell script to meet your requirements, such as adding validation, record locking, and so forth.

4. Run the executable shell script to input data and write a new table.

Example

A simple form looks like this:

```
                        Mail List Entry Form
                    <!date!>

Name  : <Name>            Company: <Company>
Street: <Street>
City  : <City>            State  : <State>       ZIP: <ZIP>
Phone : <Phone>
```

Everything is written to the screen except what is between angle brackets "<>", which are variables to be input. Exclamation marks within angle brackets (e.g., **<!date!>**) mark commands to execute when the program is running.

When **screen** reads the above form and compiles it, it writes the following shell script:

```
: paint crt screen
exec 1>/dev/tty
cat <<SCREEN
${CLEAR}                          Mail List Entry Form
```

LEXICON

```
Name  :                    Company:
Street:
City  :                    State  :              ZIP:
Phone :

SCREEN

: read user input
cursor  1 23 ; date;
cursor  3  8 ; read Name;
cursor  3 36 ; read Company;
cursor  4  8 ; read Street;
cursor  5  8 ; read City;
cursor  5 36 ; read State;
cursor  5 55 ; read ZIP;
cursor  6  8 ; read Phone;

: output table head
exec 1>&2
echo "Name  Company  Street  City  State  ZIP  Phone"
echo "----  -------  ------  ----  -----  ---  -----"
: append row
echo "$Name  $Company  $Street  $City  $State  $ZIP  $Phone"
```

This shell progam is in three parts, each introduced by a comment:

paint crt screen
This section clears the screen, then paints the form onto it.

read user input
This section moves the cursor to each input field, then waits for the user to enter a string of characters. Each field is linked to a column; that is, what the user types in that field will be written into the column to which the field is linked.

output table head
The final section writes the new table in standard **/rdb** format; that is, a column header followed by the row of data entered by the user.

To let you use the screen program in a pipe, the generated commands send the interactive output to the screen through the standard output temporarily tied to **/dev/tty**, and to the output table through the standard-error device. To accomodate COHERENT's implementation of **exec**, you must tie the standard error to the standard output when you run the script screen has created. For example

```
mail.s > mail.t 2>&1
```

The following shows the form compiled above, with example data:

```
                        Mail List Entry Form
                    Wed Sep  4 14:09:30 EDT 1987

Name  : Hi Ho              Company:
Street: 1st St.
City  : LA                 State  : CA           ZIP: 90024
Phone : 213-555-1212
```

The above input creates the following table:

```
Name    Company Street  City    State   ZIP     Phone
----    ------- ------  ----    -----   ---     -----
Hi Ho   HH Inc. 1st St. LA      CA      90024   213-555-1212
```

You can redirect output into a file or pipe. Here, it was directed into table **mail.t**. You can use this table as its stands alone, or you can append it to another table by by piping it to the **append** command. For example:

```
mail.s 2>&1 | append mail.t
```

See Also

cursor, lock, unlock, validate
COHERENT Lexicon: **cat, echo, exec, ksh, read, sh**

search — /rdb Command

Search a table
search [-m[bhirs]] [-hsnx +x] [-l [2> *location***]]** *tableorlist* **[***keycolumn ...***] [<** *keytable* **]**

The command **search** finds a row quickly. It searches *tableorlist* for each row in which the value *keycolumn* matches one or keys you supply. The key or keys can either be entered interactively, or read from *keytable*. You must use the **/rdb** command **index** to index *tableorlist* before you can use **search** to search it.

By default, a key's value must match exactly the string within *keycolumn*. The options **-x** and **+x** modify this behavior; they are described below.

You can use **search** in any of several different ways. Because it reads the key values from the standard input, you can input the keys by typing them interactively at the terminal, by diverting the contents of *keytable* into it, or by piping the output of a command into it. Each method is demonstrated in the examples given below. Therefore, it is easy to use **search** on a command line or in a shell script. For example, the command **domain** is a shell script that uses **search** to speed its searches. If you use a pipe to pass data to **search**, think of a stream of keys going into it and a stream of matching rows coming out.

search can use a number of different methods of searching. There are advantages and disadvantages to each; different sizes and structures of tables are best suited to different methods of searching. If you are not sure which method to use, you can try them all on your *tableorlist*. Use the COHERENT command **time** to find how long each method takes, then pick the one that runs the fastest. Another **/rdb** command, **timesearch**, runs each form of **search** on the sample **unixtable**, which is kept in directory **$RDB/demo**. Because **timesearch** is a shell script, you can modify it to time the different methods for your own file.

The behavior of **search** depends upon which method of searching you choose. With some methods, it first searches another table (called the "secondary index file") built by the command **index**. One method, sequential, searches every row in *tableorlist*.

If you change *tableorlist*, such as when you add a new row or delete an old one, you must reindex it. If you develop application programs, you can have them update secondary index files to handle dynamic file-processing.

Methods of Searching

The following describes the different methods of searching a table. Each method is introduced by the command-line option with which you can invoke it:

-m Perform a sequential search. This is the default method.

LEXICON

-mb Binary search. This first looks at the middle record of the file. If the key column's value is greater than that of the key that **search** is seeking, then **search** look at the record one quarter of the way through the file. **search** continues to bifurcate the table until it finds the row, or finds that the row is not there.

The method of searching requires that you first sort the table on the key column.

A binary search takes log(2)*n* accesses:

Accesses	*No. of Records*
10	1 thousand
20	1 million
30	1 billion
40	1 trillion

Binary searching is a good method for searching tables that must remain sorted on the key column.

-mh Hash searching. A hash search computes an address from the key and looks into a secondary file at that address. If the key matches, it retrieves the address of the row you want in the main file and then retrieves it. If the address **search** hashes to is empty (i.e., has a value of zero), then the key does not match and the row is not in the file.

Occasionally, another key is there that happens to hash to the same address; this is called a *collision*. When a collision occurs, **search** will keep looking at subsequent entries in the secondary table until it either finds a match or finds an empty or deleted (-1) entry.

A hash search usually takes two disk accesses when the hash table is half full. For this reason, **/rdb** sets up hash tables that are twice as large as the number of records being hashed. As the data table grows, it must be reindexed to keep the hash speed up.

Hashing is a good method for tables that are static or change only rarely.

-mi Inverted or indexed searching. An inverted search uses a secondary file that must have been created by **index**. This index table has two columns: the key column from the data table, sorted to allow binary searching; and the offset of each key-column value within the data table. When **search** finds the key in the index table's key column, it then uses value from the address column to retrieve the row from the data table in a single disk read. The inverted method takes a little longer than binary, but sorting the index file is much faster than sorting the data table.

-mr Record number searching. A record search simply retrieves the row requested. For example, if you enter '5', you will get the fifth row in the data table.

To retrieve a record by its row number, **search** reads a secondary file that that contains the offset for each record in the data table. The record method is a faster way to retrieve a record; however, you must know the number of the record you want, and you must either keep the data table in that order, or reindex it. You can use the command **replace** to move a row to the of its table and update the secondary index to point to the row's new location.

-ms Sequential or linear searching. A sequential search examines all records in the data table in the order in which they appear. It is not a fast-access method, although it may still be faster than that used by the command **row**. It is provided as a benchmark for timing other methods. If your file is not big enough to make the other access methods worthwhile, just use sequential **search** or **row** and save yourself the overhead of building and maintaining an index.

A sequential search can search multiple key columns, and and can search any column. The sequential method is better than the COHERENT command **grep** because it looks at one column, whereas **grep** scans the entire row for a pattern match.

LEXICON

Other Options

search also recognizes the following command-line options:

-h Do not print the table's head line. By default, **search** prints the head line of the table it searches as well as all rows it finds. The **-h** option suppresses the printing of the head line and prints only the rows you are seeking.

-l Output the location of the record in the data table and the location of the index record in the index table. These locations is in the form *from*to*x from*xto , and can be piped to the commands **replace**, **lock**, and **unlock**. **search** writes these data to the standard-error device, to separate them from the table it writes to the standard output.

Because **search** is a shell program, it demonstrates how you can do this in your own programs. Turning on this option also turns on the **-s** option because **/rdb** allows you to lock or replace one record at a time. Therefore, all keys in your file must be unique, or some will never be found.

-s Search for only one row. This speeds the search because **search** quits when it finds the first match and continue searching the file.

-x Partial match and case insensitive, like the Berkeley command **look**.

+x Exact match, including blank characters. The default matching ignores blank characters.

Examples

Assume you have a file called **inventory**, which appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
1        3      50      150     rubber gloves
2       100     5       500     test tubes
3       5       80      400     clamps
4       23      19      437     plates
5       99      24      2376    cleaning cloth
6       89      147     13083   bunsen burners
7       5       175     875     scales
```

If you had indexed the column **Description**, then you have several ways to find rows:

1. Interactively type key values. **search** immediately prints the rows it finds. For example, type:

```
search -mh inventory Description
```

search responds with the head line of the inventory table. Now you can type names, in this example **scales**; if **search** finds a match, it returns the row that contains the key.

2. Divert a table or list file to **search**. You can create a table with a column of key values; for example:

```
Description
-----------
scales
clamps
```

The following command line diverts this table into **search**

```
search -mh inventory Description < description
```

and produces the following output:

LEXICON

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
7       5       175     875     scales
3       5       80      400     clamps
```

3. Input keys through a COHERENT pipe. For example, the command

```
column Description < sometable | search -mh inventory | column Value
```

returns the following:

```
Value
-----
  875
  400
```

If you wish to justify a table like this, just pipe the output of **search** to the command **justify**.

This lets you use the COHERENT command **echo** to pipe a key value:

```
echo scales | search -mh inventory Description
```

which returns the following to the standard output:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
7       5       175     875     scales
```

In shell programming, you can capture the output of a program or a pipe by putting it within two grave (‘) symbols. In a shell script, you can set an environmental variable like this:

```
RECORD=‘echo key | search -mh -h bigtable column‘
```

The command **echo $RECORD** displays the data diverted into this variable; for example:

```
 7       5       175      875    scales
```

Now the **RECORD** environmental variable is equal to the matching row in the table. Note the **-h** is important to keep the head line from being written into **RECORD**.

Search vs. Select

search is important for application programs written in shell scripts, the C programming language, or other code. It can be used interactively, but the command **row** is much more powerful because it can seek combinations of logical conditions, regular expressions, and conditions such as *greater than,* whereas **search** needs a key (either exact or partial). But for pulling a row out of a big table, **search** can speed things up enough to be preferred. In summary, **search** is for high-speed access with keys, whereas **row** should be used to retrieve rows that match complex logical conditions and regular expressions.

See Also

append, column, delete, index, replace, row, seek, update
COHERENT Lexicon: **egrep, grep, look, time**

searchtree — /rdb Command

Seek a string node in tree table
searchtree [-m[bhirs]] *table root goal column1 column2*

The command **searchtree** searches *table* to see if string *root* in *column1* is associated with *goal* in *column2*. It returns TRUE to the shell if it find that *root* is associated with *goal*, or FALSE if it is not.

goal may be within *table*, in which case the COHERENT command **grep** can also find it; **searchtree**, however, will also see if *goal* be reached by a path from *root*.

The options **-m[bhirs]** name the fast-access method to use. The default is sequential (**-ms**). See the entry for the command **search** for a full description of each method of searching.

searchtree is a breadth-first search that uses the **search** fast-access methods to find all of the children at each level of a tree. This is the kind of search often done in artificial intelligence programs. However, **searchtree** stores its data in a temporary file rather than in memory, so that it will work on very large data bases.

Because **searchtree** is a shell script, you can modify it to serve many kinds of searches. For example, you can add a message like *yes/no* or *true/false*. You can also add shell code to print out a path. You might want to use heuristics to guide the search at each level, and side effects to update data or print messages along the way.

This kind of searching is also used by the PROLOG language. **searchtree**, however, differs from PROLOG in two significant respects. First, PROLOG must have its data base in memory, which limits the size of the database it can handle. Second, PROLOG uses depth-first searching, which requires it to backtrack.

Example

Consider the tree represented by table **isa**, as follows:

