the necessary coordinates to locate the unit in three dimensions. The additional variables for the features include all square/levels the feature may have extended into as determined through excavation.

The means of ariving at the stored data can best be shown by the file of excavated squares. Each record consists of the above 15-core variables arranged into 215 components including the directory, occupying less than 375,000 bytes with all data as integers for efficiency in storage.

Since the data are entered in two modes—one for counts and weights, the other for square/level dimensions and locational information—provenience and program instruction take a rather high proportion (15 percent) of the keyboard input. The level dimensions themselves are 36 percent of the input, the counts and weights are 21 percent, and single blanks separating each entry accounts for an unexpected 28 percent. On a system with fixed-format input and the same readability of input data, the percentage of blanks would be at least double, an important consideration in keeping connect time with the computer to a minimum.

System Performance

The computerized file processing and data retrieval worked successfully in the summers of 1972 and 1973 in processing a file composed of the level records of each square. This file was kept up-to-date to the very end, and was complete the day following the closing of the excavation. We found that the time-sharing system was rarely down. There were negligible queuing delays, and the weakest link proved to be the telephone lines.

A few statistics will prove useful in evaluating the speed of our routines. The updating (on an IBM 370/145) took 120 minutes of elapsed time and one minute of computer time for 60 records of 60 bytes each.

Retrieval on an IBM 370/158 used correctly is quite acceptable: for a single parameter search from a file of 2,097 records of 60 bytes each in 91 components, 54 sec of elapsed time and 5.4 sec of central processor time were used prior to printing the retrieved records. Because of the nature of APL's array processing, multiple parameter searches add a negligible percentage to the central processor time.

Conclusion

The viability of this field experiment indicates that the potential of computers for solving the data management problems of large scale multi-disciplinary archaeological programs is strong.

Short Communications
Programming Techniques

A Simple Technique for Representing Strings in Fortran IV

David R. Hanson
The University of Arizona

Key Words and Phrases: string processing, Fortran IV, string representation, structured programming, data structures
CR Categories: 3.70, 4.0, 4.19, 4.9

FORTRAN IV is one of the most widely used programming languages, and programs written in FORTRAN tend to be quite portable. Although the language has, at best, only primitive string-handling capabilities, many programs that deal primarily with strings are sometimes written in FORTRAN simply to achieve a degree of portability. It is the purpose of this note to illustrate a very simple technique for structuring and manipulating varying-length string data in FORTRAN IV [3]. The scheme is conducive to such operations as concatenation, substring extraction and duplication.

There are many books on FORTRAN IV programming [1, 4–5], but none treats variable-length strings in detail. Both Harrison [4] and Day [1] describe several methods for handling strings in FORTRAN, but their methods do not lead to the heuristics for string operations that exist in our scheme. McKeeman [6] has used a technique similar to ours in XPL a dialect of PL/I.

Two pieces of information characterize varying-length strings—their location in memory and their length. In languages such as SNOBOL4 [2] this information is usually contained in a descriptor. We may implement a similar descriptor containing the location and size of a string in FORTRAN by perverting the use of complex variables. The actual characters of the string are stored in an array set aside for that purpose. Using this descriptor scheme, it is not important whether the characters are stored in packed or unpacked format.

We choose type COMPLEX for string descriptors since

Results of this study were presented at the 1974 Computer Science Conference in Detroit, Michigan, February 14, 1974.
Author's address: University Computer Center, The University of Arizona, Tucson, AZ 85721.

one complex variable contains two pieces of information. In order to make efficient use of the values of the location and size of a string, we store them as integers in a complex variable. This can be effected by an EQUIVALENCE statement.

```
COMPLEX D
INTEGER ID(2), OFF, LEN
EQUIVALENCE (D,ID(1)), (ID(1),OFF), (ID(2),LEN)
```

The location field, or first word of the complex variable, may be referenced by referencing the integer variable *OFF*. Likewise, the second word is referenced via the integer *LEN*. The complete string descriptor, {*loc*,*size*} is referenced via the complex variable *D*. Using this method, a string may be passed to a function or subroutine as a single complex argument. Furthermore, a function may return a descriptor for a string as its value. As mentioned above, the actual characters which constitute the string are stored in an array. We simply maintain a character index, *avail*, that points to the next available character location. When a new string is inserted into the string storage array *avail* is incremented by its length.

