THE INTERPRETATION OF STRUCTURED STORED DATA USING DELIMITERS

E. H. BEITZ

CONTROL DATA CORPORATION

JUNE 1970

THE INTERPRETATION OF STRUCTURED STORED DATA USING DELIMITERS.

The paper presented below puts forward a method of delimiting strings of data with dedicated codes from a specific alphabet. The purpose of the delimiters is to permit both variable length data and variable occurrence of data-items in a particular context. The advantages of using such a scheme are felt to be numerous. The most important of the advantages are the reduction in storage requirements and the flexibility in data-item representation.

The terminology used in this paper is consistent with that of the COBOL language. More precise definition of the terms used is not necessary and makes it possible to avoid the ontological and other philosophical problems of describing data. The representation of the various data types is also of little consequence and will not be discussed either. The technique requires that data descriptive information be available at run-time rather than at compile-time.

A data description such as the Data Division entries of COBOL serves two primary purposes. The first is to assign a data-name to an item and to list the attributes for its associated values. The second is to prescribe an order to the values and to show their hierarchical relationships.

Unfortunately the representations of the different values do not always have the same storage requirements. Even a set of values associated with a particular data-name will have large disparities between the storage requirements of its members. Equally disarming is the fact that the number of occurrences of the same item varies from record to record. COBOL makes provisions for solving both of these problems, but the cost is storage. (Some data description techniques eliminate the problem by not even trying to solve it.)

Let us consider that the values, which all data-names may take, belong to one of two types. The first is simply a string of codes from a specific alphabet. The second type is any specific representation which uses a finite number of bits to represent each and every possible value for a specific elementary item. The former type is characterized by the variable length alphanumeric string and the latter by a floating point number.

For the purpose of discussion we will assume a particular alphabet of codes for the first data type, say the 8-bit ASCII code-set. Four of the ASCII codes have been assigned as delimiters. They bear a specific relationship to each other; namely they have a definite hierarchy. The names which they have been given may be confusing in the context of this paper so we will give them some simple symbolic representation. Only three will be used here and will be:

GS {ASCII Group Separator} will be	√
RS {ASCII Record Separator} will be	|
US {ASCII Unit Separator} will be	•

This small table is ordered having the most exclusive delimiter at the top and the least exclusive one at the bottom.

We will look at a technique, which uses the delimiters and the data types described above. This technique will allow variable length strings and variable occurrence of both elementary items and group items. Our data description must include the size of values associated with the second data type. (The problem solution would be simplified if this data type had a size which was some multiple of the size of each of the symbols used for the first data type.)

Consider a simple record which consists only of elementary items. A partial COBOL description of this record follows:

```
01  SAMPLE—RECORD1
  02  A
  02  B
  02  C
```

All three data-names are elementary items and let us assume that all may take alphanumeric strings as values. Suppose that each data-name may take no values, or one value, or many values. To separate one alphanumeric string from the next, we will use our delimiters.

Only three of the delimiters will be used. The delimiter associated with a data-name will follow the value associated with that same data-name. The delimiter itself will control the context in which the string following it will be considered. The meaning ascribed to the three delimiters will be:

•	Terminates all but the last value for the current data name.
|	Terminates the last value for the current data-name.
√	Terminates the record.

Using the description for SAMPLE_RECORD1 and the above definitions, let us discuss some examples. For the purpose of illustration we will show an alphanumeric string representing a value for the data-name A as the small letter a

Example 1: **a | b | c √**

Here we see a single value for each of the three elementary items.

Example 2: **| b • b √**

Data name A has no value and data-name B has two values. The record terminates before we consider data-name C so it is assumed to have no value.

Example 3: **a • a • a • a | | c √**

Example 4: **| b • b | c • c • c √**

Some observations about the data description as it relates to the records and the data-names are in order. Examples 2 and 4 both show no value for data-name A. Example 2 shows premature termination. A delimiter is required for all non-existent item values preceding the last extant item value. Non-existent item values are implied for all items following premature termination. This suggests that data-names for mandatory items, those which must take at least one value, should precede those for optional items in the data description. The position of a data-name in the data description should be a function of the probability of that data-name taking a value in any record. (The grouping of elementary items does not always make this possible.)

