1981

The Flat File Database Generator Ffg

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
81-379

The Flat File Database Generator Ffg

Douglas Comer

Computer Science Department
Purdue University
West Lafayette, IN 47907

September 1981

ABSTRACT

A flat file is the simplest possible database. It consists of a single, unformatted text file in which each line corresponds to a record. k-1 occurrences of a separator character divide each record into k variable length fields; the separator character does not otherwise appear in the file. Unlike most database systems, the flat file system is not a single, large program. Instead, it consists of a set of small, independent programs, called primitives, that each perform one basic operation. The user composes a subset of the primitives by directing the output of one to the input of the next in order to perform complex retrieval or update operations. Because they are independent, primitives are easily modified or replaced, and one can add programs to the set of primitives. Both the selection of primitives as well as their implementation are discussed.

CSD-TR-379

1. Introduction

A flat file is the simplest possible database. It consists of a single text file, F, containing zero or more lines, where each line is thought of as a record. Records are further divided into k fields, f_1, f_2, ...,f_k by k-1 occurrences of a distinguished separator character, S. Although k is fixed over all records, the length of individual fields is not. The flat file generator, ffg, is a database system that provides facilities to create, query, and modify a flat file database.

Unlike most commercial database systems that consist of one or two large programs to process queries and modify the stored data (see DATE75, ULLM80), ffg consists of many small programs, called primitives, that each perform one basic operation. A user composes a subset of the primitives by directing the output of one to the input of the next in order to perform a complex task. Because the primitives each perform one basic operation, selecting an appropriate combination is straightforward and natural. Because primitives are independent programs, they can be modified or replaced, and one can add programs to the set. The ease of extension and modification is important in achieving flexibility because it allows one to tailor ffg for each application.

Constructing systems as a set of primitives is not new. Kernighan and Plauger [KEPL76] describe primitives for program and text manipulation; Hanson [HANS79] extends them. Borden et. al. [BOGS79] describe an electronic mail preparation and reading system implemented in primitives.

Interestingly enough, most of the experiments with primitives have their roots in the UNIX operating system [RITH78,KEMA79]. Unlike most systems which encourage one to build large integrated programs, UNIX encourages one to build independent programs and connect them together. It provides a simple and efficient mechanism for passing data between running programs. It includes a convenient and simple notation for describing a composition. It

treats I/O to files, devices (like terminals), and other programs uniformly, so one does not need to know how a program will be used when writing it. UNIX contains sets of primitives for text processing and language development.

The next section of this paper describes pertinent parts of the UNIX environment in more detail, and gives the reader some appreciation of how UNIX influenced the flat file design. The following sections describe the flat file primitives, give an example of using flat files, and discuss their implementation. The paper concludes by discussing the merits of systems constructed from primitives.

2. The UNIX Environment

UNIX contains a large set of independent primitives, called *commands*, and a mechanism for composing them, called a *shell*. The UNIX shell [BOUR78] is a simple programming language that can be used interactively (as a command interpreter would be), or invoked to read input from a file (as a programming language interpreter would be). Shell programs are called *shell scripts*, or just *scripts*. If one tries to execute a file that contains a shell script, the system automatically invokes the shell to interpret it. Thus, a shell script functions just like a compiled program. In fact, some of the system commands are implemented as shell scripts.

The shell has facilities to invoke commands, direct the output of one command to the input of the next, or direct the input (output) of a command to a file or I/O device. Control statements (e.g., *while*, *for*, *if*, etc.) provide indefinite iteration, conditional execution, and definite iteration much like conventional programming languages. Unlike conventional languages, the shell only supports one data type – that of character string. It relies on commands to evaluate numeric expressions, test file status and accessibility, and handle complex computations.[1] One learns quickly that the art of constructing shell programs lies in

[1]Other shells, like the C-shell written at U.C. Berkeley, evaluate expressions directly.

composing and invoking commands, not in using the shell exactly as one would use Algol 60 or Pascal.

