               Specifications for Datalanguage, Version 0/9

Preface

   Datalanguage is the language processed by the Datacomputer, a data
   utility system being developed for the Arpanet.  The Datacomputer
   performs data storage and data management functions for the benefit
   of computers on the network.

   Version 0/9 is currently running at CCA.  This version is extremely
   primitive; however, it does offer an opportunity for experience with
   the Datacomputer and with fundamental Datalanguage concepts.

   Subsequent versions will provide greater portions of the full
   Datalanguage capability, which has been described earlier
   (Datalanguage, Working Paper No. 3, Datacomputer Project, October,
   1971, NIC 8028).  For example, one of the primary restrictions in
   0/9--elementary data items must be fixed-length ASCII strings--will
   be eliminated in Version 0/10, which is currently being implemented.

   Based on the experience gained in the implementation of these early
   versions, and based on the feedback from their use, a revised
   specification of the full language will be issued.

1. Introduction

   This document presents a precise and complete specification of
   Datalanguage, Version 0/9.  It is organized into 11 sections, of
   which this introduction is the first.  Section 2 discusses the
   capabilities of Version 0/9 in general terms.  Sections 3 and 4 are
   concerned with data description and the directory.  Sections 5
   through 8 cover the expression of data management operations.
   Section 9 discusses the recognition of names.  Section 10 covers
   miscellaneous topics and Section 11 specifies the syntax in BNF.

   This specification is to be followed with a user manual, which will
   present the language in tutorial form and treat components of the
   Datacomputer-user interface other than the language.

2. Capabilities of Version 0/9

   Version 0/9 of Datalanguage has capabilities for the storage of
   files; for addition of data to existing files, and for the deletion
   of files.  Retrievals can output whole files as well as subsets of
   files.  Data can be selected from files by content, using expressions
   formed from boolean and inequality operators.

   At the option of the file creator, an inversion is constructed and
   maintained by the Datacomputer.  The inversion increases the
   efficiency of selective retrieval, at the cost of storage space and
   file maintenance effort.  Users other than the file creator need not
   be aware of the existence of the inversion, or of which fields are
   inverted file keys.  The language is designed so that they state the
   desired result of a retrieval, and the Datacomputer uses the
   inversion as much as the request permits.

   Elementary data items are fixed-length ASCII strings.  Files are a
   restricted class of hierarchical structures.

   Many of the restrictions mentioned in this memo will be short-lived.
   In particular, those statements followed with 3 asterisks (***) refer
   to restrictions that will be considerably weakened or eliminated
   entirely in the next version of the software.

3. Data Description

   A container is a variable whose value is a data object of general
   character and arbitrary size (In Version 0/9, size is restricted.
   See section 3.4).  Examples of containers which are implemented in
   other systems are files, records, fields, groups, and entries.

   The container is distinct from the data in the container.  For
   example, space allocation is an operation on a container, while
   changing the unit price field from 25 to 50 is an operation on data
   in a container.

   A container may enclose other containers.  When a container is not
   enclosed by another container, it is said to be outermost.  If
   container A encloses container B, and no other container in A also
   encloses B, then A immediately encloses B.

   A Datalanguage description is a statement of the properties of a
   container.

   All containers have the attributes ident and type.  Ident is a
   character string by which users refer to the container.  Type
   determines the form of the container's value; the value can be

elementary, or it can consist of other containers.  There are 3
types: LIST, STRUCT, and STRING(***).  A LIST contains a group of
containers having the same description.  A STRUCT contains a group of
containers, each of which has its own description.  A STRING is a
sequence of ASCII characters.  While a STRING is not really an
elementary item, it is handled as one in Version 0/9.

Certain containers can have other attributes.  An outermost container
has a function.  The function attribute specifies whether the
container is to be used for storage or for transmission.

Size is some meaningful dimension of the container, which is type-
dependent.  It is used for space allocation and data stream parsing.

An aggregate container (i.e., one that contains other containers) has
as an attribute the description or descriptions of its components.
Thus if S is a STRUCT containing A, B, and C, then the descriptions
of A, B, and C are attributes of S.

