DATA STRUCTURES THAT GENERALIZE RECTANGULAR ARRAYS

Samuel A. Hoffman *

Burroughs Corporation, Burroughs Laboratories
Paoli, Pennsylvania

Summary

Three problems associated with structured data sets -- description, allocation, and retrieval -- are briefly considered. The first is a problem in describing data to a compiler in a formal manner that will allow the compiler program to allocate storage and retrieve data when the data is referenced. The second is the problem of providing contiguous storage for the data set in such a way that when suitably described pieces of the data are required at run time, the positions of these pieces of data relative to some base will be determinable. The third problem is that of referencing or describing subsets of the data set. A class of data structures is defined. This class is shown to be a generalization of the class of structures which are representable as n-dimensional rectangular arrays. These structures are termed generalized structures; a formal method of describing these structures (by descriptors) is defined. The formal entities, called reference expressions, that describe the data to be retrieved from storage, are also defined for this class of structures. Finally, the appropriate form of the storage mapping function is derived. The storage mapping function is the mathematical expression which relates the description of an item of data to its position in memory. The manner in which this function is derived from the descriptor is shown. In every case, the work performed for generalized structures is shown to be a direct generalization of the corresponding considerations for rectangular arrays.

Finally, an ALGOL program for the Burroughs 220 computer is briefly described. The program simulates the actions that a compiler would take upon receiving a descriptor in forming the storage mapping function, and the actions that would be carried out at run time when a reference expression is presented.

Description, Allocation, and Retrieval

When a program is being written for a digital computer, the programmer or programming system must see to it that memory space is suitably allocated for the data to be processed by the program. This is true regardless of the language in which the program is being written. When the data is structured, the allocation should be made so that calls for the data can reflect this structure. Because computer memories are sequentially organized -- by words, characters, bits, etc. -- the allocation and retrieval problem has usually involved linearly mapping the data into memory.

When a compiler is being used by the programmer, there are three facets to the structured data problem.

1. The structure of the data must be supplied to the compiler program in a suitable symbolic form.

2. The compiler program must have a routine that allocates a contiguous portion of memory for each instance of the structured data set.

3. A means must be available for the programmer to refer to individual (or groups of) items of the stored data. These three facets may be characterized as description method, storage mapping function, reference expressions; the three facets are bound, respectively, to the more general problems of description, allocation, and retrieval.

The classical employment of structured data in computer programs has been that of n-dimensional rectangular arrays; that is, the data is considered to be indexed by n coordinates, where the allowable values of the ith index are 1, 2, ..., ai.

The description method for such a structure is simply given by the n-tuple of ai's (a1, a2, ..., an) and, for identification purposes, a preceding name. It should be recognized at this point that the n-tuple itself defines the structure in question. The preceding identifier can then be interpreted as a free variable for which substitution will be made when a particular data set (an instance of the structure) is being declared as having this structure. However, in this paper, structures are assumed to be defined only in the context of particular instances of those structures.

In any case, then, a name or identifier followed by an n-tuple of integers provides a complete description of the data set so named as an n-dimensional rectangular array. To refer to a particular element of a data set so structured, it is necessary only to prescribe another n-tuple of, say, x1's, where x1 is (an integer) less than or equal to ai. Thus, a name (the same as in the descriptor) followed by an n-tuple is the general form of the reference expression for such structures. The typical storage mapping function in these cases is a program which computes

\[ (x_n - 1) + \sum_{i=1}^{n-1} \left( \frac{n}{i+1} \right) (x_i - 1) \]

* Now with Kettelle & Wagner, Consultants, Paoli, Pennsylvania
The range of values for this expression is from 0 to \((\sum_{k=1}^{n} a_k) - 1\). These values are typically added to some base address to determine the position of the referenced item of data. In general, the quantities

\[
p_i = \prod_{j=1}^{i} a_j, \quad i = 0, 1, 2, \ldots, n - 1
\]

are computed at compile time. In addition, \(p_i\) units of memory are reserved for the overall data set, and the quantities \(p_0, p_2, \ldots, p_{n-1}\) are stored to be used as coefficients in determining specific positions when \(x_i\)'s are specified at run time.