```
Name    Isa
----    ---
barbara human
bill    human
evan    human
rod     human
human   primate
primate mammal
cat     mammal
dog     mammal
mammal  animal
bird    animal
fish    animal
animal  lifeform
plant   lifeform
```

Remember that all trees, in fact all graphs, can be listed as tables when the nodes (vertices) that are connected by a line (arcs) that is on each row. A directional graph is handled by having one column be **From** and another be **To**.

The following command uses **searchtree** to examine table **isa** to see if entity **rod** in column **Name** is of type **lifeform** as given in column **Isa**:

```
searchtree isa rod lifeform Name Isa ; echo $?
```

The command **echo $?** prints the value returned by **searchtree**. In this case, you see '0' (or FALSE), because **rod** is not a **lifeform**; rather it (or he) is a **human**.

Because **searchtree** returns only a status code, you can use it as a test condition. Your shell program could have a statement like this:

LEXICON

```
prompt "What is your name: "
read NAME
if searchtree isa $NAME lifeform Name Isa
then
        echo "So, I am talking to a lifeform."
else
        echo "So, I am talking to another machine."
fi
```

The following gives an sample exchange with this script; text in **bold** gives what the user types:

```
What is your name: rod
So, I am talking to a lifeform.
```

The result of the search of the tree was tested by the **if** statement, and used to decide which message to print.

See Also

search

Notes

If you wish to experiment with PROLOG, volume 2 of **COHware** includes the source code to a PROLOG interpreter.

see — /rdb Command

Display nonprinting as well as printing characters
see < *tableorlist*

The command **see** displays the contents of *tableorlist*. Unlike the COHERENT command **cat**, **see** represents all nonprinting characters by converting them to the form **^***some-printing-character*. It is essentially the same as the Berkeley UNIX commands **see** or **cat -v**.

see is very useful for seeing if you have placed tabs into your tables or lists. Tabs print out as **^I** because they are also obtained by typing **<ctrl-I>**. **see** is also helpful for finding nonprinting characters.

Example

To see the if the tabs are placed correctly in table **badtable**, type:

```
see < badtable
```

You'll see something like:

```
Date^IAccount^IDebit^ICredit^IDescription$
------^I-------^I-------^I-------^I-------------------------$
820102^I101^I25000^I^Icash from loan$
820102^I211.1^I^I25000^Iloan number #378-14 Bank Amerigold$
820103^I150.1   10000^I^Itest equipment from Zarkoff$
820103^I101^I^I5000^Icash payment$
820103^I211.2^I^I5000^Inote payable to Zarkoff Equipment$
820104^I130^I30000^I^Iinventory - parts from CCPSC$
820104^I201.1^I^I15000^Iaccounts payable to CCPSC$
820104^I101^I^I15000^Icash payment to CCPSC for parts$
```

See Also

COHERENT Lexicon: **ASCII**

Notes

see assumes ASCII characters with a value between 0 and 127. **see** adds 128 to negative characters and to characters with values greater than 127. It then adds the value of ASCII 'A' to characters with a value less than that of a blank character (ASCII 32) so they map to the upper-case characters.

For example, NUL is printed as **^@**, a tab character as **^I**, and a form-feed character as **^L**.

seek — /rdb Command

Return the beginning and ending offset of a row
seek [-m[bihrs]] [-o *outfile***]** *tableorlist* **[***keycol...***] <** *file*

The command **seek** returns the offsets of the beginning and end of a record in *tableorlist*. It uses the command **search** with its location option **-l**.

The offset and size integers returned can be used by the **replace** to write a new record. They are also used by the commands **lock** and **unlock** to set and remove record locks.

Options

seek recognizes the following options:

-m[bhirs]
Fast access methods. See the manual entry for the command **search** for a full description of various search options and how you can invoke each.

-o Write the found record into **outfile**.

Example

Let's use **seek** on the table **inventory**, which is as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

The following command invokes **seek** to find row 5:

```
echo 5 | seek -mb inventory Item
```

This writes the following to the standard output:

```
207     245     0       8
```

This means that the record is 207 bytes into the file and that it is 38 bytes long, ending at byte 245.

The next example extracts a row from **inventory** blanks it out, then writes the row back into **inventory** on top of the original row. First, extract the row:

```
LOCATION='echo 5 | seek -mb -o tmp inventory Item'
```

The location of the selected record is stored in environmental variable **LOCATION**. The option **-o tmp** writes the record with key equal to 5 into table **tmp**, which appears as follows:

LEXICON

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   5        99    24     2376   cleaning cloth
```

The next command blanks the contents of **tmp**, then writes the blanked row back into **inventory** at the location stored earlier at **LOCATION**:

```
blank < tmp | replace inventory $LOCATION
```

inventory now appears as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates

   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

See Also

index, lock, replace, search, unlock

select — /rdb Command

Output selected rows
select '*column condition column-or-value*' **<** *table*

The command **select** writes a new file that consists of the rows of *table* that match the expression *condition* in *column*.

select is a link to the command **row**. It is included for those users who prefer the name **select** (which is a relational theory name) to **row**.

See Also

row

sorttable — /rdb Command

Sort a table by one or more columns
sorttable [*sort options***] [***columnname ...***] <** *tableorlist*

The command **sorttable** uses the COHERENT command **sort** to sort *tableorlist*. Unlike **sort**, it knows about table head lines and column names.

With **sorttable** you use the column names rather than column numbers, as in **sort**. **sorttable** also tells **sort** to use tabs as field separators, instead of white space (i.e., blanks and tabs).

sorttable completely sorts by the first column you name. If there are any duplicate values in that column, it goes to the second column and sort on those values. It continues across the named columns until it either finds a column that consists of unique values, or it runs out of columns. For example, if you sort a ledger on columns **Account** and **Date**, **sorttable** sorts the entire table on column **Account**, and sorts by **Date** within each account.

If no columns are named, **sorttable** sorts *tableorlist* by all of its columns from left to right.

Options

sorttable passes all the options you give it to **sort**. See the entry for **sort** in the COHERENT manual's Lexicon for details on this command's options.

Example

Consider table **inventory**, which is structured as follows:

```
Item#   Amount   Cost    Value  Description
------- ------- ------- ------- ---------------
      1       3    5.00   15.00 rubber gloves
      2     100    0.50   50.00 test tubes
      3       5    8.00   40.00 clamps
      4      23    1.98   45.54 plates
      5      99    2.45  242.55 cleaning cloth
      6      89   14.75 1312.75 bunsen burners
      7       5  175.00  875.00 scales
```

To see **inventory** sorted with the highest **Cost** items on top, type:

```
sorttable -n -r Cost < inventory
```

This produces:

```
Item#   Amount   Cost    Value  Description
------- ------- ------- ------- ---------------
      7       5  175.00  875.00 scales
      6      89   14.75 1312.75 bunsen burners
      3       5    8.00   40.00 clamps
      1       3    5.00   15.00 rubber gloves
      5      99    2.45  242.55 cleaning cloth
      4      23    1.98   45.54 plates
      2     100    0.50   50.00 test tubes
```

Note that **Cost** is in reverse order (the **-r** option) and that it was treated as a number (**-n** option) instead of a character string.

If you use **sorttable** frequently, you should carefully study the COHERENT Lexicon entry for **sort**. For example, you can control which columns **sorttable** sort on. If you name several columns, **sort** sees each column's numbers proceeded by a '+'. The '+' means "start sorting on this column", the hyphen '-' means "stop on the next column". To see this, use the **-D** (debug) option for **sorttable.** (By the way, it is because **sort** has a **-d** option that we use a capital **-D** for the debug option for the **/rdb** commands). For example, typing

```
sorttable -D Cost Description < inventory
```

yields:

```
Item    Onhand   Cost    Value  Description
------- ------- ------- ------- ---------------
sorttable: command executed: sort -t'   '   +2 -3 +4 -5
2       100     1       100     test tubes
4       23      2       51      plates
5       99      3       297     cleaning cloth
1       3       5       15      rubber gloves
3       5       8       40      clamps
6       89      18      1602    bunsen burners
7       5       175     875     scales
```

The line beginning with **sort** is the line passed to COHERENT. Note the **+1**, **-2**, and so on. That tells **sort** to sort on columns 2 and 4 (where the first column is 0). Thus, **sorttable** correctly turns the column names you type into a **sort** command by looking up the column names in the table head line and converting them to column numbers for the COHERENT command **sort**.

People are often confused by **sort** because it does not behave the way one would expect. Study its entry in the COHERENT Lexicon.

LEXICON

See Also

COHERENT Lexicon: **sort**

soundex — /rdb Command

Convert a name into soundex code
soundex < *file* **>** *file***.x ; index -mb** *file.x* **Soundex**

The command **soundex** creates a secondary index to *file* based on Knuth's soundex code. It is used by the **/rdb** command **like** to find a name that sounds like another name.

The output must be written into file *file***.x**; then run the command **index** on column **Soundex** within *table***.x** to ensure a high-speed search.

Example

Consider table **name**, which has many similar names in it:

```
Name
----
Abraham
Abraham
Ackerman
Actor
Adams
Adams
Adams
Adams
...
```

Use **soundex** to create the secondary index file, as follows:

```
soundex < name > name.x ; index -mb name.x Soundex
```

The soundex'd index file appears as follows:

```
Name    Soundex
----    -------
Avila   A140
Avila   A140
Avila   A140
Abraham A160
Abraham A160
Aissen  A220
Ajeska  A220
Augustine       A223
...
```

Now you can use the **like** command to find a name. For example, the command

```
like Manis name
```

produces something like:

```
Name
----
Mann
Mann
Means
Mink
Monahan
Moniz
Mooney
Mooney
Munoz
Munoz
```

Or the command

```
like Schaffer name
```

produces something like:

```
Name
----
Schaeffer
Schaffer
Schiffrin
```

See Also

like

splittable — /rdb Command

Divide a table horizontally
splittable [-n] *table*

The command **splittable** uses the COHERENT command **split** to split *table* horizontally into several smaller tables, each with its own head line. *n* is the number of rows to write into each new table, not counting the two head lines.

This command is most useful for report writing because it can split tables into page-sized chunks.

Like **split**, **splittable** creates file names that end with **aa**, **ab**, **ac**, and so on. This give 676 (26 x 26) possible output files.