A STRING defined in certain contexts can have an inversion attribute.
This is an access property that is not really local to the STRING,
but is associated with it for convenience.

## 3.1 Ident

The ident of a container is composed of alphanumeric characters,
the first of which is alphabetic.  It may not consist of more than
100 characters.

The elements of a STRUCT must have idents unique in the STRUCT.

## 3.2 Function

The function of a container is either FILE, PORT, or TEMPORARY
PORT.  When the function is FILE, then the container is used for
storage of data at the Datacomputer.  When the function is PORT,
then the container is used for transmission of data into or out of
the Datacomputer.  When the function is TEMPORARY PORT (which may
be abbreviated TEMP PORT), the container behaves like a PORT;
however, its description is not retained in the Datacomputer
beyond the session in which it is created.

3.3 Type

    Type is one of: LIST, STRUCT, or STRING.  These are defined on the
    preceding page.

    In an occurrence of a STRUCT, the elements appear in the order in
    which their descriptions appear in the STRUCT description.  All
    elements are present in each occurrence of the STRUCT.

    An element of a STRUCT or LIST can be a container of any datatype.
    However, the outermost container must be a LIST(***).

3.4 Size

    The size of a STRING is the number of characters in it.  The size
    of a STRUCT is not defined (***).  The meaning of the size of a
    LIST depends upon other properties of the LIST (***).

    Ordinarily, the size of a LIST is the number of LIST-members.  An
    exception is the case of the outermost-LIST.  In an outermost-LIST
    with a function of FILE, the size is the number of LIST-members
    for which space should be allocated.  When no size is present in
    this case, the system computes a default.  In an outermost-LIST
    with a function of PORT, the size is ignored (***).

    Only outermost containers may be larger than a TENEX page (2560
    ASCII characters)(***).

3.5 Inversion

    An inversion is an auxiliary data structure used to facilitate
    retrieval by content.

    Its basic application is the fast retrieval of sets of outermost-
    LIST-members (this can be extended to other container sets, and
    will be after release 1).  Consider a list of weather
    observations, stored as a file on the Datacomputer.  If quick
    retrieval of observations by COUNTRY is desired, then this is
    indicated in the description of the COUNTRY container.  According
    to common usage in information retrieval, this makes COUNTRY a key
    in the retrieval of observations.

    Note that the inversion option only affects the efficiency of
    retrieval by COUNTRY, not the ability to retrieve by COUNTRY.

There are restrictions on use of the inversion option.  First, it
can be applied only to STRINGs.  Second a STRING having the
inversion option must occur only once in each
outermost-LIST-member.  Third, it is ignored when applied to
STRINGs in PORT descriptions.

Eventually there will be several types of inversion option; in
Version 0/9 there is only the 'D' option (for distinct).

3.6 Syntax

The description is simply an enumeration of properties; these
properties are specified in the order:

        <ident> <function> <type> <size> <other>

Properties which do not apply are omitted.  An example:

        F FILE LIST (25) A STR (10)

Here 'F' is the <ident>, 'FILE' is the <function>, 'LIST' is the
<type>, '(25)' is the size, and 'A STR (10)' is the <other> of one
description.  Of course, 'A STR (10)' is itself another
description: the description for members of the LIST named F.

An example of a complete description for a file of weather
observations keyed on location:

        WEATHER FILE LIST
        OBSERVATION STRUCT
            LOCATION STRUCT
                    CITY STR (10), I=D
                    COUNTRY STR (10), I=D
                    END
            TIME STRUCT
                    YEAR STR (2)
                    DAY STR (3)
                    HOUR STR (2)
                    END
            DATE STRUCT
                    TEMPERATURE STR (3)
                    RAINFALL STR (3)
                    HUMIDITY STR (2)
                    END
            END
The ENDs are needed to delimit the list of elements of a STRUCT.
', I=D' indicates that the string is to be an inversion key for
the retrieval of outermost-LIST-members.

4. Directory

   The directory is a system file in which the names and descriptions of
   all user-defined containers are kept.