The hierarchy of data-items as represented by their level numbers in the data description will now be discussed. The relationship between the level number of an item and that of the item following it serves to indicate whether the first item is an elementary item or a group item. A quick look at a number of examples will illustrate this.

Example 1:

```
05   CURRENT—ITEM
05   SUCCESSOR
```

The level number of `SUCCESSOR` is equal to the level number of `CURRENT-ITEM`. `CURRENT-ITEM` is “elementary.”

Example 2:

```
        05   CURRENT—ITEM
          06   SUCCESSOR
```

The level number of **SUCCESSOR** is greater than the level number of **CURRENT-ITEM**. **CURRENT-ITEM** is “group.”

Example 3:

```
        05   CURRENT—ITEM
      04   SUCCESSOR
```

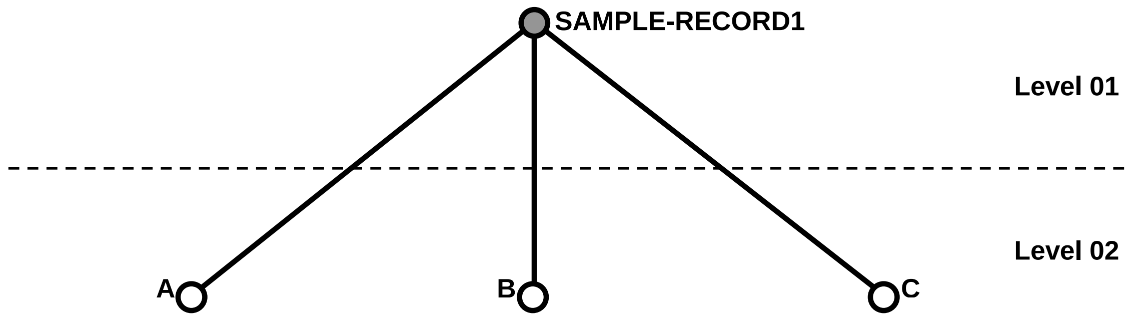
The level number of **SUCCESSOR** is less than the level number of **CURRENT-ITEM**. **CURRENT-ITEM** is “elementary.”

It is possible to make a further deduction from the relationships illustrated in the above examples. That is, we can determine which item will be considered when **CURRENT-ITEM** has been completely dealt with. But this requires some elaboration. Using the same record as the one used above to illustrate the use of delimiters, we will show how the order in which the data-items are considered may be prescribed.

The partial COBOL data description for this sample record is:

```
01   SAMPLE—RECORD1
  02   A
  02   B
  02   C
```

This same record may be shown as a hierarchical tree.

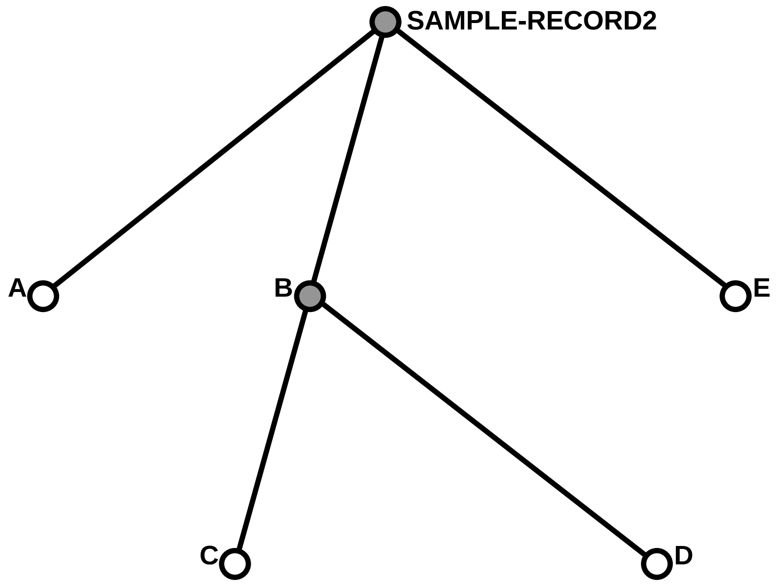


Elementary items are the leaves of this inverted tree (and will be represented by unfilled circles). With the exception of the node at Level 01 which must be part of every record in a file, other branching points must be group items.