Example

Here is an example using a very short table, called **inventory**:

```
Item#   Amount  Cost    Value   Description
-----   ------  ----    -----   --------------
    1        3    50      150   rubber gloves
    2      100     5      500   test tubes
    3        5    80      400   clamps
    4       23    19      437   plates
    5       99    24     2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

The command

```
splittable -4 inventory
```

splits **inventory** into two tables, one with the first four lines of **inventory** and the other with the last three. Each new table also has the same head line as **table**.

LEXICON

The command

```
ls inventory*
```

shows us that the current directory now contains files:

```
inventory
inventoryaa
inventoryab
```

The two new tables are named, respectively, **inventoryaa** and **inventoryab**.

See Also

COHERENT Lexicon: **ls, split**

Notes

Because a COHERENT file can have no more than 14 characters in its name, **splittable** will not work correctly with tables whose names are longer than 12 characters. *Caveat utilitor.*

substitute — /rdb Command

Replace old string with new string
substitute *oldstring newstring file ...*

The command **substitute** replaces *oldstring* with *newstring* throughout each *file*. Because it uses the COHERENT command **sed**, be sure to protect with backslash '\' all **sed**'s special characters: '*', '^', '$', and '\'.

Example

Again, consider table **inventory**, as follows:

```
Item    Amount  Cost    Value   Description
-----   ------  ----    -----   -------------
    1        3    50      150   rubber gloves
    2      100     5      500   test tubes
    3        5    80      400   clamps
    4       23    19      437   plates
    5       99    24     2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

To replace string **rubber** with **latex** throughout **inventory**, type:

```
substitute 'rubber' 'latex' file
```

inventory now appears as follows:

```
Item    Amount  Cost    Value   Description
-----   ------  ----    -----   -------------
    1        3    50      150   latex gloves
    2      100     5      500   test tubes
    3        5    80      400   clamps
    4       23    19      437   plates
    5       99    24     2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

See Also

COHERENT Lexicon: **sed**

subtotal — /rdb Command

Output subtotals of columns in a table
subtotal [-l] [*break-column column ...* **] <** *table*

The command **subtotal** computes subtotals in each numeric *column* within *table*.

subtotal uses *break-column* to mark where computation should begin and end: as long as a value in *break-column* remains the same, the values in the other columns are accumulated. When the value in the *break-column* changes, **subtotal** prints the value or values accumulated since the last break.

If no columns are specified on the command line, **splittable** uses the first, or leftmost, column of the table as the break column, and computes subtotals for all other columns.

The option **-l** prints whole table, not just the subtotals.

Example

There are two forms of output. One produces only the subtotals. If you wanted to see the daily subtotals of table **journal**, type:

```
subtotal Date Debit Credit < journal
```

This produces something like the following:

```
Date    Account Debit   Credit  Description
------  ------- -----   ------  ----------------------------------
891222          25000    25000
891223          10000    10000
891224          30000    30000
```

Note that columns **Description** and **Account** are blank, because they contain non-numeric data.

The **-l** prints the entire table, as well as the subtotals. For example, the command

```
subtotal -l Date Debit Credit < journal
```

produces:

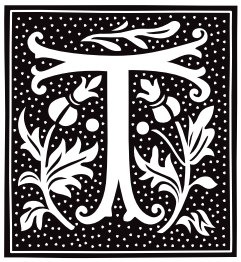
```
Date    Account Debit   Credit  Description
------  ------- -----   ------  ----------------------------------
891222  101     25000           cash from loan
891222  211.1            25000  loan number #378-14 Bank Amerigold
------  ------- -----   ------  ----------------------------------
891222          25000    25000

891223  150.1   10000           test equipment from Zarkoff
891223  101               5000  cash payment
891223  211.2             5000  note payable to Zarkoff Equipment
------  ------- -----   ------  ----------------------------------
891223          10000    10000

891224  130     30000           inventory - parts from CCPSC
891224  201.1            15000  accounts payable to CCPSC
891224  101              15000  cash payment to CCPSC for parts
------  ------- -----   ------  ----------------------------------
891224          30000    30000
```

This lets you see each row as well as the subtotals. This could help you find out when you got out of balance.

See Also

compute, maximum, mean, minimum

LEXICON



tableorlist — /rdb Command

Report whether a file has table or list format
tableorlist [*tableorlist ...***] [<** *tableorlist***]**

The command **tableorlist** checks whether *tableorlist* is a table or a list. If it is a list (that is, its first character is a newline), then **tableorlist** prints the string "list" and returns status code 1 to the shell; otherwise, it prints the string "table" and returns status code 0.

Example

To check if **inventory** is a table or list, type:

```
tableorlist inventory
```

This prints

```
table
```

on the standard output. To check what status value **tableorlist** returned, type:

```
echo $?
```

You'll see:

```
0
```

See Also

listtotable, tabletolist
COHERENT Lexicon: **file**

tabletofact — /rdb Command

Converts a table to PROLOG fact-file format
tabletofact *table ...* **>** *prologfactfile*

The command **tabletofact** converts each *table* into a file of facts in predicate calculus functor form, which can be input to a PROLOG interpreter.

Facts

PROLOG facts are in the form *relationship(subject,object)*. For example, the fact

```
parent(sally,fred)
```

means that "sally is the parent of fred".

PROLOG is a programming language whose name is a contraction of the phrase *programming in logic*. Its input consists of *facts*, *rules*, and *questions*. If you build a data base of facts and rules, you can ask questions and get answers. If a question you ask does not have a corresponding fact in the data base, PROLOG looks through its rule file to see if other facts might logically deduce the fact requested.

LEXICON

Example

Suppose we have these simple table **parent**

```
Parent  Child
------- -------
Sally   Fred
Mike    Fred
Fred    Jane
```

the simple table **female**

```
Name
----
Jane
Sally
```

and the simple table **male**:

```
Name
----
Fred
Mike
```

The command

```
tabletofact parent female male > fact
```

produces the following facts:

```
parent(Sally,Fred).
parent(Mike,Fred).
parent(Fred,Jane).
female(Jane).
female(Sally).
male(Fred).
male(Mike).
```

As you can see, the relationship is set by the name of the table, and the subject and object (if any) are set by the data in each row.

PROLOG can now use its command **consult** to read this file. If PROLOG had rules about father, mother, grandparent, and so on, then it could answer questions like: **mother(sally,fred)?** That fact is not in the data base, but can be deduced from rules like:

```
mother(X,Y) if parent(X,Y) and female(X)
```

See Also

tabletorule

Notes

If you wish to experiment with PROLOG, volume 2 of **COHware** includes the source code for a PROLOG interpreter.

tabletofix — /rdb Command

Convert /rdb table format to fixed-length format
tabletofix *column*=*n*[**r**] **...** **<** *tableorlist*

The command **tabletofix** converts *tableorlist* from variable-length **/rdb** format to the fixed-length format used by many other data-base management systems.

tabletofix works by either padding or truncating each *column* to make it *n* characters wide. If the

letter *r* is used with the width specification, **tabletofix** right-justifies the *column* in question — that is, it pads the column with leading blanks, assuming any are necessary. Otherwise, the *column* is left justified — that is, it pads the column with trailing blanks.

tabletofix also throws away the head lines, as well as all tab characters and blank records.

Example

Consider table **journal**, which looks like this when viewed with the command **see**:

```
Date^IAccount^IDebit^ICredit^IDescription$
----^I-------^I ----^I -----^I-----------$
820102^I101^I25000^I^Icash from loan$
820102^I211.1^I^I25000^Iloan number #378-14 Bank Amerigold$
820103^I150.1^I10000^I^Itest equipment from Zarkoff$
820103^I101^I5000^I^Icash payment$
820103^I211.2^I^I5000^Inote payable to Zarkoff Equipment$
820104^I130^I30000^I^Iinventory - parts from CCPSC$
```

The symbol **^I** indicate a tab, and the symbol **$** indicates a newline.

Now, let's generate a fixed-length field data base from **journal**. We'll use the fields **Date**, **Account**, **Credit**, and **Debit**. We'll give the first two fields six bytes each; and we'll give the last two fields seven bytes each, right justified:

```
tabletofix Date=6 Account=6 Debit=7r Credit=7r < journal > newjournal
```

This commands writes its output into file **newjournal**, which appears as follows:

```
820102101     25000
820102211.1          25000
820103150.1   10000
820103101              5000
820103211.2            5000
820104130     30000
```

See Also

column, fixtotable, see

tabletolist — /rdb Command

Convert a table to list format
tabletolist [-l] < *table*

The command **tabletolist** converts *table* to list format.

Option **-l** tells **tabletolist** not to write a newline at the beginning of the new list file. This is for compatibility with the older list format and for other uses. Note that most **/rdb** commands cannot read a list file if it lacks the initial newline.

Example

Consider **mailtable**, which has the following format:

```
Number  Name    Company Street  City    State   ZIP     Phone
------- ------- ------- ------- ------- ------- -------
1       Ronald McDonald McDonald's      123 Mac Attack  Memphis
TENN    30000   (111) 222-3333
2       Chiquita  Banana        United Brands   Uno Avenito De La
Revolution      San Jose        Costa Rica      123456789       1234
```

As you can see, the information in each row is often too wide for the screen; these data would be better stored in a list-format file. To convert **mailtable** to list format, type the following:

LEXICON

```
tabletolist < mailtable
```

This writes the following to the standard output:

```
<newline>
Number  1
Name    Ronald McDonald
Company McDonald's
Street  123 Mac Attack
City    Memphis
State   TENN
ZIP     30000
Phone   (111) 222-3333

Number  2
Name    Chiquita  Banana
Company United Brands
Street  Uno Avenido de la Reforma
City    San Jose
State   Costa Rica
ZIP     123456789
Phone   1234
 ...
```

See Also

listtotable

tabletom4 — /rdb Command

Convert a table to m4 define-file format
tabletom4 < *table* **>** *definefile*

The command **tabletom4** converts a two-column table into a file of define statements that can be input to the COHERENT macro processor **m4**. Define macros are in the form *define(old,new).*

m4 can do many things, including word-for-word substitution. It can be used for language translation (in a stiff, inflexible way) and for other conversions of data bases.

The *definefile* that **tabletom4** produces is required by the **/rdb** command **translate**. Only words that have a corresponding translation in the second column will be put into *definefile*. In this way, you can use the file before you have all of the translations entered.

Example

Consider the simple translation table **ed.t**:

```
English Deutch
------- -------
I       Ich
love    liebe
you     dich
widgit
```

The following command converts **ed.t** to m4 format and writes its output into file **ed**:

```
tabletom4 < ed.t > ed
```

ed now contains the following:

```
define(I,Ich)
define(love,liebe)
define(you,dich)
```

Note that **widgit** was not converted because it did not have a translation string in the second column of the table. **ed** can now be used by **m4** or **translate**. For example, if file **text** contains the following:

```
I love you.
```

The command

```
m4 ed text
```

writes

```
Ich liebe dich.
```

onto the standard output.

See Also

translate
COHERENT Tutorials: **Introduction to the m4 Macro Processor**
COHERENT Lexicon: **m4**

tabletorule — /rdb Command

Convert a table to PROLOG rule-file format
tabletorule *table ...* **>** *prologrulefile*

The command **tabletorule** converts each *table* into a file of rules in predicate-calculus implication form that can be input to a PROLOG interpreter.

PROLOG rules are in the form:

relationship(X,Y) :- class(X), relationship(X,Y).

For example