   The directory is structured as a tree.  Each node has an ident, which
   need not be unique.  There is a single path from the root of the tree
   to any node.  The idents of the nodes along this path are
   concatenated, separated by periods, to form a pathname, which
   unambiguously identifies the node (e.g., A.B.C could be a pathname
   for node with an ident of C).

   In a later version of the software, the directory will be generalized
   to provide for links between nodes, so that it will not properly be a
   tree.  For now, however, the tree model is convenient and adequate.

   A node may represent a container, or it may simply hold a place in
   the space of pathnames.  When it represents a container, it cannot
   (currently) have subordinate nodes.

   Eventually, it is planned to model the directory as a structure of
   containers, with its description distributed throughout the
   structure.  Most operations defined on the directory will be defined
   on user data, and vice versa.  Access privileges and privacy locks
   will be part of the data description and will likewise be applicable
   both to directory nodes and data structures below the node level.

4.1 CREATE

     A CREATE-request either; (a) adds a node to the directory,
     optionally associating the description of either a PORT or a FILE
     with the node, or (b) creates a temporary container which is not
     entered in the directory, but has a description and can be
     referenced in requests.  If the description defines a file, CREATE
     causes space to be allocated for the file.

     To create a node with a description:
         CREATE <pathname> <description> ;
     To create a node with no description:
         CREATE <pathname> ;
     Note that the description determines whether or not the container
     is temporary (see section 3.2 for details).

A CREATE-request adds a single node to the directory.  Thus to add
CCA.RAW.F to an empty directory, three requests are needed:
```
CREATE CCA ;
CREATE CCA.RAW ;
CREATE CCA.RAW.F ;
```
Notice that the last ident of the pathname doubles as the first
ident of the description:
```
CREATE CCA.RAW.G FILE LIST A STR (5) ;
```

That is, G is both the ident of a node and the ident of an
outermost container of type LIST.

## 4.2 DELETE

A DELETE-request deletes a tree of nodes and any associated
descriptions or data.  The syntax is:
```
DELETE <pathname> ;
```
The named node and any subordinates are deleted.

Note that to delete data while retaining the directory entry and
description, DELETE should not be used (see section 6.3 for the
proper method).

## 4.3 LIST

The LIST-request is used to display system data of interest to a
user.  It causes the data specified to be transmitted through the
Datalanguage output port.

Several arguments of LIST apply to the directory.  LIST %ALL
transmits all pathnames in the directory.  LIST %ALL.%SOURCE
transmits all descriptions in the directory.  Instead of %ALL, a
pathname can be used:
```
LIST <pn>.%ALL
```
Lists pathnames subordinate to <pn>.
```
LIST <pn>.%SOURCE
```
lists descriptions subordinate to the node represented by <pn>.

For details about the LIST-request, see section 10.1.

## 5. Opening and closing containers

Containers must be open before they can be operated on.

A container is open when it is first created.  It remains open until
closed explicitly by a CLOSE-request or implicitly by a DELETE-
request or by session end.

A closed container is opened by an OPEN-request.  A temporary
container is always open; a CLOSE-request deletes it.

5.1 Modes

An open container has a mode, which is one of: READ, WRITE, or
APPEND.  The mode determines the meaning and/or legitimacy of
certain operations on the container.

The mode is established by the operation which opens the
container.  It can be changed at any time by a MODE-request.  A
CREATE leaves the container in WRITE mode.  An OPEN either
specifies the mode explicitly or implicitly sets the mode to READ.

5.2 Syntax

To open a container:
    OPEN <pathname> <mode> ;
or:
    OPEN <pathname> ;
where <mode> is defaulted to READ.

To close a container:
    CLOSE <ident> ;
where <ident> is the name of an outermost container.

Two containers with the same outermost <ident> can not be
opened at the same time (***).

To change the mode of an open container:
    MODE <ident> <newmode> ;

5.3 LIST

LIST %OPEN transmits name, mode and connection status of each open
outermost container through the Datalanguage output port. (The
Datalanguage output port is the destination to which all
Datacomputer diagnostics and replies are sent.  It is established
when the user initially connects to the Datacomputer.)  For
details of the LIST-request, see section 10.1.

6. Assignment

Assignment transfers data from one container to another.