One advantage of the descriptor representation is that strings can "overlap" since there are no special markers embedded in the storage array. Suppose the string *THIS STRING* is represented by the descriptor $\{\alpha,11\}$. The string *STRING* is represented by the descriptor $\{\alpha+5,6\}$, sharing some of the characters used by the other string. This makes operations such as substring extraction trivial—we only need to return a modified descriptor. One can easily implement a function *SUBSTR*(*S*,*I*,*L*) that returns a descriptor representing the substring of *S* beginning at the *I*th character which is *L* characters long. Note that *S* and *SUBSTR* are type *COMPLEX*, but *I* and *L* are type *INTEGER*.

The descriptor representation allows the introduction of heuristics that help make some string operations more efficient. For example, consider a function *CONCAT*(*S*1,*S*2) which concatenates the strings represented by the descriptors *S*1 and *S*2 and returns the appropriate descriptor. At first glance, one might just copy the characters of the strings represented by *S*1 and *S*2 to the "top" or free portion of the string storage array and return the appropriate descriptor. But using the descriptor representation, there are three heuristics that may greatly affect the speed of concatenation.

1. If either *S*1 or *S*2 represents the null string, {0,0}, then *CONCAT* is just the other descriptor.
2. If *S*1 represents a string that is at the "top" of the string storage array, only the characters of the string represented by *S*2 must be copied. In other words, if $loc(S1)+size(S1)=avail$, then we copy only the second string and return the descriptor $\{loc(S1),size(S1)+size(S2)\}$.
3. Finally, if the strings represented by *S*1 and *S*2 are already adjacent, we need do no character movement but just return the appropriate descriptor. That is, if $loc(S1)+size(S1)=loc(S2)$, *CONCAT* is the same descriptor as in (2) above. This heuristic is often overlooked, as in [6].

If none of the above heuristics succeeds, we are forced to copy the characters of both strings. Heuristics may be determined for a multitude of other common string operations such as duplication, lexical comparison, identity, and reversal.

The fact that a function can effectively return a string as its value and that a string may be passed to a function or subroutine as a single argument leads to more readable and organized programs in a language that is already notorious for badly structured programs. For example, a statement such as

```
S = CONCAT(SUBSTR(S1,1,4),SUBSTR(S2,3,2))
```

is significantly more readable and self-documenting than a series of *CALL* statements containing numerous repetitive arguments.

The method has three inconveniences: (1) additional storage is required for the descriptors and the machine code necessary to perform double-word data movement; (2) the variables used for descriptors and string functions must be declared *COMPLEX*; and (3) the printing of a string is somewhat cumbersome if the characters are stored in packed format. Despite these inconveniences, a system of string functions using this technique has been successfully implemented on the IBM/360, CDC 6000 series and DECsystem-10 computers. On machines such as the CDC 3300 or IBM 1130 whose FORTRAN compilers allocate two words for floating point numbers, a descriptor may be represented by a *REAL* variable. It should be noted that the use of complex variables in this fashion is in violation of the ANSI standards [7].

Received February 1974, revised April 1974

References

1. Day, A.C. *FORTRAN Techniques*. Cambridge U. Press, Oxford, England, 1972.
2. Griswold, R.E. *The Macro Implementation of SNOBOL4*. W.H. Freeman, San Francisco, Calif., 1972.
3. Hanson, D.R. Descriptor representation of strings and other objects in FORTRAN IV. 1974 Comput. Sci. Conf., Program Abstracts, 87, Feb. 1974.
4. Harrison, M.C. *Data-Structures and Programming*. Courant Inst. of Math. Sci., New York, 1970, Chs. 3 and 4.
5. McCracken, D.D. *A Guide to FORTRAN IV Programming*. Wiley, New York, 1972.
6. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970, Ch. 8 and Appendix 4.
7. USA Standards Institute. Fortran. USAS X3.9, 1966.