```
mother(X,Y) :- female(X), parent(X,Y).
```

means that "X is the mother of Y, if X is female and X is the parent of Y."

PROLOG is a programming language used for artificial-intelligence applications. Its name is a contraction of the phrase *programming in logic.* Input to PROLOG consists of *facts*, *rules*, and *questions*. If you build a data base of facts and rules, you can ask questions and get answers. If a question you ask does not have a corresponding fact in the data base, PROLOG will look through its rule file to see if other facts might logically deduce the fact requested.

Example

Suppose we have the simple table **parent**

```
Parent  Child
------- -------
Sally   Fred
Mike    Fred
Fred    Jane
```

the simple table **female**

```
Name
----
Jane
Sally
```

LEXICON

the simple table **male**

```
Name
----
Fred
Mike
```

and the simple table **ruletable**:

```
True          If
-----------   ------------------------
mother(X,Y)   female(X), parent(X,Y)
```

The command

```
tabletorule ruletable > rule
```

writes the following into file **rule**:

```
mother(X,Y) :- female(X), parent(X,Y).
```

Now PROLOG can use its **consult** command to read **rule**. It could then answer questions like:

```
mother(sally,fred)?
```

See Also

tabletofact

Notes

If you wish to experiment with PROLOG, volume 2 of **COHware** includes the source code for a PROLOG interpreter.

tabletosed — /rdb Command

Convert table format to sed file format
tabletosed < *table* **>** *sedfile*

The command **tabletosed** converts a two-column *table* into a file of **sed** statements that can be input to the **sed** stream editor. **sed** macros are in the form:

```
s/old/new/g
```

sed can do many things, including word-for-word substitution. *sedfile* produced is used by the **/rdb** command **translate**. It can be used for language translation (in a stiff, inflexible way) and for other conversions of data bases.

Example

Suppose we have a simple translation table, called **ed.t**

```
English Deutch
------- -------
I       Ich
love    liebe
you     dich
widgit
```

The following command turns **ed.t** into a file of **sed** editing instructions, called **ed.sed.1**:

```
tabletosed < ed.t > ed.sed.1
```

ed.sed.1 appears as follows:

```
s/I/Ich/g
s/love/liebe/g
s/you/dich/g
```

We can now use **ed.sed.1** with **sed** or **translate**. For example, if file **text** contains the sentence

```
I love you.
```

then typing

```
sed -f ed.sed.1 text
```

prints the following on the standard output:

```
Ich liebe dich.
```

You can also use this command to construct form letters.

See Also

translate
COHERENT Tutorials: **Introduction to the sed Stream Editor**
COHERENT Lexicon: **sed**

tabletostruct — /rdb Command

Convert table to C-language struct declaration
tabletostruct *tag name* **<** *table* **>** *table***.h**

The command **tabletostruct** converts a standard **/rdb** table into a C-language **struct** declaration. **struct** is the C language's record format.

tabletostruct is useful for writing table-driven C code. The tables used by your programs can be manipulated by the data base and your users, or by nonprogrammer developers (engineers, experts, managers, operators), then converted to a **struct** and compiled into the C code for high-speed access.

tag gives the **struct** tag for declaring other variables. *name* is the **struct** name to reference the structure values.

Please note once code compiled from a table, changing that tables no longer affects the program until you recompile it. Therefore, you gain maximum speed at the price of recompiling the system and losing the flexibility of being able to modify the tables during execution.

Example

Here we start with a simple table, convert it to **struct,** make it a header file, and compile it into a simple program that prints out each field. To begin, consider table **table**, which is structured as followed:

```
A       B       C
-       -       -
1       2       3
```

The following commands converts **table** to a **struct** and writes it into file **table.h**:

```
tabletostruct Table table < table > table.h
```

table.h now holds the following:

LEXICON

```
struct Table
{
char    *A;
char    *B;
char    *C;
}       table [] =
{ "1","2","3" }
;
```

The following gives some C code that prints the contents of the **struct** we call **Table**:

```
#include "table.h"

main ()
{
        printf ("A=%s\n", table[0].A);
        printf ("B=%s\n", table[0].B);
        printf ("C=%s\n", table[0].C);
}
```

When compiled with the command

```
        cc -o printtable printtable.o
```

the newly created command **printtable** yields the following output:

```
A=1
B=2
C=3
```

Note that the data are stored in an array of **struct**s, so we must give a subscript for each row we want.

We can also put the command **tabletostruct** into a **Makefile**, so that the COHERENT command **make** can automate the building of the **struct** and its compilation. For example:

```
printtable:     table.h printtable.o
        cc -o printtable printtable.o

table.h:        table
        tabletostruct Table table < table > table.h
```

This says that the printable program depends upon **table.h** being up to date, and **table.h** depends upon table **table** being up to date. If you had modified **table** since the last compile, **make** reexecutes the command **tabletostruct**.

See Also

listtotable, tabletolist
COHERENT Tutorials: **The C Language, The make Programming Discipline**
COHERENT Lexicon: **array, make, struct**

tabletotbl — /rdb Command

Convert /rdb table format to UNIX tbl/nroff format
tabletotbl < *tablefile* **>** *tblfile*

The command **tabletotbl** converts an **/rdb** table to UNIX **tbl** format. **tbl** is a UNIX program that formats tables for the **nroff/troff** formatting and typesetting programs.

Notes

The COHERENT system's version of **nroff** and **troff** do not yet support **tbl**.

tax — /rdb Command

Compute tax from income and tax table
tax *income* **<** *taxtable*

The command **tax** reads tax-table information from *taxtable* and computes taxes from it. *income* is the income you wish to check. *taxtable* must be in the three-column format shown in the following example.

tax writes to the standard output the computed tax rounded off to the nearest dollar.

Example

The following table, **X83**, holds the federal income tax for single taxpayers for 1983 (Schedule X):

```
income  tax     percent
------  ---     -------
2300    0       11
3400    121     12
4400    251     15
6500    566     15
8500    866     17
10800   1257    19
12900   1656    21
15000   2097    24
18200   2865    28
23500   4349    32
28800   6045    36
34100   7953    40
41500   10913   45
55300   17123   50
```

If your taxable income is $20,000, type the following:

```
tax 20000 < X83
```

taxtable writes the following to the standard output:

```
3369
```

That is, according to the data in **X83**, a single person with an income of $20,000 in 1983 owed the IRS $3,369.

You can also catch the output in a shell variable, like this:

```
TAX=‘tax 20000 < X83‘
```

This initializes environmental variable **TAX** to **3369**. You can now use **TAX** to write the tax into a table. For example:

```
compute "line == 35 { value = $TAX }" < form1040
```

Note that this command uses the quotation mark (") instead of the apostrophe ('), because we want the shell to substitute the value of tax where we have typed $TAX. To quote something within quotation marks ("), you must use apostrophes (').

Notes

Please note that the tax tables included with **/rdb** are for 1987. Mark Williams Company makes no claim as to their accuracy, or their usefulness in computing current taxes. *Caveat utilitor!*

LEXICON

termput — /rdb Command

Get terminal capability from /etc/termcap file
termput *capability*

The command **termput** reads and returns the current capabilities for your terminal, as set in file **/etc/termcap**. A two-letter capability code is followed by a string of characters that, when sent to the terminal screen, turn on or off that capability. This assumes, of course, that your terminal is properly described in **/etc/termcap**.

Capabilities

The capabilities are two-letter codes recognized by **termcap**. Here are some examples:

cl Clear the terminal screen
cm Cursor movement
so Start standout mode (e.g., reverse video)
se End standout mode

Using most of these codes is straightforward: just echo the appropriate code to the terminal to get the effect you want. Cursor movement, however, is more complicated; the cursor-movement code that **termput** returns must be edited to hold the row and column to which you wish to move the cursor. For details, see the manual page for the command **cursor**.

Example

You can embed **termput** in a shell script to help invoke reverse video, blinking, and any other your terminal has defined in **/etc/termcap**.

The first example clears the screen:

```
termput cl
```

The next example puts the screen into reverse video:

```
termput se
```

and the following turns off reverse video:

```
termput so
```

You can use **termput** to initialize environmental variables, which you can then invoke in a shell script. This is much faster than making repeated calls to **termput**. To set up these shell variables, insert the following commands your file **$HOME/.profile**:

```
export AE=‘termput ae‘           # end alternate character set - graphics
export AS=‘termput as‘           # start alternate character set - graphics

export CLEAR=‘termput cl‘        # clear the screen
export CURSOR=‘termput cm‘       # cursor movement - for the cursor command
export MB=‘termput mb‘           # start blinking mode
export ME=‘termput me‘           # turn off all attributes
export SE=‘termput se‘           # end stand out mode
export SO=‘termput so‘           # start stand out mode
export UE=‘termput ue‘           # end underline mode
export US=‘termput us‘           # start underline mode
```

Once this is set up, you can use these environmental variables in your shell program. For example, if you want a message to stand out, bracket it with **$SO** and **$SE**:

```
echo "${SO} LOGOFF NOW ${SE}"
```

See Also

clear, cursor, screen
COHERENT Lexicon: **console, stty, termcap, terminal-independent operations**

testall — /rdb Command

Test all /rdb programs in directory $RDB/demo
testall [> *testall.new* **]**

The command **testall** runs most of the **/rdb** programs with the files in the **$RDB/demo**. You should run **testall** in directory **$RDB/demo**, because the test files are there.

testall writes the output of each example to the standard output. We suggest that you store the output in file **$RDB/demo/testall.old**. Thereafter, you can check if the behavior of a test program has changed, by using the COHERENT command **diff** to compare the output of **testall** with the contents of **$RDB/demo/testall.old**. **/rdb** performs this testing when the **/rdb** database is installed on a computer, and when code is changed.

Example

First go to the demo directory, assuming the path is **/usr/rdb/demo**:

```
cd /usr/rdb/demo
testall > testall.new
diff testall.old testall.new
```

The command **diff** finds and displays any differences between **testfile.old** and **testfile.new**. After careful checking, if differences are correct, type:

```
mv testall.new testall.old
```

or

```
rm testall.new
```

This saves space, because these files are large.

See Also

COHERENT Lexicon: **diff**

testsearch — /rdb Command

Test the fast-access methods
testsearch

The command **testsearch** runs and times the **/rdb** commands **index** and **search**.

You must **cd** to directory **$RDB/demo** before you run **testsearch**, because it uses tables **unixtable** and **unixlist** in that directory. If you wish, you can edit **testsearch** to time the search of a big list or table that you are manipulating.

See Also

testall, timesearch
COHERENT Lexicon: **time**

timesearch — /rdb Command

Time fast-access methods
timesearch [> *timesearch.new* **]**

The command **timesearch** runs and times the **/rdb** commands **index** and **search**. Its purpose is to find the fastest fast-access method for a given situation. Theory often fails when working with access methods; a sequential search by the select command may prove faster than one of the fast-

LEXICON

access methods when the overhead of building the index is included in the evaluation.

timesearch is in directory **$RDB/demo** rather than in **$RDB/bin** because it is written to test just the tables **unixtable** and **unixlist**. Because it is a shell script, you can edit it to test searching on a big table you are manipulating.

Example

The following command test-runs **timesearch**:

```
cd $RDB/demo ; timesearch | more
```

timesearch writes to the standard output the outputs of the **/rdb** commands **index** and **search**, plus the output of the COHERENT command **time** that it uses to time the programs.

See Also

index, search, testall, testsearch
COHERENT Lexicon: **time**
COHERENT tutorials: *Introducing sh, the Bourne Shell*

todaysdate — /rdb Command

Print today's date in YYMMDD format
todaysdate

The command **todaysdate** displays the today's date in the form: *YYMMDD*. Under **/rdb**, this format is called the "computer format." It is the best format for entering data in a column because it sorts correctly.

See Also

computedate, gregorian, julian
COHERENT Lexicon: **date, time, timezone**

total — /rdb Command

Sum a column
total [-l] [column ...] < *table*

The command **total** adds up the values in each *column* in *table*. If no *column* is named on the command line, it totals every column in *table*.

The option **-l** prints the entire table as well as the total for each *column*.

Several other commands behave just like **total**, except that the value they produce is not a total but: **datatype**, **maximum**, **minimum**, **mean**, **precision**, **width**, **length**, and so on. The all call the same program, but decide what to print depending upon the name used to call it.

Example

There are two formats. Without the option **-l**, **total** just gives the head lines and the total line. For example, to find the totals of columns **Debit** and **Credit** in table **journal**, type:

```
total Debit Credit < journal
```

This returns something like:

```
Date    Account Debit   Credit  Description
----    ------- -----   ------  -----------
                65000   65000
```

However, if you wish to see the whole table, as in a report, type:

```
total -l Debit Credit < ledger
```

This returns something like:

```
Account Date    Debit   Credit
------- ------- ------- -------
101     890102  25000
101     890103          5000
101     890104          15000
130     890104  30000
150.1   890103  10000
201.1   890104          15000
211.1   890102          25000
211.2   890103          5000
------- ------- ------- -------
                65000   65000
```

See Also

maximum, mean, minimum, subtotal

translate — /rdb Command

Word-for-word substitution using a translation file
translate *language* **<** *text* **>** *translatedtext*

The command **translate** performs word-for-word replacement on *text*. You can use it to perform crude translations from one language to another.

translate uses either **m4**, **sed**, or both. For **m4**, **translate** needs a define file generated by the **/rdb** command **tabletom4**; the define file must be named *language*. For **sed**, it needs a preprocessing file and can also use a postprocessing file, both generated by the command **tabletosed**; the preprocessing file must be named *language***.sed.1**, and the postprocessing file must be named **language.sed.2**. See the manual pages for these **/rdb** commands for a description of the sort of table they require as input.

Example

Suppose we have a simple translation for English to Deutsch (German), called **ed.t**:

```
English Deutsch
------- -------
I       Ich
love    liebe
you     dich
widgit
```

The command

```
tabletom4 < ed.t > ed
```

generates the **m4** definition file **ed**, which contains the following:

```
define(I,Ich)
define(love,liebe)
define(you,dich)
```

Note that **tabletom4** did not convert **widgit** because this word had no translation in the table.

Now, if file **text** contains the sentence

```
I love you.
```

you can use **translate** to translate it into German. Typing

LEXICON

```
translate ed text
```

prints the following on the standard output:

```
Ich liebe dich.
```

translate has also been used in a biomedical expert system. The idea is to convert each word into its value after looking it up in a table. For example:

```
Var     Value
------- -------
age     42
weight  80
 ...
```

This is something the Lisp language does frequently.

translate can also be used to construct form letters.

See Also

tabletom4, tabletosed, word
COHERENT Lexicon: **m4, sed**

trim — /rdb Command

Trim excess white space from a table
trim [-[l|n|ln] *col...*]**[-r[***n***]** *col...*]**[-t'***c***']...<** *table*

The command **trim** trims excess white space from each column in *table*. This lets you print a wide table with the minimum width for each column. Please note that **trim** removes all tabs from *table*, which renders it unreadable by any **/rdb** command, including **trim** itself.

trim without any options does the best it can to squeeze *table* without discarding any information. It make each column as wide as its widest column name or data item. You can also tell **trim** to reduce a column's width, even if it has to throw away characters.

Note that **trim** trim all columns, even if some are listed with the options. Most other **/rdb** commands, such as **justify**, affect only the columns named on the command line — unless none are listed, in which case they do the same thing to all of them.

Options

trim recognizes the following options:

n The number of characters remaining after the trim.

-*n column ...*
Return only *n* characters from the left of *column*.

-l*n column ...*
Return only *n* characters from the left of *column*.

-r*n column ...*
Return only *n* characters from the right of column.

-t´*c*´ Use character *c* as the column separator. For example, you may want to keep the tab character so that the table can be used by other **/rdb** commands.

When use this option, remember that the COHERENT shell gives special treatment to certain characters, such as the tab and the vertical bar '|'. You must quote these characters by enclosing them between apostrophes.

The above options affect the columns named. The options are *sticky,* which means that you can follow an option with as many column names as you wish.

Example

Consider, once again, table **inventory**, which appears as follows:

```
Item    Amount  Cost    Value   Description
------- ------- ------- ------- --------------
1       3       5.00    15.00   rubber gloves
2       100     0.50    50.00   test tubes
3       5       8.00    40.00   clamps
4       23      1.98    45.54   plates
5       99      2.45    242.55  cleaning cloth
6       89      14.75   1312.75 bunsen burners
7       5       175.00  875.00  scales
```

To trim it, type:

```
trim < inventory
```

This prints the following:

```
Item Amount Cost   Value   Description
---- ------ ------ ------- --------------
1    3        5.00   15.00 rubber gloves
2    100      0.50   50.00 test tubes
3    5        8.00   40.00 clamps
4    23       1.98   45.54 plates
5    99       2.45  242.55 cleaning cloth
6    89      14.75 1312.75 bunsen burners
7    5      175.00  875.00 scales
```

Note that the length of the column names **Item** and **Amount** determined the width of their columns, but in all other columns the data determined the width.

To examine what **trim** did exactly, you can use the command **see** For example:

```
Item Amount Cost   Value   Description
---- ------ ------ ------- --------------
1    3        5.00   15.00 rubber gloves
2    100      0.50   50.00 test tubes
3    5        8.00   40.00 clamps
4    23       1.98   45.54 plates
5    99       2.45  242.55 cleaning cloth
6    89      14.75 1312.75 bunsen burners
7    5      175.00  875.00 scales
```

Note that there are no tabs **^I**. This table is only for printing. Don't input or pipe this output to another **/rdb** command unless you use the **-t** option, which retains the tab as a column separator.

We can further squeeze the table with the options **-***n*, **-l***n*, and **-r***n*. For example, the command

```
trim -1 Item < inventory
```

reduces column **Item** to only one character. This produces the following:

LEXICON

```
I       Amount  Cost    Value   Description
-       ------  -----   ------- -------------
1       3       5.00    15.00   rubber gloves
2       100     0.50    50.00   test tubes
3       5       8.00    40.00   clamps
4       23      1.98    45.54   plates
5       99      2.45    242.55  cleaning cloth
6       89      14.75   1312.75 bunsen burners
7       5       175.00  875.00  scales
```

Note that to reduce *Item* to one character, **trim** had to throw away all but the 'I' of the column name **Item**. Note that the other columns were trimmed normally.

Because options are sticky, you can name several columns after one option: For example, the command

```
trim -5 Item Amount Cost Description < inventory
```

reduces the four named columns to a maximum of five characters. This produces the following:

```
Item    Amoun   Cost    Value   Descr
-----   -----   -----   ------- -----
1       3       5.00    15.00   rubbe
2       100     0.50    50.00   test
3       5       8.00    40.00   clamp
4       23      1.98    45.54   plate
5       99      2.45    242.55  clean
6       89      14.75   1312.75 bunse
7       5       175     875.00  scale
```

You can use the option **-r***n* both to save the *n* right characters of a column and to line up numbers on the right. For example, the command

```
trim -r3 Amount < inventory
```

produces:

```
Item    Amo     Cost    Value   Description
----    ---     -----   ------- -------------
1       3       5.00    15.00   rubber gloves
2       100     0.50    50.00   test tubes
3       5       8.00    40.00   clamps
4       23      1.98    45.54   plates
5       99      2.45    242.55  cleaning cloth
6       89      14.75   1312.75 bunsen burners
7       5       175.00  875.00  scales
```

Note that the column name **Amount** has been reduced to **Amo**. Also, the column is right justified as a result of removing the right three characters of each field in the column.

A complex trim might be done like this:

```
trim -1 Item -r3 Amount -r5 Cost -l7 Value Description < inventory
```

This command produces:

```
I       Amo     Cost    Value   Descrip
-       ---     -----   ------- -------
1       3       5.00    15.00   rubber
2       100     0.50    50.00   test tu
3       5       8.00    40.00   clamps
4       23      1.98    45.54   plates
5       99      2.45    242.55  cleanin
6       89      14.75   1312.75 bunsen
7       5       175.0   875.00  scales
```

We have squeezed this table a lot.

We have been working with an unjustified table. What if the table had been justified? First let's justify the table:

```
justify < inventory > tmp ; mv tmp inventory
```

Now **inventory** is justified, as follows:

```
   Item  Amount    Cost   Value Description
------- ------- ------- ------- ---------------
      1       3    5.00   15.00 rubber gloves
      2     100    0.50   50.00 test tubes
      3       5    8.00   40.00 clamps
      4      23    1.98   45.54 plates
      5      99    2.45  242.55 cleaning cloth
      6      89   14.75 1312.75 bunsen burners
      7       5  175.00  875.00 scales
```

Now, if we trim, you see:

```
Item Amount Cost   Value   Description
---- ------ ------ ------- -------------
               5.0   15.00 rubber gloves
         10    0.5   50.00 test tubes
               8.0   40.00 clamps
          2    1.9   45.54 plates
          9    2.4  242.55 cleaning cloth
          8   14.7 1312.75 bunsen burners
             175.0  875.00 scales
```

This mess results from trimming the right characters in the right justified columns **Item**, **Amount**, and **Cost**. This preserved the blank spaces, which are to the left in each column, and threw away the data. One way to fix this is to tell **trim** that the columns are right justified, like this:

```
trim -r Item Amount Cost < inventory
```

This produces:

```
Item    Amount  Cost    Value   Description
----    ------  ------  ------- -------------
1       3       5.00    15.00   rubber gloves
2       100     0.50    50.00   test tubes
3       5       8.00    40.00   clamps
4       23      1.98    45.54   plates
5       99      2.45    242.55  cleaning cloth
6       89      14.75   1312.75 bunsen burners
7       5       175.00  875.00  scales
```

You can also compress the file before piping it into **trim**. For further compression, use the command:

LEXICON

```
trim -r1 Item -r3 Amount -r6 Cost -l11 Description < inventory
```

This produces:

```
I Amo   Cost   Value Description
- ---   ----   ----- -----------
1   3   5.00   15.00 rubber glov
2 100   0.50   50.00 test tubes
3   5   8.00   40.00 clamps
4  23   1.98   45.54 plates
5  99   2.45  242.55 cleaning cl
6  89  14.75 1312.75 bunsen burn
7   5 175.00  875.00 scales
```

See Also

compresss, justify, trimblank

trimblank — /rdb Command

Remove leading and trailing blanks from a string
trimblank [*string***] [<** *file* **]**

The command **trimblank** trims all white-space characters from the beginning and end of *string*. This is useful in shell programming, in which you need to trim white space from a string that has been padded (justified) with space characters.

Example

The command

```
echo       '     here is the good stuff       ' | trimblank
```

prints:

```
here is the good stuff
```

See Also

compresss, justify, trim

tset — /rdb Command

Fetch termcap entry for a terminal type
tset [*TERM* **]**

The command **tset** duplicates the Berkeley UNIX program by the same name. It finds the entry in file **/etc/termcap** for terminal type *TERM*. If no terminal type is named on the command line, it returns the **termcap** entry for the type of terminal you are now using.

You can use **tset** to initialize an environmental variable, which you then filter with other **/rdb** or COHERENT commands.

Example

If you are working on your computer's console, your terminal type is probably **ansipc**. In that case, the command **tset** prints the following on the standard output:

```
:al=\E[L:am:bs:bt=\E[Z:bw:cd=\E[O:ce=\E[K:ch=\E[%i%d‘:cl=\E[2O:\
:cm=\E[%i%d;%dH:co#80:cs=\E[%i%d;%dr:cv=\E[%i%dd:dl=\E[M:ho=\E[H:\
:is=\E[25f\E[7m 1=Line_L 2=Line_R 3=D_Ln 4=Und_Ln 5=Undo 6=Und_Blk 7=Tag\
 8=Join 9=Rptx 10=Rptd \E[m\E[H:\
:k0=\E[0x:k1=\E[1x:k2=\E[2x:k3=\E[3x:k4=\E[4x:\
:k5=\E[5x:k6=\E[6x:k7=\E[7x:k8=\E[8x:k9=\E[9x:\
:kb=^h:kd=\E[B:kh=\E[H:kl=\E[D:kr=\E[C:ku=\E[A:\
:li#24:ll=\E[24;1H:hd=\E[C:se=\E[m:sf=\E[S:sg#0:so=\E[7m:sr=\E[T:\
:te=\Ec:ue=\E[m:up=\E[A:us=\E[4m:\
:KI=\E[5x:KD=\E[3x:Kd=\E[P:KB=\E[6x:KU=\E[4x:Ku=\E[@:KM=\E[7x:KJ=\E[8x:\
:Kt=\E[Z:KT=\t:KL=\E[1x:KR=\E[2x:KP=\E[U:Kp=\E[V:KX=\E[9x:KC=\E[0x:\
:KE=\E[24H:KW=^F:Kw=^R:Kr=^N:do=\E[B:
```

The command

```
TERMCAP=‘tset‘
```

initializes the environmental variable **TERMCAP** to your terminal's **termcap** entry. Storing this information in an environmental variable greatly speeds processing of programs that manipulate the screen.

See Also

termput
COHERENT Lexicon: **termcap**

LEXICON

union — /rdb Command

Concatenate tables

union *tableorlist ...* **[-] [<** *tableorlist* **]**

The command **union** concatenates all of the *tableorlist*s named on the command to form a new table or list. The new table or list contains all the rows in the first input table or list followed by the rows of the second, and so on. The input tables or lists must have the same number of columns. It is best that the columns have the same name and the same type of data, in the same order.

The hyphen '-' tells where in the list of files the standard input is to go. This gives you the freedom to use **union** in a pipe and to place the input file anywhere in the output.

Example

Consider table **journal**, as follows:

```
Date    Amount  Account Ref     Description
----    ------  ------- ---     -----------
820107  14.00   meal    v       meal with jones
820119  81.72   vitamin c       sundown vitamins
820121  20.83   meal    v       meal with scott
820121  2500.00 keogh   c       keogh payment
820125  99.00   dues    v       dues to uni-ops
```

and table **carexpense**, as follows:

```
Date    Amount  Account Ref     Description
----    ------  ------- ---     -----------
820113  101.62  car     v       car repairs
820114  81.80   insur   c       car insurance allstate
820114  93.00   car     c       car registration dmv
```

The command

```
union journal carexpense
```

produces the following output:

```
Date    Amount  Account Ref     Description
----    ------  ------- ---     -----------
820107  14.00   meal    v       meal with jones
820119  81.72   vitamin c       sundown vitamins
820121  20.83   meal    v       meal with scott
820121  2500.00 keogh   c       keogh payment
820125  99.00   dues    v       dues to uni-ops
820113  101.62  car     v       car repairs
820114  81.80   insur   c       car insurance allstate
820114  93.00   car     c       car registration dmv
```

The command

```
echo journal | union - carexpense
```

accomplishes the same thing as the previous example, but shows demonstrates how to use **union** with a pipe.

See Also

jointable, split
COHERENT Lexicon: **cut, paste**

uniondict — /rdb Command

Combine three tables into translation dictionary
uniondict *langtolang* **>** *definefile*

The command **uniondict** creates an **m4** define-file out of three tables. *langtolang* is a two-letter code that indicates which language is being translated to which; for example, **ed** stands for "English to German" (Deutsch).

uniondict expects one table of lower-case letters, such as **ed.low**, and one of all capital letters, such as **ed.allcap**, and builds its own table with initial capitals. Finally **uniondict** uses the **/rdb** command **tabletom4** to create a definefile for **m4**.

Example

To generate a definefile for English-to-German translation, type the command:

```
uniondict ed
```

If the two precursor files already exist, then this command generates a new file named **ed** that can be used by **translate**.

See Also

tabletom4, translate, word
COHERENT Lexicon: **m4**

unlock — /rdb Command

Unlock a record or field of a file
unlock *tableorlist processid from to indexfrom indexto*

The command **unlock** unlocks a record or file by removing a row from a common unlock file. File **/tmp/L***tableorlist* contains one line for each locked record or field plus *processid* of the process that locked it. *from* and *to* give, respectively, the offsets of the beginning and end of the locked record or field; and *indexfrom* and *indexto* give, respectively, the offsets of the beginning and end of the row or field's entry in the secondary index file.

Example

Let us use **unlock** on table **inventory**, which is as follows:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   -------------
   1         3    50      150   rubber gloves
   2       100     5      500   test tubes
   3         5    80      400   clamps
   4        23    19      437   plates
   5        99    24     2376   cleaning cloth
   6        89   147    13083   bunsen burners
   7         5   175      875   scales
```

LEXICON

First, we use the **/rdb** command **seek** to locate the record and return the offset and size:

```
LOCATION=‘echo 5 | seek -mb inventory Item‘
```

The above command returns the offset and size of column **Item** in row 5 of table **inventory**, and writes the informatio into environmental variable **LOCATION**. When we look at the contents of **LOCATION**, we see the following:

```
207 245 0 8
```

This means that the record begins 207 bytes into the file, ends 245 bytes into the file, and therefore is 38 bytes long.

Now we can call **unlock** to unlock the record:

```
unlock inventory $$ $LOCATION
```

Note that we use the environmental variable **$LOCATION** to supply the location. We also use the shell's built-in variable **$$** to write the process identifier onto the command line.

Let's see what the lock file looks like. Since we're locking table **inventory**, the lock file is named **Linventory**. If we type the command:

```
cat /tmp/Linventory
```

we see the message:

```
cat: cannot open /tmp/Linventory
```

unlock removes the lock file if it removes the last lock line.

See Also

lock, seek

update.inv — /rdb Command

Multi-user update with screen form and record locking
update.inv

The command **update.inv** is a sample shell program that demonstrates several features of **/rdb**, including multiuser input, fast access, screen-form input, and record locking. It updates an inventory file named **inv** that is in the directory **$RDB/demo** Use it to see how to use the COHERENT shell to tie **/rdb** and COHERENT programs together to form fast, multiuser, large-file applications.

The environmental variable **TERM** must be set to the correct name of your terminal, as described in the COHERENT system file **/etc/termcap**. To make the reverse video and underlining work, you will have to initialize environmental variables in your **.profile**.

Example

cd to directory **$RDB/demo** and run **update.inv**:

```
cd rdb/demo
update.inv
```

You'll the following template on your screen:

```
Makeapile, Inc.      Inventory Update       Mon Jan 8, 1990
Item Number: _______        (or hit Return key to exit.)
Item        Cost       Value      Description
----        ----       -----      -----------
Amount Onhand: _______
```

Try the program for yourself and study the program source.

See Also

lock, search, seek, replace, unlock
COHERENT Lexicon: **TERM, termcap**

update.rdb — /rdb Command

Display and edit records in any sized file
update.rdb [-l -m[bhirs] -v] *tableorlist* **[***keycolumn ...* **]**

The command **update.rdb** lets you find a record in *tableorlist*, display it, and update it.

update.rdb can find a record in any of several different ways: by moving up and down in the file, by record number, by matching a string pattern in a record, and a fast-access method search.

update.rdb lets you use a text editor to edit the record you find. To lock out other users who might be looking at your file, **update.rdb** uses the **/rdb** command **lock** to lock the record being edited. When you finish editing, **update.rdb** replaces the edited record back into *tableorlist*.

If the new record is smaller than the old record, it adds space characters to the end of the last column to pad out the record. If the new record is larger than the old record, it trims trailing and leading spaces from the columns. If **update.rdb** cannot find enough blank spaces to trim, it does not replace the record, but appends it to the end of the table and updates the fast-access method indexes, if any. If this happens, you have several options: You can reedit the record. You can append the record to the end of the file using **enter** and delete the current record. You can also pad each record with extra spaces so that there will be room when you do updates. To add padding, use the command **pad**.

Because **update.rdb** replaces an edited record to where the original was extracted, you do not have to rebuilt entirely any fast-access indexes that *fileorlist* owns. As long as you don't change the key columns, the fast-access methods can still find the records. This can mean a considerable savings in the time needed to update a record.

Also, **update.rdb** can handle any sized file, whereas the text editors and word processors mostly cannot handle files of more than a few thousand characters.

Options

update.rdb recognizes the following options:

-l No lock. Do not lock the record by blanking out the record in the file while updating. This is much faster than using record-locking; but it is truly safe if you are the only person who can update *tableorlist*. If more than one person can access *tableorlist* at the same time, do not use this option.

-m Fast access method. See the manual page for the **/rdb** command **search** for a description of these options.

-v Verbose. With this option, **update.rdb** prints information about what it is doing during each step of updating a record. You can use this information to find the bottlenecks in updating; for example, you may find that you are not using the best method of accessing the records you

wish to update.

Getting

To invoke **update.rdb** to edit *tableorlist*, simply type:

```
update.rdb tableorlist
```

update.rdb prompt you with a '>'.

You can exit from **update.rdb** by any of three methods: Type **<ctrl-D>**, type "quit", or simply type 'q' (because **update.rdb** looks at only the first character of the one-word commands).

Finally, the *kill* character on your terminal (usually **<ctrl-C>**) kills the program, as it does other COHERENT commands.

Help Menu

update.rdb has a help menu built into it. To see it, type **help** at the '>' prompt. You will see the following:

```
      Command Description | update.rdb [-l -m[bhirs] -l] file
-------------------------------------------------------------
CTRL-d          exit, hold down CTRL key and press d
                (see quit below)
Return key      display next record
number          display record number you enter
                (number = 1,29,...)
+number         move forward the number of records in the file
-number         move backwards the number of records in the file
/pattern        find next record with pattern
=key            find records with key matching what you type
!command        execute the unix command while still in update.rdb
!enter file     enter new records at end of the file

(You need only type the first letter of the commands below)

delete          delete the current record
m[bhirs] [col]  set up fast access method for key column
                b=binary, h=hash, i=index, r=record, s=sequential
lock            lock update.rdb record by blanking (toggle, true)
search key      find records with matching key
help            help list displayed (this help list)
quit            quit (also cntl-d)
update.rdb      update current record, allows you to edit, replace
verbose         report internal actions (toggle, false)
```

Finding a Record

There are several ways to find the record you want.