The following discussion defines a class of data structures that is more general than the \(n\)-dimensional rectangular array, as well as a description method and a form for the reference expressions which are entirely suitable for inclusion in high-level symbolic programs. The discussion also derives the form of the storage mapping function for structures so described.

### Generalized Structures

This section describes and defines a class of data structures; the class includes \(n\)-dimensional rectangular arrays as special cases. This generalization is not an idle one, but is motivated by the fact that, although any such structure can, by suitable manipulation, be eventually incorporated into \(n\)-dimensional arrays, it is very often neither convenient nor efficient to do so. The result of such incorporation is often a "sparse" matrix which is wasteful of memory. That any such structure can be eventually described by an \(n\)-dimensional array follows from the fact that any such structure is mappable into a linear (one-dimensional) \((n = 1)\) array. Furthermore, in this day of procedure-oriented languages, the description technique which is most natural is the preferred one, and the contention here is that the structures and description technique to be defined are the natural ones for many data sets.

An English language description of the set of structures under consideration might help in understanding the formal definition to follow. The structures under consideration are, broadly, those that can be presented in an "outline" or "list" or "level" form, with the additional property that multiple instances of any "level" can occur. For example, a personnel file might be made up of 40 records, each record being composed of, say, three distinct items -- name, salary history, and work evaluation list. The item "salary history" may allow for the ten most recent raises and dates; the item "work evaluation list" may provide for the ten most recent ratings. In outline or list form, the items might be stated as

Personnel Record (40 instances)
- Name
- Salary History (10 instances)
- Date
- New Salary
- Evaluation List (10 instances)
- Rating

For each item in the structure that is not a heading for a list of subitems -- that is, for each datum -- the description must assign the number of data "particles" of which the item is composed. The particles may be bits, characters, bytes, etc., but it is assumed that, whatever the basic particle is, it is the same throughout. Thus, for each of the data items (name, date, new salary, rating) in the above example, a number of characters, say, would be assigned.

It should be understood that whenever multiple instances are indicated, the structures below each instance are all identical.

### Generalized Structure Descriptors

The set of descriptions of these structures are now defined in a formal manner, using the now well-known Backus definition form.\(^1\) In effect, a set of strings of symbols are defined, which, when given the proper interpretation, are, in each case, a description of the kind of data structure discussed above.

Remarks similar to those above, concerning the difference between a skeletal structure with free variable names, and particular instances of such structures where these variables are bound by context, must also be made here. The difference will little affect this discussion, and, for convenience, the identifiers will be assumed to be contextually meaningful. The metalinguistic variables ::=, <, >, and | will be employed, and the classes <IDENTIFIER> and <IDENTIFIER> are assumed to be already defined. Let the class of formal representations of these descriptions be called FIELD, where this is defined by

\[
\text{FIELD} ::= \text{IDENTIFIER} (\text{<INTEGER>}) | \text{IDENTIFIER} (\text{<INTEGER>}, \text{<FIELD SEQUENCE>})
\]

Using single Roman capitals as IDENTIFIERS, the following examples of FIELDs can be given:

A (4)
A (4, B(5))
A (4, B(5), C(6))
A (4, B(5), D(3) E(2) C(6)).

The personnel file above would have the form:

\[
P (40, U(20) S(10), D(6) T(5) E(10), R(2)).
\]

Those structures definable by FIELDs will be referred to as generalized structures, and an interpreted FIELD will be referred to as a gen-

\(^1\) The present writer is not aware of any formal treatment of "skeletal" structures, with free variable names, and particular instances of such structures where these variables are bound by context.
eralized structure descriptor, or, simply, a descriptor.