The equal sign ('=') is the symbol for assignment.  The value of the
operand on the right of the equal sign is transferred to the operand
on the left.  (Eventually, both operands will be weakly-restricted

Datalanguage expressions, which may evaluate to sets as well as to
single containers.  Now, the left must be a container name, the right
may be a container name or a constant.)

Assignment is defined for all types of containers.  When the
containers are aggregates, their elements are paired and data is
transferred between paired elements.  Elements of the target
container that do not pair with some source element are handled with
a default operation (currently they are filled with blanks).

The operands of an assignment must have descriptions that match.  The
idea of matching is that the descriptions must be similar enough so
that it is obvious how to map one into the other.

6.1 Conditions for legitimate assignment

    Assignment must reference objects, not sets.  An object is:
       (a)    an outermost container, or
       (b)    a constant, or
       (c)    in the body of a FOR-loop, either
              (c1) a member of a set defined by a FOR-OPERAND, or
              (c2) a container which occurs once in such a member

    In the case of a reference of type (c1), the object referenced is
    taken to be the current member.  In the case of (c2), the object
    referenced is that which occurs in the current member.  This is
    explained further in section 7.

    The left operand of an assignment is subject to further
    restriction.  If it is an outermost container, it must be open in
    either WRITE- or APPEND-mode.  If it is not an outermost
    container, then the reference is of type (c), which means that
    some FOR-operand has established a context in which the assign-
    operand is an object.  The FOR-operand which establishes this
    context must be the output-operand of the FOR.

    When the assign-operand is an outermost container, it must be
    open.  Such an operand must be referenced by its simple container
    ident(***), not its directory pathname.

    In the body of a loop nested in one or more other loops,
    assignments are further restricted, due to a 0/9 implementation
    problem.  See section 7.2 for details.

Finally, the descriptions of the operands must match.  If one is a
constant, then the other must be a STRING(***).  If both are
containers, then in the expression:
   A = B;
the descriptions of containers A and B match if:
1. A and B have the same type
2. If A and B are LISTs, then they have equal numbers of
   LIST-members, or else A is an outermost-LIST.
3. If A and B are aggregates, then at least one container
immediately enclosed in A matches, and has the same ident as, one
container immediately enclosed in B.

6.2 Result of assignment

If the operands are STRINGS, then the value of B, left-justified,
replaces the value of A.  If B is longer than A, the value is
truncated.  If B is shorter than A, then A is filled on the right
with blanks as necessary.

If the operands are STRUCTs, then assignment is defined in terms
of the STRUCT members.  If a member of A, mA, matches and has the
same name as a member of B, mB, then mB is assigned to mA.  If no
such mB exists, then mA is filled with blanks.

If the operands are LISTs, the result depends on several factors.
First, notice that the descriptions of the LIST-members must
match;  otherwise the assignment would not be legitimate by the
matching rules of 6.1.

If A is an outermost-LIST, then it can be in either of two modes:
WRITE or APPEND.  If A is in WRITE-mode, its previous contents are
first discarded; it is then handled as though it were in APPEND-
mode.

If A is not an outermost-LIST, then it is always effectively in
WRITE-mode.

After taking the mode of A into account, as described above, the
procedure is:
    for each member of LIST B
    (a) add a new member to the end of A
    (b) assign the current number of B to the new member of A

6.3 Deletion of Data Through Assignment

     If A is an outermost container in WRITE-mode, and B is a container
     with description that matches A, and if B contains no data, then
     A=B has the effect of deleting all data from A.  Note that if A is
     in APPEND-mode in these circumstances, then A=B is a no-operation
     (i.e., has no effect).

7. FOR

     FOR <output set spec>, <input set spec> <body> END ;

  The output set is optional: that is, FOR need not produce output.
  When the output set is omitted, the syntax is:

     FOR <input set spec> <body> END ;

  The operations specified in the body are performed once for each
  member of the input set.  References in the body to the input set
  member are treated as references to the current input set member.
  When an output set is present, a new member is created and added to
  the output set for each iteration (i.e., for each input set member).
  References to the output set member, similarly, are treated as
  references to the current output set member.