UNIX includes a mechanism for composing primitives called a *pipe*. Pipes, denoted by "|" in shell scripts, connect the output from one program to the input of another. One writes

```
a | b
```

to invoke programs *a* and *b* with the output from *a* connected to the input of *b*. The line

```
a arg1 | b arg2 arg3 | c
```

specifies a pipeline connecting the output of program *a* to the input of program *b* and connecting the output of program *b* to the input of program *c*. Program *a* has one argument (arg1), program *b* has two (arg2 and arg3), while program *c* has none. *a*, *b*, and *c*, could be the names of shell scripts or compiled programs.

UNIX contributed to the construction of ffg in several other ways:

1. All system services are available at the command level. One can create files, change protection modes, trap exceptions, and perform other tasks directly from the shell in UNIX. On many systems such tasks require special programs, often in assembler language.
2. UNIX provides a rich set of text manipulation primitives that ffg uses extensively.
3. UNIX is a late binding system. There is little distinction between data and program; one can write a file and invoke the shell to run it as a script.

The UNIX environment is not perfect, but it contributed nicely to the experiment.

3. Evolution of Flat File Primitives

Recall that a flat file consists of a single, unformatted text file where each line corresponds to a record, and that each record is divided into variable length fields by occurrences of a separator character. The operations that one typically performs on a flat file include:

add	new records to the file,
select	record(s) from the file having certain characteristics,
delete	record(s) from the file having certain characteristics,
change	field(s) on specified record(s),
format	the file for human consumption,
sort	the records according to the contents of some field(s)

Our local version of UNIX contains many flat files. One of the more well known, */etc/passwd* contains a record for each user; it associates a symbolic name, encrypted password, and other information with the user's login id. Another flat file contains inventory information for computer terminals.

The terminal inventory database is significant for two reasons. First, it provided the early motivation and testbed for the flat file experiment; it will be used here to illustrate the process of choosing primitives. Second, it demonstrates how the flat file system can be extended to each application. In particular, the historical narrative that follows shows how the flat file primitives evolved, and how they have been extended for the terminal inventory application.

The first terminal inventory database consisted of two flat files -- one for "terminals" and the other for "ports". Both were maintained by hand, using a text editor. The former contained a record for each computer terminal, giving its type, serial number, physical location, and connection to the machine. The latter contained a record for each machine connection (port), giving, among

other things, the terminal that was connected to it. When the number of ports and terminals grew to more than a handful, keeping the information in the two files accurate and consistent became difficult. It was decided that a program could be written to help with the maintenance. Unfortunately, thinking of the data as two separate files with many cross references made a program to manipulate it both highly specialized and cumbersome.

The first step toward a flat file database system occurred when the two data files were combined into one, called *terminfo*. Records in the terminfo file corresponded either to a port or to a terminal: those with a null "terminal" field corresponded to I/O ports on the machine that were not connected to a terminal, those with a null "port" field corresponded to terminals that were not connected to the machine, and those with data for both "port" and "terminal" fields corresponded to a connection. The point here is that the basic operations worked on connections (terminal,port) rather then on terminals or ports; the process of identifying those primitive operations brought this out.

Terminfo became the source of all information about terminals and ports throughout the system. All system files are generated from it automatically, eliminating the need to change them by hand every time a terminal is moved. For example, the system expects the file */etc/ttys* to contain a line for each port on the system, with a code indicating whether that port is connected to a terminal or not (i.e. allows user login). A utility program was built to scan terminfo and extract the information for */etc/ttys*. Other utility programs were built to extract information for other files. Whenever terminfo changes, the utility programs are run to correct other files throughout the system.

The set of utility programs to manipulate terminfo grew quickly, but there was little or no attempt to maintain uniformity or to make them work well together. There was a program called lookup to retrieve the record for a termi-

nal with a given serial number, and one called format to write the database in a neatly formatted fashion. The move toward a consistent, uniform set of primitives began when lookup and format were modified to work in conjunction with each other. Using the modified versions, a user could type

```
lookup -terminal 53 | format -
```

to obtain a neatly formatted listing (including headings) of the data for terminal 53. The change to the format primitive was simple: given a minus sign ("-") as an argument, it formatted its input; otherwise, it formatted the entire terminfo file. Although a general purpose format primitive was a good idea, forcing the user to distinguish when it was used in a pipeline was not. Often, one forgot the argument as in:

```
lookup -terminal 53 | format
```

and received a listing of the entire file.