If \( \text{FIELDs} \) defined as follows:
\[
\begin{align*}
\text{FIELD} &::= \text{IDENTIFIER} (\text{INTEGER}) | \\
& \quad \text{IDENTIFIER} (\text{INTEGER}), \text{FIELD}
\end{align*}
\]
are considered, it is clear that these FIELDs represent a subset of the set of all FIELDs, and that this subset is equivalent to the set of \( n \)-dimensional arrays. The association proceeds as follows:
\[
\begin{align*}
\text{A} (a_1, a_2, \ldots, a_n) &\text{ corresponds to} \\
A_1 (a_1, A_2(a_2, \ldots, A_n(a_n)) \ldots).
\end{align*}
\]

There is, to be sure, a redundancy in the description when the generalized structure is, in fact, an \( n \)-dimensional array. This follows from the fact that there is descriptive information in the position of the elements of the \( n \)-tuple. It will be shown, when discussing the reference expressions, that an important feature of the generalized structure method is that there are no restrictions engendered by position.

Storage Mapping Functions and Reference Expressions

The class of structures under consideration, and the method of describing the structures in a symbolic program, have been defined by FIELDs; this section discusses the form of the reference expressions and the storage mapping function, as well as the processing of the FIELDs that is carried out at compile time. For the discussion, it is convenient to consider a class of pictorial representations known as L-trees -- trees that have legs, as well as branches.

It is well known how "lists" or "outlines" can be uniquely associated with trees. The structures discussed here, which can exist as multiple copies of identical substructures, require another type of branching symbolism -- thus, the legs.

Fig. 1. L-Tree Representation of the FIELD \( A(n, B_1() \ldots B_k()) \)

In general, the FIELD \( A(n, B_1() \ldots B_k()) \) has the representation indicated in Fig. 1. As shown, the number of legs emanating from the table below leg node \( A \) is \( n \). Each leg terminates in a branch node from which emanates the \( K \) branches of the FIELD SEQUENCE. These branches, along with the integer \( n \), define FIELD \( A \).

Further, if the FIELD is a special one of the form \( A(n) \), it is represented as indicated by Fig. 2. In this figure, the \( n \) endpoints have no further emanations and represent particles of data. In short, to each FIELD corresponds a leg node, and to each FIELD SEQUENCE correspond \( n \) branch nodes, below each of which the structure is identical. The branch nodes are not named, but they do have a natural order, and, in a picture, an integer indicating the number of branch nodes present will follow the last of a set of identically structured branch nodes, if they are not all drawn.

This pictorial representation clarifies the manner of referring to a specific particle of data. Since leg nodes are referenced by identifiers, and branch nodes by integers, reference to a specific particle of data is accomplished by specifying a sequence of identifier-integer couples, rather than a sequence of integers (with one identifier) as for rectangular arrays. The identifier portion of the couple refers to a leg node, and the integer part to one of the \( n \) branch nodes immediately following the referenced leg node -- or, in the special case, the integer part of the couple refers to a particular particle.

More formally, then, a reference expression is defined as a set of couples \( \{ (A_i, x_i) \} \) for \( i = 1, 2, \ldots, k \), such that \( A_i \) is an identifier and \( x_i \) is an integer. Thus, if there is, in the descriptor, \( A_1(a_1), B_1(\ldots)B_i(\ldots) \), then \( 1 \leq x_i \leq a_1 \), and \( A_{i+1} \) is one of the \( B \)'s, or, if the form \( A_j(a_j) \) appears in the descriptor, then \( k = j \).

Fig. 2. L-Tree Representation of the FIELD \( A(n) \)

Thus, the example given above, namely:
\[
P(40, U(20), S(10, D(6), T(5)), E(10, R(2))),
\]
has the L-tree indicated in Fig. 3. The arrows in the figure point to the particles referred to by the following reference expressions:
\[
\begin{align*}
\{(P, 2), (U, 2)\} \quad &\{(P, 2), (S, 3), (T, 4)\} \\
\{(P, 2), (E, 2), (R, 1)\} \\
\end{align*}
\]
It is now clear that the uniqueness of reference is not destroyed if the couples in a reference expression are rearranged from the natural ordering. This characteristic is at least partial repayment for the redundancy which is present when the structure is an n-dimensional rectangular array.