  The output set spec must be the name of a LIST member.  Each
  iteration of the FOR will create one such member, and add it to its
  LIST (hereafter called the output LIST).  The body terminates the
  value that the new member receives.  Any STRING in the new member
  which is not given a value by the body receives he default value of
  all blanks.

  The input set spec must be an expression evaluating to a set of
  LIST-members (see section 7.1 for details of input set
  specification).  Each iteration for the FOR will input one such
  member; the FOR will terminate when each member of the set has been
  processed.  The LIST from which the input set members are drawn is
  called the input LIST.

  FOR is effectively a means of accomplishing variants of assignment
  between a pair of LISTS.  FOR is less concise than assignment, but
  offers more flexibility.  Its advantages are:
     (a) not all the input LIST-members need be transferred to the
         output LIST.  A subset can be selected by content.
     (b) the user has explicit control over the assignment of values to
         output LIST-members.

This is most easily understood by an example:

```
P PORT LIST                F FILE LIST
   R STRUCT                    R STRUCT
      B STR                       A STRUCT
      C STR                          A1 STR
      END                            A2 STR
                                  B STR
                                  C STR
                                  END
```

```
(1)    P = F ;
(2)    FOR P.R, F.R
            P.R = F.R ;
            END ;
(3)    FOR P.R, F.R WITH A1 EQ 'XY' OR A2 GE 'AB'
            B = C ;
            C = A2 ;
            END
```

Here, (1) and (2) are entirely equivalent requests.  However, (3) is quite different and is not expressible as assignment.  It selects a subset of F.Rs.  The values it gives to the P.Rs could not result from application of the matching rules to F and P.

Because FOR is effectively assignment between a pair of LISTs, the LISTs referenced by a legitimate FOR-operation are largely subject to the same restrictions as LISTs referenced in an assignment.  One exception is that the descriptions of the LIST-members need not match.

These restrictions are:
(a)    both LISTs must be objects in the context in which the FOR appears.
(b)    both LISTs must be open or contained in open outermost containers.
(c)    if the output LIST is an outermost container, it must be in WRITE- or APPEND-mode.
(d)    If the output LIST is not outermost, the LIST which most immediately encloses it must be the output LIST of an enclosing FOR.

The mode of the output LIST of the FOR affects the result much as it would in an assignment: that is, a FOR outputting to a LIST in WRITE-mode overwrites previous contents, while a FOR outputting to a LIST in APPEND-mode adds to previous contents.

   CAUTION TO THE READER: For convenience, these specifications use
   phrases such as 'LISTs referenced by a FOR'.  Recall that such a
   phrase is not literally correct, in the sense that the operands of a
   FOR are always LIST members, not LISTs.

7.1 Details of input set specification

   The input set is specified by a Datalanguage expression that
   evaluates to a set of LIST-members.  Such an expression can be
   simply the set of all members of a LIST, or it can be a subset of
   the members of a LIST, specified by content.  For example, with
   the description:

      F FILE LIST
           R STRUCT
                  A STR (1)
                  B STR (2)
                  END

   the expression:
      F.R
   references all R's on the LIST F.  However:
      F.R WITH A EQ '5'
   references only those R's containing an A having the value '5'.
   The expressions permitted as input set specifications are of the
   form:
      <list-member-name> WITH <boolexp>
   The <boolexp> is constructed of comparison expressions joined by
   the Boolean operators AND and OR.  Any expression can be negated
   with NOT.

   Comparison operators have the highest precedence.  Next highest is
   AND, then OR, then NOT.

   The comparison expressions are restricted to the form:
      <container name> <comop> <constant>
   where:
   (a)    <constant> is a string constant enclosed in single quotes
          (see section 10.2 for a discussion of constants)
   (b)    <comop> is one of six operators:
                    EQ          equal
                    NE          not equal
                    LT          less than
                    GT          greater than
                    LE          less than or equal to
                    GE          greater than or equal to
   (c)    <container name> is the name of a STRING that appears once
          in each LIST-member.