When **update.rdb** comes up, it displays the first record in *tableorlist*. To move to the next record, simply press **<Return>**. You can move forward any number of records with a plus sign and a number, for example, **+5**; and you can move backwards with a minus sign and a number, for example: **-16**. You can also select a specific record by typing its number. Record numbers are displayed with each record.

Pattern Search

If you try to go beyond the last record or before the first record, **update.rdb** warns you.

You can also search for a record that has a given string by typing a slash character followed by string. For example, to search for pattern "Ronald", type: **/Ronald**. **update.rdb** searches forward starting with the next record, then wraps around and goes to the first record in the file and

continues till it finds a match or returns to the current record.

Fast-Access Methods

update.rdb can use fast-access methods to find a record within *tableorlist*. You can turn on the fast-access methods in either of two ways:

1. Use the appropriate option on the command line. For example, the command

   ```
   update.rdb -mh inventory Description
   ```

   invokes the fast-access method for table **inventory**.

2. Type the appropriate option at **update.rdb**'s prompt. For example, typing

   ```
   mh Description
   ```

at **update.rdb**'s '>' prompt invokes the fast-access method for column **Description**.

Both ways tell **update.rdb** which method to use and which column is the key column.

Once invoked, you can search with either the **=** or **s** commands. For example:

```
> =keyvalue
> s keyvalue
```

Updating

To update the current record, simply type:

```
> update
```

Just the letter **u** is sufficient.

The steps in using **update.rdb** are as follows:

1. Invoke **update.rdb** with the name of the table or list to be edited.
2. Find the record you want by entering its row number, a string it contains, or through a fast-access method.
3. When the record is found, type **u** to update it.
4. **update.rdb** copies the record to a file with a header so that it is a one record table or list.
5. **update.rdb** replaces the selected record with a string of blanks of the same length, hold the record's place within its table, and to lock out other users.
6. **update.rdb** then copies the temporary record file to a backup file in case you damage it.
7. **update.rdb** then reads environmental variable **EDITOR** and calls the editor it names to let you edit the record you selected. If **EDITOR** is not set in your environment, it invokes **vi** by default.
8. Use the editor to edit the record.
9. When you exit your editor, **update.rdb** returns the edited the record into its table or list. It adds or removes trailing spaces to make the record fit into the same amount of space it previously occupied in its table or list. If the record doesn't fit into its old space, **update.rdb** lets you choose whether to re-edit the record, or append it to the end of the table or list.

   You can also validate a record before leaving your editor by typing:

   ```
   validate file.v < file
   ```

LEXICON

See the COHERENT Lexicon entries for the editor you are using for information on how to pass that command to the shell without exiting from the editor. The entry for the command **validate** in this manual describes how it works.

Other Useful Commands

If you need to see the current record again because it has run off the screen, simply type a period at **update.rdb**'s prompt.

To execute a COHERENT shell command, type an exclamation point and the command you wish to execute. For example, to execute the command **ls** from within **update.rdb**, type

```
> !ls
```

This feature also lets you append new records to a table while still within **update.rdb**. To do so, just type:

```
> !enter thisfilename
```

When you leave the **enter** command, you will be back in **update.rdb**.

You can also index the file you are working with from within **update.rdb**. Just type:

```
> !index -mi filename keycolumn
```

This command creates index file *filename***.i**, which **update.rdb** can then use for fast-access searching of file you are updating.

Padding

When you set up a table that you intend to update, you probably should pad it — that is, add extra space characters to each record. Then you can add characters to a record and **update.rdb** will trim off the space characters to fit it back into the file. For details, see the manual page for the command **pad**.

Notes

This command is named **update.rdb** rather than **update**, as in other implementations of **/rdb**, to avoid clashing with the COHERENT command **update**, which does something very different.

uppercase — /rdb Command

Convert input to all upper-case characters
uppercase [*string ...***] [<** *file***]**

The command **uppercase** converts to to upper-case characters every letter in *string*. If you do not pass it a *string*, it reads the standard input. It uses the COHERENT command **tr** to perform conversion.

Example

Here we convert a string to upper case:

```
uppercase little words
```

This displays

```
LITTLE WORDS
```

on the standard output.

If the file **lowers** contains the text

```
these are all lowercase.
```

then the command

```
uppercase < lowers
```

prints the following on the standard output:

```
THESE ARE ALL LOWERCASE.
```

See Also

cap, lowercase
COHERENT Lexicon: **tr**

LEXICON

validate — /rdb Command

Find invalid data
validate *'pattern* { *action* } **...' <** *tableorlist*

The command **validate** checks your tables and a list of invalid data. It also prints messages of your choosing, and shows you the line that caused the error. It is almost identical to a combination of the commands **row** and **compute**.

validate passes your commands to **awk** after it has converted your column names to **$1**, **$2**, and so on, which is how **awk** refers to columns.

pattern is a logical condition that indicates invalidity in the data; for example, it may give values that are negative and should not be, or greater than they should be. *action* is what to do about it; usually, this is to print a message.

Example

The first example is simple. Once again, consider table **inventory**, which is as follows:

```
Item#   Onhand  Cost    Value   Description
-----   ------  ----    -----   --------------
    1        3    50      150   rubber gloves
    2      100     X      500   test tubes
    3       -5    80     -400   clamps
    4       23    19      437   plates
    5      -99    24    -2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

Several invalid values have crept into this table. You might wish to check that no item in column **Onhand** is less than zero. To do so, type the command:

```
validate 'Onhand < 0 {print "negative Onhand in line " NR}' < badinventory
```

This prints the following on the standard output:

```
Item#   Onhand  Cost    Value   Description
-----   ------  ----    -----   --------------
    1        3    50      150   rubber gloves
    2      100     X      500   test tubes
negative Onhand in line 3
    3       -5    80     -400   clamps
    4       23    19      437   plates
negative Onhand in line 5
    5      -99    24    -2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

The variable **NR** means number of record or row. This variable is built into **awk**, which replaces it with the number of the line it is processing.

The next example is complex. Note that if you wish, you can write a command line for **validate** into a file, to create a customized validation command. You can put many tests and messages in one validation file. For example, the following command file, called **validateinv**, thoroughly checks table inventory:

```
echo validating badinventory

validate '{if (Onhand < 0 ) print "negative Onhand in line " NR; \
        if (Value < 0 ) print "negative Value in line " NR; \
        if (Cost ~ /[A-Za-z]/) print "letter in Cost in line " NR; }' \
        < badinventory
```

To validate table **inventory**, just type:

```
     validating badinventory
```

This prints the following on the standard output:

```
Item#   Onhand  Cost    Value   Description
-----   ------  ----    -----   --------------
    1        3    50      150   rubber gloves
letter in Cost in line 2
    2      100     X      500   test tubes
negative Onhand in line 3
negative Value in line 3
    3       -5    80     -400   clamps
    4       23    19      437   plates
negative Onhand in line 5
negative Value in line 5
    5      -99    24    -2376   cleaning cloth
    6       89   147    13083   bunsen burners
    7        5   175      875   scales
```

You may wish to redirect the output of **validate** to a file, bring it into your editor, and correct the errors and remove the error lines as you go. To make it easier to find those error lines in the editor, you could have all error lines contain a string that is easy to find, such as **ERROR**.

If you only want to see the error lines, use the command **row**.

See Also

Logical Conditions

compute, row
COHERENT Tutorials: **Introduction to the awk Language**

ve — /rdb Command

Visually edit a table
ve [*table [-n[n]* **-f***c* **-m***c* **-i -s [***file***] -h [***file***] -a [***file***] -v [***file***]]][-b]**

ve is an editor that is designed to edit tables in a multi-user environment. Because it is designed to work with tables, it understands rows and columns; and because it is designed for a multi-user environment, it knows how to lock records and perform other useful tasks to protect the integrity of of your data.

ve defines a row, or *record*, as containing a fixed number of columns, or *fields*. It defines a field as being a string delimited by an exclusive *column separation character*. Normally, the first two rows in a **ve** table are *header records* that describe the columns in each row. Each field in the first header

LEXICON

record names its column; fields in the second header record are hyphens '-'. When you invoke **ve** without an argument, it displays a menu of on-line **ve** help topics and instructions.

table is the table you wish to edit. When you name a *table* on the command line, all other arguments must follow it. If *table* does not exist, **ve** create sthe necessary header records from a screen file or a header table.

Options

ve recognizes the following options:

-b Use the terminal's highlighting capability. If specified, **ve** does its best to emphasize non-data text and the on-line help file's keywords by displaying them in high intensity. This may produce odd results if the entry in file **/etc/termcap** for your terminal is incorrect or missing, or if your terminal type is set incorrectly.

-n*n* Use automatic record numbering. **ve** assumes that the data in the first column of each record are unique numbers, and keeps track of the largest value in this column. As you add new records to the table, **ve** increments this number and stores it in the first column of the new record.

When you specify *n*, **ve** begins numbering records at that value. For example, **-n2001** means, "Use automatic record numbering and begin the numbers at 2001."

-f*c* Use the character *c* to delimit columns instead of the default tab character. With the exception of the screen file, all files associated with the **ve** table *must* use the same column separation character.

-m*c* Invoke **ve** in mode *c*, where *c* is one of the following:

i	Insert mode
n	Next mode
/	Search mode

ve returns to this mode after each successful *write*. Invoking **ve** in insert mode is convenient when entering data.

-i Initialize the table. **ve** maintains a private sequential index of all records in a table, so it can track the records in use and the records that have been deleted during an editing session. It also uses this index to clean up the table when the edit session is over. If the table is modified by a program other than **ve**, then you must use the **-i** switch during the next **ve** edit session to reinitialize its private index. You can use this index for fast searching by record number if each record has a number and no records are ever deleted.

If the current directory contains any files that follow **ve**'s naming convention **data-***c*, where -*c* is a file-option switch (**-s**, **-h**, **-a**, **-v**), **ve** incorporates them automatically into the command line before it processes the argument list. You must name on the command line all files that not named according to this convention; please note that this overrides the inclusion of any default names in the command line.

-s *file* Tell **ve** to use *file* as the screen file, which contains a template of how you want the screen to appear. The entire screen file should not exceed 23 lines. **ve** scans the screen file for column names enclosed between angle brackets (**<>**) and displays the data in those columns between the brackets. Both brackets must be on the same line. If *file* is used to initialize the table, all column names must be listed; otherwise, excluding a column name from a template also keeps data in those columns from being viewed, sought, or modified. If a column name is preceded by an exclamation point, **ve** lets you search for that column's data and display them, but no user can modify them.

-h *file* The header records are in *file*. If no *file* is named, **ve** assumes that the header records are in file **data-h**. **ve** can create a header file only when the table is initialized with a screen file.

-a *file* If the audit table does not exist, tell **ve** to create it and write it into *file*. If you do not name *file*, **ve** writes the audit table into **data-a**.

ve audits a table by recording each addition, modification, and deletion you make to it, and storing this information in its audit table. The audit table looks like the table, with the addition of two columns in the beginning of each record: the *time stamp* column and the *modification code* column.