For rectangular arrays, there is a natural way to map the set onto a linear array. The effect of this mapping procedure is that \((v_1, v_2, \ldots, v_n)\) precedes \((w_1, w_2, \ldots, w_n)\) in the linear array if and only if

1. \(v_t < w_t\), or
2. There exists a \(t\) such that \(v_1 = w_1\) for \(i < t\) and \(v_t < w_t\).

Thus,

\[
\begin{align*}
(1, 1, \ldots, 1) & \text{ is in the 0th position} \\
(1, 1, \ldots, 2) & \text{ is in the 1st position} \\
& \quad \vdots \\
(1, 1, \ldots, 1, a_n) & \text{ is in the } (a_n - 1)\text{st position} \\
(1, 1, \ldots, 2, 1) & \text{ is in the } a_n\text{th position} \\
& \quad \vdots
\end{align*}
\]

It is apparent that this ordering is precisely that provided by the storage mapping function previously given for n-dimensional arrays.

With the aid of L-trees, the natural mapping of the data particles onto a linear array may be discussed, as well as the storage mapping function that is implied, for the case of generalized structures.

At each node of an L-tree, a natural ordering on the legs or branches -- a left-to-right ordering -- can be defined. This, in turn, imparts an ordering on all of the data particles. To achieve this ordering, the tree is traversed according to the regime of moving always down the leftmost leg or branch not already traversed; when such downward movement cannot continue, the traversing moves to the next leg to the right, or, if there are none, back to the branch node at the next level up, then to the next branch to the right, if possible, or else up to the next level, and so forth. In this way, each particle is encountered once and only once; the order in which the particles are encountered is the order which will be generated by the storage mapping function to be defined for generalized structures.

This ordering can be more explicitly defined as follows. Let \(\{(A_i, X_i)\}\) be the reference expression for data particle \(P_i\), and \(\{(C_j, Y_j)\}\) be the reference expression for \(P_j\), where \(i = 1, 2, \ldots, k_1\), \(j = 1, 2, \ldots, k_2\), and \(A_1 = C_1\). Then \(P_2\) will precede \(P_1\) in the ordering if

1. \(X_1 < X_2\), or
2. There exists a \(t > 1\) such that
   \[C_i = A_i \text{ and } Y_i = X_i, \text{ for } i < t, \text{ and}
   \]
   \[C_t \text{ precedes } A_t \text{ in the FIELD SEQUENCE that defines } A_{t-1}, \text{ or}
   \]
3. There exists a \(t > 1\) such that
   \[C_i = A_i \text{ and } Y_i = X_i, \text{ for } i < t, \text{ and}
   \]
   \[C_t = A_t \text{ and } Y_t < X_t.
   \]

The position in the linear array, then, of a particle, \(P_1\), referenced by \(\{(A_i, X_i)\}\), \(i = 1, \ldots, k\), is equal to the total number of particles that precede \(P_1\) by virtue of 1, 2, or 3, above. It is clear that these conditions are mutually exclusive; that is, particle \(P_1\) can precede \(P_2\) by virtue of only one of these conditions, and if the conditions 2 or 3 are pertinent, then a specific \(t\) is involved. Thus, \(s_t\) is defined to be the total number of particles that satisfy condition 1, \(a_t\) to be the total number of particles that satisfy condition 2 or 3, where \(2 \leq t \leq k\). It follows from the above that the sets comprising these totals are disjoint sets and the number of particles in their union is the total number of particles preceding \(P_1\). Thus, the position of \(P_1\) is given by

\[
\sum_{t=1}^{k} (a_t + s_t),
\]