In spite of the problems with the utilities, others began to copy and modify the set of programs to create their own databases. After some experience with terminfo and related databases, the set of primitives were redesigned completely to achieve several goals:

1. The primitives should supply the ability to retrieve, sort, format, delete, replace, and edit any database stored as a flat file.
2. All fields in the file should be named; one should not have to specify a field by its relative position as in the early version and in some of the UNIX commands.
3. The database system should work from a single descriptor file that described the fields, their names, and their format.

4. All programs should work together in a simple, uniform, and automatic way. For example, it should be possible to retrieve a subset of the records, sort them, and format them. One should not have to type special names or arguments to use the programs in a pipeline.

5. The system should protect against loss of information.

6. The system should be implemented as a set of primitives.

The redesign resulted in a flat file generator called ffg. The next section describes the ffg primitives in detail and shows how they work together.

4. Primitives in Ffg

The set of ffg primitives includes the following (see Appendix A for a detailed description of the parameters for each primitive).

delete	Omit specified records, write out the others.
edit	Allow the user to invoke a text editor on the database directly. Edit makes a backup of the database for the undo primitive.
enter	Interactively enter records one at a time.
fielded	Change the contents of specified fields on specified records.
format	Format the data for human consumption.
lookup	Retrieve records satisfying given criteria. Lookup is shorthand for simple retrieval requests.
retrieve	Retrieve records satisfying given criteria.
showfields	Display the fields description file (usually as an aid for users who forget field names).
sortby	Sort the input according to one or more fields.

undo Restore the entire database to its previous value.

update Replace the database by the file given as input.

verify Verify the internal consistency of the data.

Ffg achieves most of the design goals listed above. The above primitives all expect their arguments to contain symbolic field names as specified in a *fields description file* that the user supplies when creating the database. They work together, and automatically detect whether their input is connected to the output of another program, reading from the database if it is not. The system is implemented as a set of primitives, and the system does have a limited form of protection. The following discussions show, in more detail, how the flat file primitives achieve these goals.

The ffg primitives depend on a fields description (FD) file to relate symbolic field names to relative positions. The FD file contains k lines, one line for each of the k fields in the flat file. Each field is described by giving its relative position, its name, its sort type (e.g., numeric, to be placed in descending order), its length on a formatted listing, and two lines of heading information to be printed on formatted listings. The six items for each field are terminated by colons. For example, the FD file:

```
1:last::20: Last Name:----------:
2:first::20: First name:----------:
3:phone:n:13:Phone Number:----------:
```

describes the three fields for records in a phone book. The first field, named "last", holds a last name, the second field, named "first" holds a first name, and the third field, named "phone", holds a phone number. For purposes of sorting, the third field is considered numeric; the first two are sorted in dictionary order. When a flat file is formatted using this description file, it will look like:

Last Name	First Name	Phone Num.
llllll	fffff	pppppp

where the actual data for last names, first names and phone. numbers appears in place of llllll, fffff, and pppppp. Fields longer than the number of columns allocated in the listing are truncated, and fields shorter than the number of columns allocated are padded with blanks; the user can specify whether the padding is to the right or left.

The retrieve primitive is especially interesting because it illustrates the power of the flat file system. Retrieve takes as an argument a Boolean expression, B, and retrieves all records that satisfy B. The expression can contain comparative operators *less than*, (<), *greater than*, (>), *equal to* (==), *not equal to* (!=), etc., logical operators *and* (&&), *or* (||), and *not* (!), arithmetic operators (+, -, *, /, etc.), and pattern matching operators *matches* (expr~/pattern/), and *does not match* (expr!~/pattern/).[2] One can ask questions like "find all records where the last name starts with the letter *C* and contains the letter *r*"

```
retrieve 'last~/^C.*r.*/'
```

or "find all records where the tax field is greater than 50 and the department field is equal to *cs* or where the manager is *smith* and the department is not *cs*"

```
retrieve '(tax>50&&dept=="cs") || (manager=="smith"&&dept!="cs")'
```

It is important to note that one can only ask for intra-record comparisons, not for inter-record ones. Thus, one cannot ask for records with salary field greater than the previous one, nor can one ask for all records where the salary field is greater than the salary field of the 2nd record.