The constant is truncated or padded with blanks on the right to
make it equal in size to the container to which it is being
compared.  Notice that padding on the right is not always
desirable (users will have control over the padding in a future
release).  In particular, care must be exercised when using
numbers in Version 0/9.  (A number represented as a STRING should
actually be described as a number; eventually it will be possible
to do this).

7.2 FOR-body

Two operations are legitimate in a FOR-body:  FOR and assignment.

These are subject to the restrictions discussed in Section 6.1 and
in the introduction to Section 7.  The restrictions are related to
three requirements:  (1) that the names be recognizable (see
Section 9 for details), (2) that a request be consistent regarding
direction of data transfer between containers, both within itself
and with the MODE of outermost containers, and (3) that transfers
occur between objects, not sets of objects.  The first two
requirements are permanent, but will become weaker in later
versions of the language.  The last requirement is temporary and
will be present only in early versions.

Due to an implementation problem associated with Version 0/9,
there is a somewhat bizarre restriction applied to references made
in the body of a loop nested in another loop.  This restriction is
not expected to pose any practical problems for users, and is not
part of the language design, but is discussed here for
completeness.

The restriction is most easily understood by example:

given the description
    F LIST
        R STRUCT
            A STR (3)
            BL LIST (3)
                    B STR (3)
            C STR (3)
            END
and the request fragment:
    FOR ...,R
        FOR ...,B
            ... = A ;
            ... = C ;
            END
        END

observe:
(a)    The outer loop processes the set of R's in F.
(b)    For each R in F, the inner loop processes the set of B's in
       the BL contained in that R.
(c)    In the body of the inner loop, there are references to A and
       C, which do not occur in B, but do occur in R.  That is, the
       objects referenced in the inner loop body are defined by the
       outer loop, not the inner loop.  In general, this is fine;
       in the case of C, however, we have a problem.
(d)    C occurs beyond the end of BL.

The 0/9 compiler is capable of neither (1) looking ahead enough to
locate C before it compiles code for the loop, nor (2) while
generating code to loop on the B's in BL, generating a separate
body of code that skips to the end of BL to locate C.  Thus it can
handle A, which has been located before it begins loop generation,
but it cannot handle C, which requires a little foresight.

The request fragment shown would not cause problems if the
description were changed to:

```
    F LIST
          R STRUCT
                  A STR (3)
                  C STR (3)
                  BL LIST (3)
                        B STR (3)
                  END
```

Then both A and C would have been found before code for the inner
loop was generated.

8.  Data Transmission

   Data is transferred from container to container by execution of
   assignment and FOR operations.  The outermost containers involved in
   transfers can be files or ports.  If both are files, then the
   transfer is internal to the Datacomputer.  If either is a port, then
   an address in the external world is needed to accomplish the data
   transmission.

Such an address is supplied through a CONNECT-request, which
associates a container (having a function of PORT) with an external
address:
    CONNECT <ident> TO <address> ;
Here <address> is either a specifications of host and socket number,
or a TENEX file designator (for CCA's TENEX) enclosed in single
quotes.  The host and socket form is:
    <socket> AT <host>
where <socket> is a decimal number, and <host> is either a decimal
number or a standard host name (since standard host names don't exist
right now, it has to be the TENEX 'standard' name for the host.
Contact the author for the latest list.)  If <host> is omitted, it is
taken to be the host from which the Datalanguage is being
transmitted.

The address associated with a port can be changed by issuing another
CONNECT-request.

A DISCONNECT-request simply breaks an existing port/address
association without establishing a new one.  (A CLOSE-request that
references an open port executes a DISCONNECT.)  The syntax of
DISCONNECT is:
    DISCONNECT <ident> ;

A port is disconnected when: (a) no successful CONNECT-request has
ever been issued for it, or (b) a DISCONNECT for the port has been
executed since the last successful CONNECT.

When a disconnected port is referenced in an assignment, it is
connected by default either to:
(a) the connection used for the transmission of Datalanguage to the
        Datacomputer, or
(b) the connection used for the transmission of Datacomputer
        diagnostics to the user
The choice between (a) and (b), of course, depends on whether the
reference is for input or output.  These connections are established
by the network user's ICP to the Datacomputer at the beginning of the
session.