where \(a_1 = 0\). What this means in terms of the L-tree is demonstrated next.
For each leg node (or FIELD), \( F \), let \( M(F) \) be the total number of particles that lie below each of the \( n \) legs of \( F \). If the legs of \( F \) are particles, \( M(F) = 1 \). Let \( N(F) = n \cdot M(F) \). Note that, if \( B_1, B_2, \ldots, B_i \) are the FIELDs in the FIELD SEQUENCE for \( F \) (that is, successive branches of a branch node), then \( M(F) = \Sigma [N(B_i)] \). Let us define \( Q(B_i) = 0 \), and \( Q(B_i) = \Sigma \left[ \prod_{j=1}^{i-1} N(B_j) \right] \) for \( i = 2, \ldots, j \). Note that \( Q(B_i) \) is the total number of particles below \( B_1, B_2, \ldots, B_{i-1} \).

It should be clear that, for \( P_i \) given by \{ \( A_i \), \( x_i \) \}, \( i = 1, 2, \ldots, k \),

\[ S_i = M(A_i)(x_i - 1), \]

where \( Q(A_i) \) is set equal to 0.

The \( Q \)'s and \( M \)'s can be computed at compile time quite readily, given the generalized structure descriptor. The amount of space reserved for such a structure is \( N(A) = a_i M(A_i) \) particle positions. At run time, when the \( x_i \)'s and \( A_i \)'s are supplied, the relative address is easily computed. Note that the \( A_i \)'s must be supplied in this general case, in addition to the \( x_i \)'s -- for \( n \)-dimensional arrays only the \( x_i \)'s need be given -- since not all particles are derived from the same set of \( A \)'s, although all of the particles have \( A_i \) in common.

It is interesting to note the relationship between these quantities to the quantities in the storage mapping function of an \( n \)-dimensional rectangular array. Let the generalized structure descriptor be \( A \), \( A_1, A_2, \ldots, A_k \). Since each FIELD, \( Q(A) = 0 \) for each \( i \). Now, \( M(A_k) = 1 \), \( N(A_k) = a_k \), and \( M(A_k - 1) = N(A_k) = a_k \). In general, \( M(A_{j-1}) = N(A_j) = a_{j-1}M(A_j) \). Thus, by induction,

\[ \sum_{i=1}^{k} M(A_i) = a_i \]

This is precisely the expression for the \( p \)'s derived above for rectangular arrays.

This completes the assertion that the description method, reference expressions, and storage mapping functions for generalized structures are direct generalizations of those for \( n \)-dimensional rectangular arrays.

Storage Mapping Function Computation

Appendix A is a Burroughs ALGOL program for the Burroughs 220 computer which simulates, first, the computations carried out by a computer upon receiving the generalized structure descriptor at compile time, then the computations that would be carried out at run time when presented with reference expressions.

The first task of the program is to place the input FIELD into a vector, one word for each symbol. A table is then constructed which has one row for each identifier or FIELD of the input. This table is, in some respects, similar to a Perlis thread list; the same names for the columns have, in fact, been adopted.

The \( f \) column of the table is defined, for the FIELD (or IDENTIFIER or ROW) named \( A \), as follows. If \( A \) is a FIELD of the form \( A(n) \), then \( f(A) = 2 \), depending upon whether or not \( A \) is the last FIELD in a FIELD SEQUENCE. If \( A \) is a FIELD of the form \( A(n, B_1(\ldots,B_k(\ldots)) \), then \( f(A) = 3 \), depending, again, upon whether or not \( A \) is the last FIELD in a FIELD SEQUENCE.

The \( t \) column is defined as follows. \( A \) is of the form \( A(n, B_1(\ldots,B_k(\ldots)) \), then \( t(A) = B_1 \); if \( A \) is of the form \( A(n) \), \( t(A) \) is undefined.

For the \( r \) column, if \( A \) is the overall FIELD, \( r \) is undefined. If \( f(A) = 0 \) or 1, then \( r(A) \) is the succeeding FIELD in the FIELD SEQUENCE in which \( A \) occurs. Otherwise (that is, \( f(A) = 2 \) or 3), \( r \) is the FIELD which is defined by the FIELD SEQUENCE of which \( A \) is the last FIELD in the sequence.