Consider a slightly more complex example:

```
01   SAMPLE—RECORD2
  02   A
  02   B
    03   C
    03   D
  02   E
```

and its tree representation:



The node B is not a leaf and is therefore a group item. The leaves C and D are the sole members of the group of B. We will assume that all occurrences of the group B must be dealt with before the leaf E is considered.

Let us consider a flag which indicates the existence or absence of the group B. There are only two specific instances in which the flag would be needed in SAMPLE-RECORD2. The first is when all the values for the elementary item A have been processed. The second is after all the members of the group B have been dealt with and we wish to determine whether or not there is another occurrence of B prior to considering the elementary item E.

A group item does not really have its own value. Collectively the values of all the elementary items contained in the group comprise its “value.” This does not preclude having a delimiter for the group data-name. We have seen that it is possible to deduce whether the data-name with which we are working is an elementary item or a group item. We can use a delimiter associated with the group data-name to tell us if the group is present. If it is present then we simply move to the next data-name in the data-description be it elementary or group. When the group is absent then the first item following the entire group will be considered.

We can use the same delimiters as before. This is only possible because we know the context of their usage. Only two delimiters are required for group data-names. Let us assign the following meaning to the delimiters “•” and “|” when used to indicate whether or not a group is present:

•	The group, whose data-name is being considered, is present.
|	The group, whose data-name is being considered, is absent.

The following examples illustrate the use of the group delimiters. The record description is:

```
01   SAMPLE—RECORD2
  02   A
  02   B
     03   C
     03   D
  02   E
```

Example l: **a • a | | e • e √**

Here the second "|" tells us that the group B is absent; so what follows this delimiter must be a value for the elementary item E.

Example 2: **a | • c | d | | e √**

The "•" indicates that the group B is present and the fourth "|" indicates that there is no second occurrence of the group B.

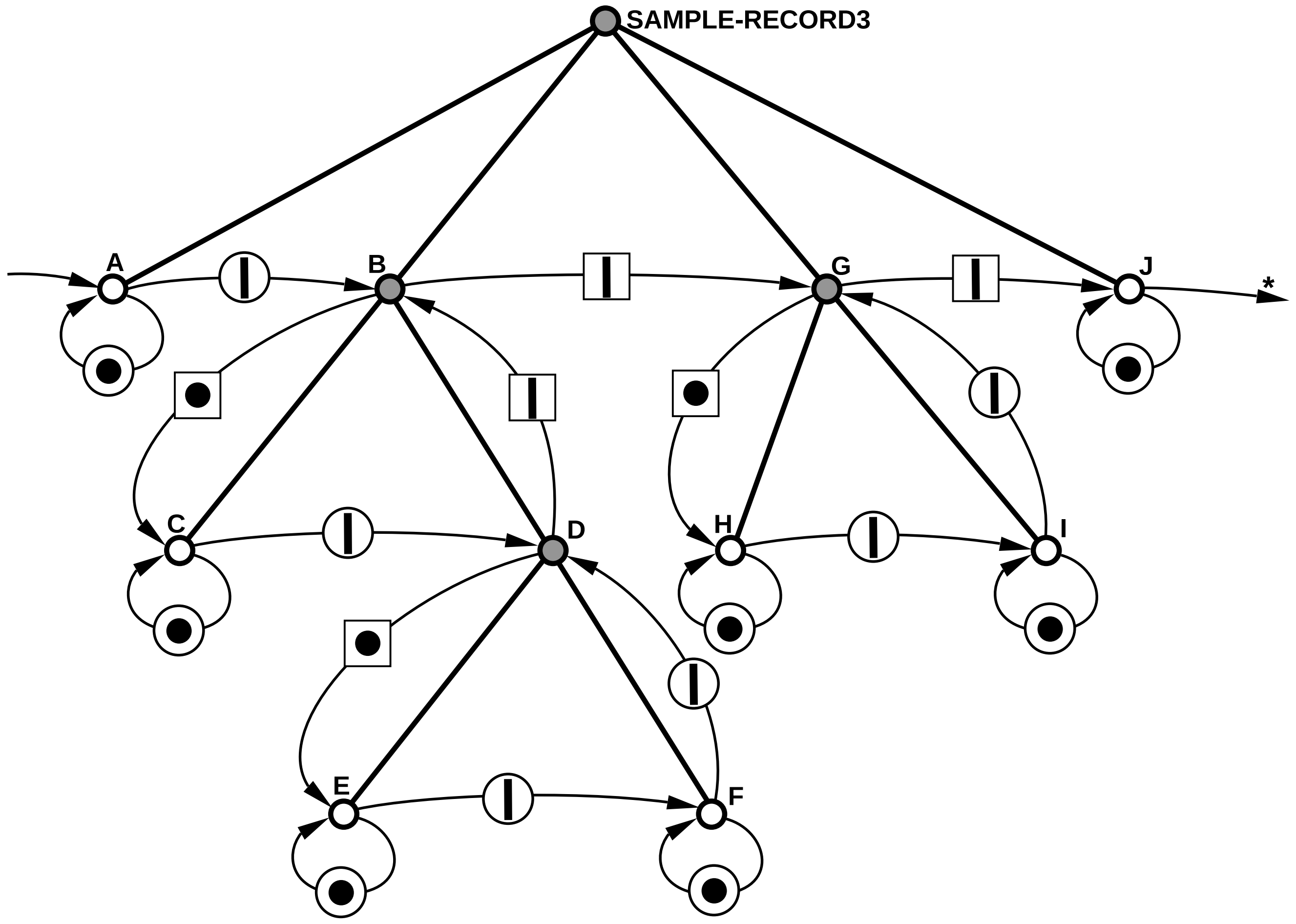
Example 3: **a • a | • c • c | | • c | | • d | | e √**