[2] See Appendix B for details of expression syntax.

The flat file primitives work well together, and automatically read from the database when their input is not connected to another program. For example, once the field description file and database are in place, one merely types:

```
format
```

to obtain a formatted listing of the data with headings. Typing

```
sortby phone | format
```

instead, causes format to read and format the output of the sortby primitive. In this example, the listing will be sorted by phone number. Similarly, typing:

```
retrieve last=="comer" | sortby phone | format
```

causes the retrieve primitive to select all records with last name equal to the string "comer", pass the results to the sortby primitive which will order them by phone number before passing them to format where they will be formatted.[3] One need not specify the origin of the data as a parameter.

Ffg helps prevent the loss of information through the update primitive. To make a permanent change to the data, one must create the new file and pipe it into update. Thus, to sort the example database according to last name, one types:

```
sortby last | update
```

Update saves a copy of the old file before replacing it, so one can recover the previous state of the database by typing:

```
undo
```

[3] The shell syntax actually requires that the quotes be escaped by typing a backslash in front of them.

Update is more sophisticated than one might expect. It actually unlocks, writes, and then relocks the database so that under usual circumstances even the owner cannot write directly to the file. Keeping the data file unwritable is especially important in UNIX where it is easy to direct the output of a program to a file, or to accidently pass a file name as an argument to a command. Update also maintains a mutual exclusion among processes that wish to update the database. The most common way to enter records interactively is by invoking the primitive enter which prompts for each field:

enter | update

One of the chief advantages of the primitives-based approach is that it allows users to intermix their own primitives with those that are supplied. For example, our ffg version of the terminfo database has a command to move a terminal from one port to another because terminals are moved frequently. In another application, a primitive called "gather" has been added to gather statistics on program use and write them into a flat file. The ffg system itself does not need to know about moving terminals, gathering statistics, or any of the other special commands that users invent. Yet having the primitives from ffg do most of the work made both applications significantly easier to implement.

The evolution of the flat file primitives took about 3 months -- much longer than expected. Most of the time went into testing. Several users built flat file databases, but measurements showed that they spent most of their time doing simple retrieval and formatting. Gradually, they added their own primitives, and began exploring new ways to connect old ones. Of course, others suggested changes that were tried in later versions.

From the experience, two observations can be made about the choice of primitives:

1. Ad hoc extensions to a unified set of primitives almost always result in disaster. For example, at one point we added a "delete" primitive that actually modified the database by retrieving records that were not to be deleted and passing them to update (unlike other primitives that had to be composed with update explicitly). One had to remember that delete worked differently than other commands, and that it could not be composed with them. Worst of all, composing delete with update created two processes that tried to modify the database, so one of them gave the cryptic report: "database is locked while another process updates it".
2. The greatest asset in the design of a clean, uniform set of primitives is a single person who has ultimate responsibility. This is akin to the chief programmer concept [BAKE72].
3. Designs by a single individual are prone to gross omissions in functionality. This should not come as a surprise, but it did.

5. The Implementation of Ffg Using UNIX Programs

If the primitives-based approach to computing works so well, why not use it to build the primitives themselves? This section answers that question by explaining how the ffg system, including the primitives, are built from existing UNIX programs. It discusses the UNIX programs upon which the flat file generator are built, the generation of a database, and binding of names.

The UNIX command awk [AHKW79], forms the backbone of the ffg retrieve and format primitives. Awk invokes an interpreter for a simple, but powerful string processing language. The interpreter reads an awk program, sometimes called an *awk script*, and then reads and processes a text file line-by-line according to the program. Awk divides each line of the input file into fields based on occurrences of a separator character, and permits one to examine or

write the contents of the i^{th} field. (To reference the i^{th} field of the current record, one writes $i in the awk program). Awk supports assignment statements, fairly powerful arithmetic, logical, and string operators, and even formatted output. In short, an awk script suffices for flat file retrieval or formatting provided one finds a way to translate field names into positional references.