Finally, a \( c \) column is defined; this column has no counterpart in threaded lists. For each FIELD, \( A \), the \( c \) column contains the associated INTEGER in that descriptor; that is, whether \( A \) is of the form \( A(n) \) or \( A(n, B_1(\ldots,B_k(\ldots)) \), \( c \) for that \( A \) is \( n \). No information appears in this table that is not in the original descriptor, although the table is now in a more usable form for computing the \( N \)'s, \( M \)'s, and \( Q \)'s.

If \( f(A) = 0 \) or 2, \( M(A) \) is set equal to 1, and \( N(A) \) is set equal to \( c(A) \). From this point on, the task is reduced to finding a FIELD (defined by a FIELD SEQUENCE) all of whose FIELD SEQUENCE components have their \( N \)'s already defined. Thus, if \( A \) is defined by \( A(n, B_1(\ldots,B_k(\ldots)) \), then

\[ M(A) = \sum_{i=1}^{k} N(B_i), \]

and \( N(A) = c(A) \cdot M(A) \), where, of course, \( c(A) = n \), in this case. At this time \( Q(B_i) \) can also be computed for these \( i \)'s by

\[ Q(B_i) = \sum_{j=1}^{i-1} N(B_j) = Q(B_{i-1}) + N(B_{i-1}). \]

This process is iterated until \( N(A) \) is found for the \( A \) which is the overall FIELD being processed.
This processing is carried out by constructing an incidence matrix, \( G \), by using the \( f, t, \) and \( r \) columns, such that the matrix has as many rows and columns as there are FIELDs, and has a 1 in row \( A \), column \( B \), if \( B \) is in the FIELD SEQUENCE defining \( A \). The \( A \)'s for which \( M \)'s are ready to be computed are then determined by comparing the rows of \( G \) to a vector which has 1's for those FIELDs for which \( N \) is already determined.

The \( G \) matrix is constructed by finding, for each FIELD \( A \), for which \( f(A) \) is 1 or 3, the constituents of its FIELD SEQUENCE, by looking at \( t(\), \( r(\), \( r\cdots(\),$ etc., until \( r(... r(t(A))... ) = A \). All of the FIELDs so encountered are constituents. The order in which the FIELDs are encountered is later used (when the \( N \)'s for the FIELDs are known) in computing the \( Q \)'s of these constituents of the FIELD SEQUENCE, \( A \).

These computations are now summarized for the example above. The input, as before, is:

\[ P(40, U(20) S(10, D(6) T(5)) E(10, R(2))). \]

The following table is constructed (where * means undefined):

<table>
<thead>
<tr>
<th></th>
<th>( f )</th>
<th>( t )</th>
<th>( r )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>U</td>
<td>*</td>
<td>40</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>*</td>
<td>S</td>
<td>20</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>*</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
<td>*</td>
<td>S</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>R</td>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td>*</td>
<td>E</td>
<td>2</td>
</tr>
</tbody>
</table>

The following are then computed in order:

- \( M(U) = 1 \)
- \( N(U) = 20 \)
- \( M(D) = 1 \)
- \( N(D) = 6 \)
- \( M(T) = 1 \)
- \( N(T) = 5 \)
- \( M(R) = 1 \)
- \( N(R) = 2 \)
- \( Q(R) = 0 \)
- \( M(S) = N(D) + N(T) = 11 \)
- \( N(S) = 10\cdot 11 = 110 \)
- \( Q(D) = 0 \)
- \( Q(T) = 6 \)
- \( M(E) = N(R) = 2 \)
- \( N(E) = 10\cdot 2 = 20 \)
- \( M(P) = N(U) + N(S) + N(E) = 150 \)
- \( N(P) = 40\cdot 150 = 6000 \)

Thus, in tabular form:

<table>
<thead>
<tr>
<th></th>
<th>( M )</th>
<th>( N )</th>
<th>( Q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>150</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>11</td>
<td>110</td>
<td>20</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>20</td>
<td>130</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The 6000 in this table indicates the total storage required for this FIELD; beyond this point, only the \( M \) and \( Q \) columns need be retained for processing the reference expressions at run time.