A COBOL-like data-description and the tree representation for a much more complex record are shown below. A fine line, which shows the processing flow, is included in the inverted tree representation. When the delimiter being considered belongs to a group item it is enclosed in a square. In all other cases the delimiters are associated with values and are in circles.

The data description:

```
01   SAMPLE—RECORD3
  02   A
  02   B
    03   C
    03   D
      04   E
      04   F
  02   G
    03   H
    03   I
  02   J
```

the tree representation:



Premature termination indicated by encountering the delimiter "√" will simply jump to the end of the flow at *. All data-items not dealt with are considered to have no values.

A few examples using the data-description shown above follow:

Example 1: **a | | • h • h | i • i • i | | j • j • j √**

This example shows the group B absent and a single occurrence of the group G.

Example 2: **a | • c | | • c • c | • e | f • f | • e | | | | • h | i | | j √**

Here we see two occurrences of the group B and one of the group G. In the first instance of group B the sub-group D is absent. In the second occurrence of B there are two occurrences of the sub-group D.

Example 3: **| | • h • h • h • h √**

Both A and B take no values, and this record terminates prematurely.

The technique described above will result in a significant saving of storage resources. Just how much storage is very difficult to say but is a function of the structural redundancy and complexity of the file being considered. The delimiters do have storage requirements but this is insignificant when compared with the potential reductions in data storage needs. The group item delimiters reduce the delimiter storage needs considerably.

The delimiter structure described here can be used in conjunction with many different structural forms. The hierarchical structure was chosen as the model in this paper since it is the most commonly used. Explicit information about data-items should be included in the data-description and should not be deduced when and as needed. For example, group items should be labeled as such in the data description; non-recurring data-items should have indicators to this effect; mandatory data-items should be marked "mandatory." This will not only speed up processing but will aid in detecting errors on input and during processing.

The delimited string used together with the run-time data-description is felt to be a realistic and flexible approach to the problem of storing large volumes of data. There is a move to on-line inquiry systems in data processing. Run-time data descriptions and the scheme described here meet the needs of the databases maintained for such systems. This technique is proffered in the belief that we must make our data-description fit the data and not try to make that data fit the data-description.