How can an expression containing field names be processed by awk which only understands positional references? One might expect the implementation of retrieve to solve the problem as follows:

1. A user invokes retrieve, passing it an expression, B, that contains field names.
2. Retrieve passes the expression to a program, T, that parses the expression, translates field names into positional references, merges the modified expression with the skeleton of an awk program, and writes the result on file F.
3. Retrieve invokes awk giving it F as input. The program contains only positional references.
4. Interpreting the program on F, awk reads the database, evaluates the expression for each record, and writes out those that satisfy it.

This design was not used because it meant writing a program to parse and translate expressions; the objective was to use existing programs.

Retrieve turns the solution around, leaving the expression alone, but giving awk enough information to evaluate it. To do so, retrieve introduces k variables into the awk program and assigns them the contents of the k fields with k assignment statements. The essential piece of the awk script is:

$field_1$=$1
$field_2$=$2

.
.

$field_k$=$k

if (EXPRESSION) write out the record

where $field_i$ denotes the i^{th} field name and EXPRESSION denotes the Boolean expression as typed by the user. When evaluating the expression, awk binds references to field names to the variables that have been assigned the contents of the field. Making the extra assignments introduced some extra overhead; measurements are given in a later section. Similar constructions were used in other commands.

Implementing most of the remaining primitives from UNIX commands was not difficult, but a few problems arose. Processing minimal abbreviations for field names presented the worst challenge because no simple combination of UNIX commands produced the desired result. For example, if the set of possible field names are: "salary", "dept", "division", "dependents", and "name", one need only give sortby a prefix of the field name that uniquely identifies it (this specification was made before the implementation was considered). It means that "n" suffices for "name", but nothing shorter than "depe" may be used to designate "dependents" because it does not distinguish "dependents" from "dept". The shell supports pattern matching, so such abbreviations can be handled there. To do so, one must translate a list of field names like "salary", "dept", division", "dependents", and "name" into a list of patterns like "s*", "dept", "di*", "depe*", and "n*". Ffg performs these translation with an awk script, although it is more or less a conventional program. The result is that ffg contains no compiled programs, but it does contain some programming.

Ffg is a more than a collection of shell scripts for the primitives; it is a flat

file database generator as well. When invoked as a command, ffg builds a flat file database system, including a copy of the primitives, a field description file, and an access command. The user supplies information on the separator character, protection modes, fields description file, and the location of the access command; ffg generates the necessary files.

Each flat file database resides in a separate directory along with copies of the primitives and two subdirectories: "Specs" and ".system". The subdirectory Specs contains specifications like the fields descriptions that a user may change. Such changes are infrequent, however, so the information is kept out of the main directory. Additional files, that the user should not change, are kept in the .system subdirectory (e.g., mutual exclusion lock files).

Each flat file has an access command that one invokes to move to the database environment. When invoked, an access command changes the user to the database directory, records the user's presence, and invokes an interactive shell that reads and processes commands. After the user finishes work and exits from the interactive shell, the access command returns to the environment from which it was invoked. Normally, only one user can gain access to a flat file at a time; the access command refuses to grant access to a database that is in use. One can obtain nonexclusive use, find the status of active users, when they began, and their system identification. One can also ask for the creation time, mode, and size of the database and backup files.

Ffg optimizes the primitives by performing some bindings early. For example, when ffg constructs the retrieve primitive, it reads the field description file and binds field names into the shell script as described earlier in this section. This simple optimization improves performance dramatically because it eliminates the need to open the field description file, build the awk program, and have awk read the program back in. It also means that the user must inform

the system of changes in the description file. Whenever such a change occurs, the primitive *rebuild* will correctly recreate the primitives (including itself, if necessary). One would expect such changes relatively infrequently, however, when compared to the other operations.