Next to be considered is the manner in which this data would be used to determine the position indicated by the reference expression \( f(P, 2), (S, 3), (T, 4) \). Using the storage mapping function:

\[ f(X) = 0 : \]

\[ f(X) = 1 : \]

\[ f(X) = 2 : \]

\[ f(X) = 3 : \]

Fig. 4. Definition of "Boxes" in Terms of the Quantities \( f(x), t(x), r(x), \) and \( c(x) \)
\[
\sum [Q(A_i) + (x_i - 1)M(A_i)] = [0 + (2 - 1) \cdot 150] + [20 + (3 - 1) \cdot 11] + [6 + (4 - 1) \cdot 1] = 150 + 42 + 9 = 201
\]

There is another useful pictorial method for representing the structures under consideration. Each FIELD has a "box." Two sections are associated with each box. The first section contains the IDENTIFIER, \(X\), of the FIELD; the second section contains \(c(X)\). There are arrows emanating from and entering the boxes, the nature of which depends upon \(f(X)\). This dependence is given in Fig. 4. Using these definitions, the descriptor

\[
P(40, U(20) S(10, D(6) T(5)) E(10, R(2)))
\]

can be indicated as in Fig. 5.

Further Considerations

There are several questions concerning generalized structures which are still under consideration, but for which results cannot be presented at this time.

The first of these questions involves a more general definition of FIELD which allows a (previously defined) FIELD to be referred to by FIELD name, alone, without indicating the associated FIELD SEQUENCE, or the number of data particles. In addition, the generalization from INTEGER to ARITHMETIC EXPRESSION is highly desirable, but will, of course, delay the M and Q computations until runtime. The same delay is incurred in computing the coefficients for the storage mapping function for n-dimensional rectangular arrays, when arithmetic expressions are used in the descriptor.

A second area of present and future interest is that of nonhomogeneous memory and/or data. For example, the particles of data may not be the same throughout. If single bits are used in some places, and six-bit characters in others, for example, then clearly the descriptor, reference expressions, and mapping function must reflect this. The problems here seem reasonable to handle, but, of course, the descriptor definition must be modified.

In keeping with the notion, alluded to before, that structure definition is distinct from instances of that structure, the IDENTIFIERS could be removed from a descriptor, and the remains considered as a structure definition. In terms of the L-tree, the effect would be merely to remove the IDENTIFIERS -- the tree does not change. In declaring data which has such a structure, it would be necessary only to supply a sequence of IDENTIFIERS which could be interleaved with the skeletal representation, placing one before each left parenthesis. Using the example once again, the structure could be defined as

\[
(40, (20) (10, (6) (5)) (10, (2)))
\]

The instance noted above could now be declared by naming this skeletal structure and providing the sequence of IDENTIFIERS \(P, U, S, D, T, E, R\). The \(M\)’s and \(Q\)’s can be determined from the skeletal structure alone, and are usable for any subsequent instance. All of this, of course, a notational simplification for writing the structure descriptor in terms of free-variable IDENTIFIERS, and then binding these IDENTIFIERS by a sequence of IDENTIFIERS.

It is interesting to consider the set of such skeletal structures as being in reality a generalization of all Euclidean spaces; that is,

\[
\langle\text{GENERALIZED POINT}\rangle ::= \langle\text{GENERALIZED-SEQUENCE}\rangle
\]

\[
\langle\text{GENERALIZED SEQUENCE}\rangle ::= \langle\text{GENERALIZED POINT}\rangle | \langle\text{GENERALIZED SEQUENCE}\rangle
\]

This defines a set of formal objects, \(\langle\text{GENERALIZED POINT}\rangle\), which is a direct generalization of the collection of all \(n\)-tuples of integers for all \(n\).

