A TABLE-DRIVEN DATA VALIDATOR

Marc J. Rochkind

Bell Laboratories
Whippany, NJ 07981

Summary

Data validation and editing are performed by most application systems. This paper describes a table-driven validator that is intended to be part of an inventory of off-the-shelf components for application building. It is particularly advantageous in distributed systems, in which uniform data validation criteria are both necessary to have and difficult to obtain.

Introduction

Most computer programs that read data subject it to validation to check for required data items, proper formatting, correct character set, and so on. While in the case of one-shot or toy programs the validation may be trivial, more often considerable thought must be given to the validation strategy. If the data comes directly from a human, in a transaction-processing or data-entry application, for example, the validation and the error messages that result play an important role in the usability and learnability of the system. If the data comes from a mechanized source, validation must be strict enough to maintain the quality of the database being updated, but not so strict as to generate thousands of extraneous messages.

In a distributed system, responsibility for the reliability and correctness of each node may also be distributed. Each node must not assume that incoming data is valid, so some validation must be performed each time data is transferred between nodes. On the other hand, if validity requirements among the nodes are even slightly at variance, the system will not run smoothly.

At Bell Laboratories, we have developed a table-driven validator as a part of a continuing effort to produce off-the-shelf components that can be used to build application systems. Some of these components, in various stages of completion, are a forms (or mask) package for CRT terminals, a report generation language, a database management system and a high-level language for writing transaction-processing programs. Only the validator will be described here.

While a program that validates its input might have the validation tests spread among the other processing it performs, it is necessary for a program that is a candidate for using the validator to be organized along functional lines, roughly as follows:

```
while (moreinput) {
        rcd = getinput();
        if (not valid(rcd, msglist))
                print msglist;
        else
                process(rcd);
}
```

In this example "rcd" is an abstract data type, called a *record*, that represents a collection of data items. Conceptually, it is a simple table of field names and values:

NAME	Mary Smith
ADDR	123 Main St
SAL	28400
.	.
.	.
.	.

We assume that there is a set of functions that can be used to manipulate *records*; at the very least, something like

```
change(rcd, NAME, "Jane Doe")
```

which changes the value of a field, and something like

```
v = value(rcd, ADDR)
```

which retrieves the value of a field.*

One could, of course, program the function "valid" conventionally. This may be tedious, but hardly difficult. The tough part is likely to be determining what validation should be done, and the validator provides no help there (except to reduce the cost of being wrong). The table-driven approach does, however, have these advantages over the hard-coded approach:

* In some programming languages, one can define abstract data types and abstract operations directly. In others, such as C [1], one must provide a library of functions that provide the necessary information hiding and convenience.

Reprinted from PROCEEDINGS OF COMPCON FALL, September 1980

- The language used to code the validation table is easier to learn and use than a programming language. The task of coding this table can be—and has been—assigned to non-programmers.
- The validator can be used in systems where the purchaser of the system is allowed to change the validations. The system's vendor might be unwilling to let the customer change the programs themselves, since that would make customer support unbearably difficult.
- As a previously existing, well-used piece of software, the validator is probably more reliable that the brand-new programs that comprise the rest of the application system. This makes the debugging and testing of the system that much easier. Debugging and testing the validation table is inherently easy—the worst that can go wrong is incorrect validation, and, if the validation is incorrect, one knows immediately where to look.
- The validation table serves as a formal specification of what validations are performed by a system. This is useful in enforcing standards and in interfacing one system to another. Such a formal specification is of considerable help in inplementing distributed systems, because it makes it easy to ensure that each node has the same concepts of "valid" and "invalid."

In the remainder of this paper we will describe the validation language used to code the table and give a brief sketch of the implementation.

Validation Language

Overview

To use the validator, one must first code the validation conditions using a special-purpose validation language. The resulting validation table is compiled into instructions for a software machine. At execution time when the function "valid" is called, the machine executes these instructions to generate the (possibly empty) list of error messages.

The validation table consists of two columns: the first contains a series of conditions similar to expressions in a programming language; the second column contains an error code that is generated if the condition is false. For example:

```
NAME % "[A-Za-z]{4,25}"           badname
ADDR != ""                        badaddr
SAL > 10000 & SAL < 75000         badsal
```

The first condition requires the value of the NAME field to match the pattern within quotes. The pattern matches if the value consists of between 4 and 25 alphabetic characters. If it doesn't match, the error code "badname" is generated. The second condition just requires the ADDR field to be present (that is, not equal to the null string). The third condition requires the value of the SAL field to be in the range 10000 to 75000—supposedly, anything else is an unreasonable salary.

In the case of the last condition, it would be better to check that SAL is numeric before checking its range. To do this, one may indent a condition under another condition:

```
SAL % "[0-9]{1,6}"                  badsal1
      SAL > 10000 & SAL < 75000     badsal2
```

Now error code "badsal1" is generated if SAL is not from 1 to 6 digits. The indented range check is performed only if the first test passes; if SAL is out of range, error code "badsal2" is generated.