Unlike most primitives which must be rebuilt manually, the *format* primitive is capable of detecting new formats automatically. The user views format as a late binding command, one that searches a special directory for a named format description file every time one invokes it. Actually, the names and format specifications are bound into the shell script to speed execution. The command searches for new formats only if the named file has not been bound previously. When it detects that a new file exists but has not been bound, format moves itself out of the way, uses rebuild to create a new version of itself, and then replaces the running version with the new one (i.e., performs a UNIX *exec*). Subsequent uses of the new format run at high speed.

6. Execution speed

The obvious advantage of early binding is execution speed; the obvious disadvantage is user impact. As on most timesharing systems, performance is best measured by response time. Users gladly tolerate a response delay of a few seconds for retrieval from a 200-line database, but they will not wait 30 seconds for the same information. Without early binding, response times for a pipeline of five primitives approached 30 seconds on our moderately loaded system. On the other hand, the optimized versions of the primitives were able to handle much larger files. Table 1 shows response times for a database of 1900 records. In the table, the command "cat" is a UNIX program that copies a file to its output unchanged; one expects cat to run at the maximum possible speed. Another UNIX command, "grep", scans a file and prints those lines that match a pattern. Finally, the UNIX command "wc" counts the lines, words, and characters in a file.

command primitive	response time in seconds
grep	4(2.2 cpu)
wc	4(1.9 cpu)
retrieve	22(18.1 cpu)
awk (retrieve program called directly)	21(17.7 cpu)
awk (retrieve program positional references)	17(10.9 cpu)
format	2:34(40.4 cpu)
format (output discarded)	46(20.8 cpu)
cat	1:48(5.6 cpu)
cat (output discarded)	6(3.2 cpu)

Table 1.

Times for various flat file primitives and UNIX commands on a file of 1923 lines, 101204 characters. Timings reported here are the mean from several runs. A large variation in real time occurred with system load.

Unfortunately, all times, especially the real time, varied under system load. Still, several observations can be made. First, the highly optimized "cat" command copies a file to the user's terminal at roughly 937 characters/second (real time), while the flat file primitive "format" displays a formatted version of the same file at 900 characters/second. In both cases, the system I/O speed, not the process speed limited the display speed (the terminal used for testing ran at 9600 baud). Second, the introduction of variables and assignments in the awk program during retrieval produced a measurable delay in processing. The average real time required to process a 1900 line file increased from 17 to 22

seconds (29%), but very few users notice any difference. Third, the time required for the shell to parse the script, redirect the input and output, and start execution remained very small. We conclude that rewriting the primitives in a lower level language would produce little or no benefit to the user.

7. Failures of Ffg

So far, the primitives-based implementation of ffg has been described in glowing terms. But the primitives approach has its limitations as well. Problems can be separated into three main categories: error detection problems, optimization problems, and error propagation problems.

Some of the error detection problems in ffg are inherent in the approach, others arise from the implementation. Because a primitive cannot know what other primitives precede it or succeed it in a pipeline, detection of some errors is impossible. For example, typing:

```
format | update
```

will replace the entire database with a formatted version because update cannot validate its input. Of course, it could detect badly misformed files like those without separator characters, but it cannot know what the user had in mind. On the other hand, syntactic errors in expressions given as an argument to the retrieve primitive go undetected until awk scans it. These errors could be detected earlier (and with better diagnostics) by a better implementation.

One of the most obnoxious errors that goes totally undetected occurs in:

```
retrieve name="Comer"
```

which retrieves no records from the file because the shell strips off the quote marks and awk assumes *Comer* is an uninitialized variable equal to the null string. One must escape quotes:

```
retrieve name==\"Comer\"
```

or put the entire expression in single quotes. The syntax is awkward at best; it is an example of the sort of handicap inherent in a system based on primitives.

Error recovery is difficult in any system; a pipeline only makes the problem harder. Consider the pipeline:

```
sortby error | retrieve expression | update
```

where the argument to sortby contains an error. What should sortby do? If it complains about the error and gives up, retrieve, the next program in the pipe, will receive an empty input file, as will update. Thus, a simple error in the argument results in an empty database. If, on the other hand, sortby complains about the error and proceeds to copy the entire database to its output, the information may be preserved, but left in an unexpected state. Clearly, a mechanism to inform pipe participants of error conditions is needed.