Acknowledgment

The author wishes to thank Mr. K. Speierman, who suggested the value of a study in this area, and who made many useful contributions to the completion of the work.

References

Appendix A - Burroughs ALGOL Program

The following is taken directly from the printer output of the Burroughs 220 computer, as employed for the Burroughs ALGOL program as described above.

```
BEGIN
COMMENT THIS PROGRAM TAKES A GENERALIZED DATA STRUCTURE FIELD AS INPUT AND GENERATES A TABLE UPON WHICH THE GENERALIZED LINEAR STORAGE MAPPING FUNCTION OPERATES. THE INPUT
REQUIRES EACH FIELD IN THE FIELD TO BE ON A SEPARATE CARD CODED IN THE 220 ALPHANUMERIC CODE. EACH IDENTIFIER AND
INVERSE MUST BE ONE CHARACTER LONG. LET THE NUMBER OF CARDS BE N.
COMMENT
GIVEN THE INPUT FIELD, THE OUTPUT WILL BE IN THE FORM OF A TABLE WITH ONE ROW FOR EACH IDENTIFIER, THESE ARE LISTED UNDER THE HEADING BILL. THE OTHER COLUMNS ARE J, L(J), M(J), AND N(J); THE POSITIONS, SIGMA(J), AND SIGMA(I), WHEN THIS IS COMPLETED, THE PROGRAM IS READY TO ACCEPT
REFERENCE EXPRESSIONS

INTEGER OTHERWISE ARRAY

INPUT FIELD FOR I=1 TO INV(J)
BEGIN
COMMENT THIS FINDS THE POSITIONS SIGMA(J), IN THE INPUT STRING OF
FOR J=1 TO INV(J)
BEGIN
INPUT J=IN(J)
BEGIN
COMMENT THIS Fills THE ROWS
FOR J=1 TO INV(J)
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
END
EN
BEGIN
IF I EQL JMA
GO TO PUTOUT
I = I+1$
GO TO EPS 11..0N$
END$

Q(l..N(I+1
IV=L..N$
ZETA.. IF RNC IV I EQ I..1$
BEGIN
IF I EQ 1.. JM$
GO TO PUTOUT$
I = I+1$
GO TO EPS 11..0N$
END$
Q(RNC IV) I = $I (IV) + Q(IV)$
IV = RN(IV)$
GO TO ZETA$

PUTOUT.. WRITE($$ HEAD)$
IMR!TE
1$$
TA8.L~~FORM)$
COMMENT AT THIS POINT THE PROGRAM WILL ACCEPT REFERENCE
EXPRESSIONS IN THE FORM IDENTIFIER. INTEGER. IDENTIFIER.
INTEGER ETC. EACH IS ON A SEPARATE CARD AND ALPHANUMERICALLY CODED. THEY MUST BE PRECEDED BY A CARD
CONTAINING TZ, THE NUMBER OF CARDS IN THE
REFERENCE EXPRESSIONS

INPUT TEST(TZ,FOR I=I..TZ)$T(I)$
OUTPUT POSITION(SMF)$
TESTSTART.. READ ($$TEST)$
FOR I=(I..I..TZ/2)$
BEGIN
T(21 )=CT(21 )-80000000001/100000000$
FOR J=(I..JM)$
BEGIN
IF B(JI EQ I.. T (21 -1)$
TNC I ) =J$
END$
END$
COMMENT THIS IS THE STORAGE MAPPING FUNCTION COMPUTATION
FOR 1=(I.I..TZ/2)$
SMF=SMF+M (TN ( 1 ) I. (T (21 )-1 ) +0 (TN ( 1 ) )$
WRITE ($$POSITION.POS) $
GO TO TESTSTART$
THFEND.. FINISH$

From the collection of the Computer History Museum (www.computerhistory.org)