Note that CONNECT and DISCONNECT do not open files or network
connections.  They simply make address associations within the
Datacomputer.  The files and connections are opened before each
request and closed after each request.

9.  Names in Datalanguage

   A name is recognized when it has been associated with a particular
   data container or set of containers.

   Datalanguage has mechanisms for the recognition of names in contexts.
   That is, the meaning of the name is related to where it appears.

   This makes it possible to attach natural meanings to partially
   qualified names.

   For example:

```
    WEATHER FILE LIST
          STATION STRUCT
                CITY STR (15)
                STATE STR (15)
                DATA LIST (24)
                      OBSERVATION STRUCT
                            HOUR STR (2)
                            TEMPERATURE STR (3)
                            HUMIDITY STR (2)
                            PRESSURE STR (4)
                            END
                END

    RESULTS PORT LIST
          RESULT STRUCT
                CITY STR (15)
                HOUR STR (2)
                TEMPERATURE STR (3)
                END

    FOR STATION WITH STATE EQ 'CALIFORNIA'
          FOR RESULT, OBSERVATION WITH HOUR GT '12'
                      AND HUMIDITY LT '75'
                CITY = CITY ;
                HOUR = HOUR ;
                TEMPERATURE = TEMPERATURE ;
                END ;
          END ;
```

   in the assignment 'CITY = CITY', the first CITY is understood to be
   RESULT.CITY and the second is understood to be STATION.CITY.

9.1 Informal Presentation of Recognition Rules

      'Ident' is used in the sense of section 3.  For example, in the
      description:
         F FILE LIST R STRUCT A STR (1) B STR (1) END
         F, R, A and B are idents.

      A context is a tree whose nodes are idents.  In such a tree, the
      terminal nodes are idents of STRINGs.  The ident of a LIST is
      superior to the ident of the LIST-member.  The ident of a STRUCT
      is superior to the idents of the STRUCT elements.  The context
      whose top node is F is said to be the context of F.


```
                +-----+
                !  F  !
                +-----+
                   !
                   !
                +-----+
                !  R  !
                +-----+
                   !
                   !
         +----------+----------+
         !                     !
         !                     !
      +-----+               +-----+
      !  A  !               !  B  !
      +-----+               +-----+
```
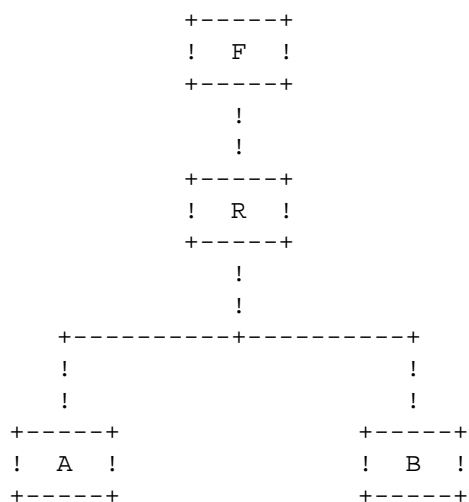

       Figure 9.1-1  The context of F

      A pathname is a sequence of idents, naming nodes along a path from
      one node to another.  A full pathname in the context starts at the
      topmost node.  Thus F.R.B is a full pathname in the context of F.
      A partial pathname starts at a node other than the topmost node
      (e.g.  R.B, B).

      In Datalanguage, pathnames omitting intermediate nodes, such as
      F.B (which omits 'R'), are not permitted.  Thus partial pathnames
      are partial only in that additional names are implied on the left.

Three attempts at recognition of a pathname, PN, in a context, CX,
are made:

(a)    recognition of PN as a full pathname in CX
(b)    recognition of PN as a partial pathname in which only the
       topmost node of CX is omitted
(c)    recognition of PN as an arbitrary partial pathname occurring
       only once in CX.

The attempts are made in the above order, and the recognition
process halts with the first successful attempt.