Computations expressed as a composition of primitives cannot always be optimized. Consider the simple example:

```
sortby name | retrieve salary==20000 | format
```

which sorts the entire database by name and then retrieves those records with salary equal to 20000. If the database is large compared to the number of records with salary equal to 20000, rearranging the pipe into:

```
retrieve salary==20000 | sortby name | format
```

might drastically reduce processing requirements, but there is no way to make such an optimization if the system handling the composition does not know about the retrieve and sortby primitives.

Finally, constructing programs out of existing parts may severely limit what one can express easily. Initially, plans called for a second type of database system, one that allowed records to span multiple lines. It turned out to be fairly difficult because awk could not recognize such a format. If awk had supported the notion of a "record separator" as it did the notion of "field separator", the task would have been trivial.

8. Conclusions

The flat file experiment proved successful in three ways. It provided concrete experience in the design of a set of primitives, it provided experience using UNIX primitives for a moderately sophisticated system, and it produced useful software along the way.

Implementing the fig primitives out of existing UNIX programs dramatically increased programmer productivity. Because the system consisted of small programs, a given primitive could be changed without affecting the others. And because primitives were constructed from existing programs, they were easier to change. Experimentation became feasible. New proposals were actually implemented, tested, and revised. As a result of more time for design and testing, the system turned out to please more users.

We conclude that creating programming as a set of independent primitives is a viable alternative to the current style of creating large programs from scratch. It increases programmer productivity by raising the level of the language in which one composes programs. To succeed, such an approach requires: a library of correct, reliable programs, an efficient mechanism to interconnect them, and experimental validation that a set of primitives is complete, uniform, consistent, and easy to use.

More research is needed to develop an adequate model and implementation

of error detection and recovery for programs connected in a pipeline.

References

[AHKW79] A. Aho, B. Kernighan, and J. Weinberger, "Awk -- A Pattern Scanning and Processing Language," *Software Practice and Experience*, 9:4 (April 1979), 267-279.

[BAKE72] F. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal* 11:1 (1972), 56-73.

[BOGS79] B. Borden, R. S. Gains, and N. Shapiro, "The MH Message Handling System: User's Manual," Technical Report R-2367-AF, Rand Corporation, November, 1979.

[BOUR78] S. Bourne, "The UNIX Shell," *Bell System Technical Journal* 57:6 Part 2 (July-August 1978), 1971-1990.

[DATE75] C. Date, *An Introduction to Database Systems*, Addison Wesley, 1975.

[HANS79] D. Hanson, "Software Tools Programmer's Manual," Technical Report TR79-15, University of Arizona, 1979.

[KEMA79] B. Kernighan and J. Mashey, "The UNIX Programming Environment," *Software Practice and Experience*, 9:1 (January 1979), 1-15.

[KEPL76] B. Kernighan and P. Plauger, *Software Tools*, Addison Wesley, 1976.

[RITH78] D. Ritchie and K. Thompson, "The UNIX Time Sharing System," *The Bell System Technical Journal* 57:6 Part 2 (July-August) 1978, 1905-1930.

[ULLM80] J. Ullman, *Principles of Database Systems*, Computer Science Press, 1980.

Appendix A: A Summary of FFG Commands

delete *Booleanexpression*

Delete writes out all records which do *not* satisfy the given *Booleanexpression* (i.e., it deletes those records that do satisfy the *Boolean-expression*).

edit [*−e editor*]

Edit invokes the user's default editor (*editor* if **−e** is specified) on the database. Only update and edit change the data.

enter

Enter is an interactive program used to create records for the database. The most common use is *enter|update -a* which adds the new records to the old ones.

fielded *Booleanexpression Fieldassignments*

Fielded writes a copy of the database in which records that match the *Booleanexpression* have modifications as specified in the *fieldassignments*. The *fieldassignments* consist of one or more assignment statements separated by semicolons. Each assignment is of the form *fieldname=expression*.

format [*−fmt*]