The first column of each row in the table must be a unique number so that rows can be tracked accurately. **ve** manages these numbers and assigns new ones to the rows you add. If the table already exists, **ve** begins record numbering at the largest record number in the table plus one; for new tables, **ve** begins record numbering at one.

Each time you add a record (code **a**), change a record (code **c**), or delete one (code **d**), **ve** records in the audit table the date and time, the modification code, and the number of the record. When you change or delete a record, **ve** also stores it in the audit table the data that the record contained before you changed or deleted it.

-v *file* Use *file* as the validation table. If you do not name *file*, **ve** uses **data-v** by default. If the validation table does not exist, **ve** creates it, then uses itself to edit the new validation table so you can enter validation requirements for desired columns before you edit the table.

When you include a validation table in the command line, **ve** checks the data as it is entered into columns, which are specified in the validation table, to make sure they meet certain requirements.

The first header record of the validation table contains a *name* column, a *character* column, a *length* column, and a *table look-up* column, in that order.

name column
This names a column from the table being validated.

character column
Give the valid characters or ranges of characters for data in this column Characters and ranges must be separated by commas (,). Ranges or characters that are preceded by an '!' exclude that range or character from the valid list. For example, the validation string **a-z,!y,1-9,#,., ,,,** means that all lower-case characters with the exception of *y* are valid, as are numerals from 1 to 9, pound symbols, periods, blanks, and commas. By contrast, the string **!0-9** excludes all numerals from the column.

length column
Set the length limit of the column, as follows:

<*n* The column cannot exceed *n* characters in length.

>*n* The column cannot be less than *n* characters long. The **<** and **>** symbols can be used with each other. For example, *>10,<28* specifies that the length of the data in this column must be at least 11 and not more than 27 characters long.

=*n* The column can contain only *n* characters.

! Zero indicator. This can used with any of the above symbols to indicate that the column can also have a length of zero. For example, *!,=5* means that the column can contain zero characters; if it does not contain zero characters, it must contain exactly five characters.

LEXICON

table look-up column

Tell **ve** to check the data entered against the contents of an existing table. When the name of the table is preceded by '!', data entered in the column must *not* be in the table; otherwise they *must* be in the table.

You can create an index table with the command **vindex**. It names them after the column you are **vindex**ing, which is called the *key column*. If the name of the key column is longer than 11 characters, **vindex** truncates the name to 11 characters. It also converts blanks in the key column's name to '_'.

For example, the specification *!name* tells **ve** to check the data against a table comprised of all text located in a column named *name*. If the data are *not* found in the table, then it passes validation.

Commands

The following table gives the **ve** edit commands. **ve** uses two modes when editing a table: *command mode* and *insert mode*. Commands that switch **ve** from command to insert mode are indicated by <ESC>. The ESC (escape) key switches **ve** from insert to command mode.

The commands **c** and **d** commands require *targets*. A target is the destination for the command being executed. For example, the command **dfa** means, "delete text from the cursor up to and including the first *a* character encountered." In this case, the target command is *f*. In the table, commands that can be used as targets for the **c** and **d** commands are indicated by **(t)**.

Some commands recognize a number that indicates how many times it should execute. For example, the command **5l** means, "Move right five characters"; the command **3rP** means, "Replace the next three characters with the character *P*." Counts may also be used with targets; for example, command **c2w** means, "Change the next two words." Commands that recognize counts are indicated by a **(c)**.

Command	*Description*
A*text*<ESC>	Append text at field end
a*text*<ESC>	Append text after cursor
B	Back field (c)
b	Back word (t,c)
C<ESC>	Change to field end
c<ESC>	Change to target
D	Delete to field end
d	Delete to target
dd	Delete record
e	End of word (t,c)
F*x*	Find x left (t,c)
f*x*	Find x right (t,c)
G	Go to last record (c)
H	First field
h	Left character (t,c)
I*text*<ESC>	Insert text at field start
i*text*<ESC>	Insert text before cursor
j	Down field (c)
k	Up field (c)
L	Last field
l	Right character (t,c)
m	Mark search field
M	Unmark search field
n	Get next record
O*text*<ESC>	Open new record

..... **o***text*<ESC>.....Open new field
..... **P**.....Put yanked record
..... **p**.....Put yanked field
..... **q**.....Quit **ve**
..... **R**<ESC>.....Replace
..... **r***x* .....Replace with *x*
..... **S**.....Redraw the screen
..... **s**<ESC> .....Substitute text (c)
..... **t** .....Table display
..... **u**.....Undo last command
..... **v**.....Validation display
..... **W**.....Write record
..... **w** .....Next word (t,c)
..... **x**.....Delete character (c)
..... **Y** .....Yank current record
..... **y**.....Yank current field
..... **ZZ**.....Write record & quit
..... **z**.....Field display
..... **;**.....Repeat F/f cmd (t)
..... **^**.....Start of field (t)
..... **$**.....End of field (t)
..... **/***text***<ESC>**.....Search for text
..... **/<ESC>** .....Repeat search
..... **-**.....Get preceding record
..... **%**.....Execute the shell
..... **!**.....Shell escape
..... **!!** .....Repeat shell escape
..... **>**.....Scroll field left
..... **<**.....Scroll field right
..... **#**.....Help file(s) display
..... RETURN.....Next field (c)
..... TAB .....Next field (c)
..... SPACE.....Right character(t,c)

The next table gives two other kinds of **ve** commands — *colon* commands and *control-key* commands, all of which map directly to commands in given in the previous table. Colon commands are provided for users familiar with **vi** or **ed**. Control-key commands let you define command macros by specifying the control key character and the command or commands to be executed each time the control key is hit. These definitions, or *macros* are defined in the **/rdb**-formatted file **.verc** located in the your home directory. If there is no **.verc** file, then the default control-key commands apply. If you are in insert mode when a control key is hit, **ve** attempts to restore that mode after successful execution of the macro.

Command	*Maps to*	*Command*	*Maps to*
:h[elp]	#	**<ctrl-A>**	#
:n[ext]	n	**<ctrl-R>**	S
:o[pen]	O	**<ctrl-T>**	t
:q[uit]	q	**<ctrl-Q>**	ZZ
:s[hell]	**<ctrl-S>**	%	
:w[rite]	**<ctrl-W>**	W	
:!cmd	!cmd	**<ctrl-V>**	v
:!!	!!	**<ctrl-D>**	dd

See Also

vindex

LEXICON

COHERENT Lexicon: **elvis, vi**

vilock — /rdb Command

Lock a table before editing, unlock afterward
vilock *tableorlist*

The command **vilock** stops others from editing the same file you are editing. It locks the table, invokes the **vi** with which you can edit the table, then unlocks the table when you have finished editing.

To lock *tableorlist*, **vilock** changes its name to **LOCK***tableorlist*; this is a temporary file that exists only as long as you are editing. **vilock** then invokes **vi** for **LOCK***tableorlist*. When you finish editing and exit from **vi**, **vilock** moves **LOCK***tableorlist* back to its original name. If anyone else trys to edit *tableorlist* while you are editing it with **vilock**, he receives a message that the file is locked or missing.

See Also

lock, unlock, ve
COHERENT Lexicon: **elvis, vi**

vindex — /rdb Command

Create and display ve look-up tables
vindex *table* **[** *key-column* **[:** *decode-column* **] ...]]**

The command **vindex** creates and displays tables composed of columns in **ve** tables. These tables are used by **ve** when validating column entries. *table* names the **ve** table. When no options are specified, **vindex** displays the contents of all existing column tables associated with *table*.

For each *key-column*, **vindex** collects the text from that column and stores it in two files; each is prefixed by the name of the *key-column* and is suffixed by **-A** or **-B**. If the name of the key column exceeds 14 characters, the table-name prefix is truncated to 14 characters. Key column whose names contain blanks must be surrounded by quotation marks; blanks are converted to underscores '_' in the table name. These files are referred to in the fourth column of **ve** validation files by the table-name prefix.

When the colon ':' argument follows a key-column name, the column name following this argument is assumed to be the name of a *decode-column*. As key-column text is collected in the table, the associated **vindex** links the decode-column's text to it as a cross-reference.

Example

The following **vindex** command create a look-up table for the key columns **Snumber** and **State** in table **personnel**; it also cross-references the information in **Snumber** with the text in the decode-column **Employee**:

```
vindex personnel "Snumber" : Employee State
```

The resulting tables are named **Snumber** and **State**.

See Also

ve

whatis — /rdb Command

Display the command description and syntax
whatis *command*

The command **whatis** displays the description and syntax of *command*. It resembles the COHERENT command **help**, except that it handles **/rdb** commands.

Example

The command

```
whatis column
```

displays the following:

```
column    - display columns of a table in any order
column    column ... < tableorlist
```

And the command

```
whatis whatis
```

displays:

```
whatis     - displays the command description and syntax
whatis     command
```

See Also

whatwill
COHERENT Lexicon: **help, man**

whatwill — /rdb Command

Display commands with functions in description
whatwill *function*

The command **whatwill** displays the description and syntax of the command that performs *function*. It resembles the Berkeley UNIX command **appropos**. It also resembles the **/rdb** command **whatis** except that it seeks *function* anywhere in the name of the command, the description, and the syntax. For example, if you ask for **total, whatwill** returns both **total** and **subtotal**.

Example

The first command looks for information about columns:

```
whatwill column
```

This returns:

```
column    - display columns of a table in any order
column    column ... < table
```

LEXICON

And the command

```
whatwill total
```

returns:

```
subtotal   - outputs the subtotal of columns in a table
subtotal   [ -l ][ break-column column ...] < table
total      - sums up each column selected and displays
total      [ -l ][ column ... ] < table
```

See Also

whatis
COHERENT Lexicon: **help, man**

widest — /rdb Command

Output the width of the widest entries in a table
widest < *table*

The command **widest** reads *table* and writes to the standard output a second table that gives the width of the widest entry in each column.

Example

The command

```
widest < inventory
```

reads table **inventory** and writes something like the following to the standard output:

```
Item    Amount  Cost    Value   Description
----    ------  ----    -----   --------------
4       6       4       5       14
```

Each number is the width, in characters, **inventory**'s the widest entry in that column. You can use this information for formatting. **/rdb** does not require a schema because it gets the parameters of the tables from the tables themselves.

See Also

width

width — /rdb Command

Display the width of each column
width [-l] [*column ...* **] <** *table*

The command **width** displays the maximum width of each *column* in *table*. If no column is named, it returns this information for every column in *table*. The option **-l** displays the entire table as well as information about the maximum width in each *column*.

Example

For an example of this command, see the entry for **total**.

See Also

total, widest

word — /rdb Command

Convert text file into list of unique words
word < *text* **>** *table*

The command **word** reads *text* converts it to a list of the unique words it contains, and writes the list to the standard output. You can use **word** to help construct tables for the **/rdb** command **translate**.

Example

If you were to type the above paragraph into a file called **FOO**, the command

```
word < FOO
```

would print the following to the standard output:

```
The
You
a
and
can
command
construct
contains
converts
for
help
it
list
of
output
rdb
reads
standard
tables
text
the
to
unique
use
word
words
writes
```

See Also

translate
COHERENT Lexicon: **sort, tr, uniq**

Notes

Because **word** is a shell script, you can modify it to map upper-case characters to lower case, if you wish.

LEXICON

Index

to _

A

B

C

INDEX

D

E

F

G

H

I

J

Q

R

S

T