The validator translates the generated error codes to English messages that can be printed by its caller. It does this be referring to a table of error codes. For example:

badname	NAME must be 4 to 25 alpha characters.
badaddr	ADDR is a required field.
badsal1	SAL must be numeric.
badsal2	SAL is out of range.

About two dozen operators and several built-in functions can be used to code conditions. Parentheses may be used to any depth. Conditions may be indented to a depth of 9 (more than enough). There may be hundreds of conditions, up to the limit of memory to hold the compiled instructions for the machine.

The next few sections describe the language used to code the conditions in more detail.

Operands

Operands may be field names, numbers, strings, regular expressions and function calls.

Field names may be whatever is legal for the *record* data type. The validation compiler accesses a table of allowable field names and translates the names to an internal code that is used by the *record* manipulation functions.

Numbers must be integers, in the range −999999999 through 999999999.

Strings are quoted and may contain any ASCII character except NUL.

Regular expressions are patterns using a notation taken from the UNIX* [2, 4] text editor. They will not be described here, but the examples indicate the facilities available. More details may be found in [5].

Functions are used for operations not handled by the operators. Some examples are:

* UNIX is a Trademark of Bell Laboratories.

substr	Obtain a substring.
size	Get the size of a string.
dump	Print the *record* being validated.
search	A simple facility for searching tables (such as a table of state names).
chg	Change the value of a field. This function makes the validator an editor as well. It can be used to supply defaults, translate symbols to numeric codes, combine fields, break down fields, etc.

Operators

Operators exist to do comparison, arithmetic, pattern matching, logical combination and concatenation.

The *comparison* operators are divided into those for numeric comparison and those for lexical comparison. The former operators treat their operands as numbers; the latter ones treat their operands as strings. The numeric ones are:

```
#==   #!=   <   <=   >   >=
```

The first two are for equal and not equal; the rest are self-explanatory. The corresponding lexical operators are:

```
==   !=   $<   $<=   $>   $>=
```

This approach to comparison operators is admittedly ugly, but the more streamlined alternatives seemed too full of surprises (is "2" equal to "02"?).

The *arithmetic* operators are the usual ones:

```
+   -   *   /
```

The *matching* operators are "%" and "%%". The first is used to match a pattern against an entire string; the second treats the string as a sequence of blank-separated words, and requires the pattern to match each word. It is useful in cases where one requires a field to be capitalized words, for example:

```
NAME %% "[A-Z][a-z]*"                 badname
```

Pattern matching is probably the most valuable feature of the validator, since it exists in none of the programming languages commonly used for application programming (PL/I, COBOL, etc.).

The *logical* operators are "|" (or), "&" (and) and "!" (not).

Finally, the only *string* operator is "$" for concatenation.

Implementation

The validation compiler was built with the compiler-building tool *yacc* [3]. It reads the validation table from a UNIX file and generates the instructions for the software machine on another file. To illustrate, the line

```
SAL > 10000 & SAL < 75000             badsal
```

would generate these instructions:

```
FIELD   SAL
PUSHN   =10000
GT
FIELD   SAL
PUSHN   =75000
LT
AND
ERR     "badsal"
```

The validation machine is a software stack-machine that reads the instructions into memory at execution time and executes them each time it receives a *record*. The instruction FIELD pushes the value of its operand field onto the stack. The instruction ERR tests the top of the stack; if it is zero, it generates the error code given as its operand. The other instructions are self-explanatory.

Both the compiler and the machine were developed originally on UNIX, and it would be difficult to move the compiler to another operating system, since it makes liberal use of system calls rarely found on other systems. The machine, however, is mostly a giant "case" statement that could easily be ported to another environment that has a C compiler, or even rewritten, in PL/I, say. If this were done, one could use UNIX to compile the table, and move the compiler output to the system on which the application runs.

Experience

The validator was developed in 1976 for a system that produced telephone books. Since then, it has been used in a system that automates telephone repair bureaus, and in several data-entry applications. The fact that it has no built-in field names that tie it to a particular application and the power of its expressions make it quite universal. Its use has encouraged application system designers to do more validation and to provide more specific messages than they would otherwise. For example, the telephone repair bureau system had one primary online error message: "FIELD IN ERROR". With the validator, it now can generate dozens of different messages, providing the repair service attendant with more guidance. The editing features (the *chg* function) have also simplified the transaction-processing programs somewhat.

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N. J.: Prentice-Hall, 1978.

[2] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *CACM,* 17, 7 (July 1974), 365-75.

[3] S. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).

[4] *The Bell System Technical Journal,* 57, 6, Part 2 (July-August 1978), 1897-2312.

[5] B. W. Kernighan and P. J. Plauger, *Software Tools,* Reading, Mass.: Addison-Wesley, 1976.