Without a parameter, format prints its input in columns; with parameter *X*, format prints input according to format file *Specs/X.fmt*

lookup [*−fieldnameprefix*] *pattern ...*

Lookup is a shorthand for simple retrievals; it writes on its output all those records which match the *pattern*. If *−fieldnameprefix* is specified before a pattern then the pattern match will be restricted to that field. The prefix must be unambiguous. Lookup matches each pattern against the database in turn, so records that match more than one pattern will be written in the output more than once.

rebuild [*commandname*]

Rebuild will reconstruct commands after a change to the field specification file. Without an argument, rebuild will reconstruct all commands.

retrieve *Booleanexpression*

Retrieve writes on its output those records for which the *Booleanexpression* holds.

status [*options...*]

Status reports the status of the mutual exclusion lock, database and backup files, users accessing the database, and exclusive use lock. It can also be used to alter the status of locks, user access, or to force a backup. Under normal circumstances, the status command is not required; it is intended to help users clean up lock files after a system crash or other abnormal process termination. In general, lower case arguments request status information, while uppercase arguments alter the status. The *options* include:

−b	– show **b**ackup file status
−d	– show **d**atabase file status
−e	– show **e**xclusive access status
−l	– show database **l**ock file status
−s	– suppress detail in output
−u	– show status of active **u**sers
−B	– force a **B**ackup
−E	– request **E**xclusive use access
−K n	– **K**ill user with process id n (−u lists process ids)
−L	– **L**ock the database
−M	– return to **M**ultiuser access
−R	– **R**eset (i.e. -K) all user access
−U	– **U**nlock the database

showfields
Showfields prints the current field description information.

sortby [*−key*] *fieldnameprefix* ...
Sortby writes a sorted version of the database using the list of *fieldnameprefixes* to determine the sort(s) to be done. The first *fieldname* gives the primary sort field. Optional *keys* may be any of the characters **bdfinr** as in the UNIX *sort*(1) command, which will override those in the field specifications file.

undo
Undo will restore the data file to its previous value. Only one backup is kept; there is no way to recover older copies.

update [*−a*]
Used at the end of a pipe, update writes its input to the database; parameter *−a* means append. Only update and edit actually change the data.

verify [*conditionname* | *Booleanexpression*] ...
Verify checks the file to make sure all records adhere to the named condition. *Conditionnames* are given in the verification file; unrecognized strings are taken to be literal conditions.

Appendix B: Boolean Expressions in flat files

```
<Boolean-exp>     ->  <Expression> || <Expression>
                  ->  <Expression> && <Expression>
                  ->  ! <Expression>
                  ->  <Expression>

<Expression>      ->  <Comparison>
                  ->  <Pattern match>

<Comparison>      ->  <Term> == <Term>
                  ->  <Term> != <Term>
                  ->  <Term> <  <Term>
                  ->  <Term> <= <Term>
                  ->  <Term> >  <Term>
                  ->  <Term> >= <Term>
                  ->  <Term>

<Pattern match>   ->  <Term> ~ /<Pattern>/      (does match)
                  ->  <Term> !~ /<Pattern>/     (does not match)

<Pattern>         ->  pattern                   (UNIX ed pattern syntax)

<Term>            ->  <Primary> + <Primary>
                  ->  <Primary> - <Primary>
                  ->  <Primary>  <Primary>      (concatenation)
                  ->  <Primary>

<Primary>         ->  <Factor> * <Factor>
                  ->  <Factor> / <Factor>
                  ->  <Factor> % <Factor>       (mod)
                  ->  <Factor>

<Factor>          ->  fieldname
                  ->  numeric value
                  ->  "any string of characters"
                  ->  NR
                  ->  NF
                  ->  length( <Term> )          (length of a string)
                  ->  substr( <Term> , start, len)  (substring function)
                  ->  log( <Term> )
                  ->  sqrt( <Term> )
                  ->  int( <Term> )             (truncate)
```

Notes:

1. Concatenation is allowed between objects of any type. In particular, numeric values may be concatenated onto strings.
2. NR is always the number of the record being examined, and NF is always the number of fields in the record being examined.