As an example, consider the description:

```
    F FILE LIST
         R STRUCT
                A STR
                B STR
                S STRUCT
                      R STR
```

which defines the context in Figure 9.1-2.

```
                        +-----+
                        !  F  !
                        +-----+
                           !
                           !
                        +-----+
                        !  R  !
                        +-----+
                           !
                           !
          +---------------+---------------+
          !               !               !
          !               !               !
       +-----+         +-----+         +-----+
       !  A  !         !  B  !         !  S  !
       +-----+         +-----+         +-----+
                                          !
                                          !
                                       +-----+
                                       !  R  !
                                       +-----+
```
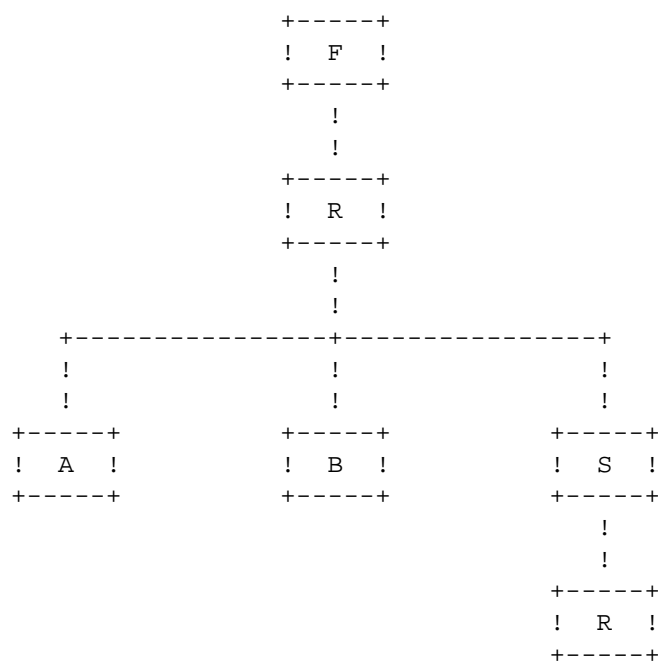
Figure 9.1-2      Example Context

In this context, F.R.A is a full pathname.  Thus, F.R.A is
recognized in attempt (a).  R is a partial pathname in which only
the topmost node is omitted.  Thus R is recognized in attempt (b).
Note carefully that R is recognized as a reference to F.R, not to
F.R.S.R.  Finally, B is an arbitrary partial pathname occurring
only once in the context.  Thus B is recognized in attempt (c).

Two stacks of contexts are maintained:  one for names used in an
input sense, and one for names used in an output sense.  When a
name is to be recognized, it is first decided whether the
reference is an input reference or an output reference.  An input
reference is (a) the right hand operand of an assign, or (b) a
name in the input set spec of a FOR.  An output reference is (a)
the left operand of an assign, or (b) the output operand of a FOR.
The first context on the appropriate context stack is then
searched, according to the procedure outlined on the previous
page.  If the name is neither recognized nor ambiguous in that
context, search continues in the next context on the stack.  If
the name can be recognized in none of the contexts on the
appropriate stack, it is unrecognizable.

When a stack is empty, the recognition procedure is different.
The search is carried on in a special context: The context of
%OPEN.  Its top node, %OPEN, is a built in system ident.
Subordinate to %OPEN is a context for each open directory node.
Each such context represents all the idents defined in the
directory nodes having data descriptions:
    F FILE LIST R STRUCT A STR (1) B STR (1)
and:
    P PORT LIST R STRUCT A STR (1) B STR (1)
then the context of %OPEN would be as in Figure 9.1-3.

```
                              +-------+
                              ! %OPEN !
                              +-------+
                                  !
                                  !
                   +-----------+-----------+
                   !                       !
                   !                       !
              +-----+                 +-----+
              !  A  !                 !  S  !
              +-----+                 +-----+
                 !                       !
                 !                       !
              +-----+                 +-----+
              !  R  !                 !  R  !
              +-----+                 +-----+
                 !                       !
                 !                       !
           +-----+-----+           +-----+-----+
           !           !           !           !
           !           !           !           !
        +-----+     +-----+     +-----+     +-----+
        ! A !       ! B !       ! A !       ! B !
        +-----+     +-----+     +-----+     +-----+
```
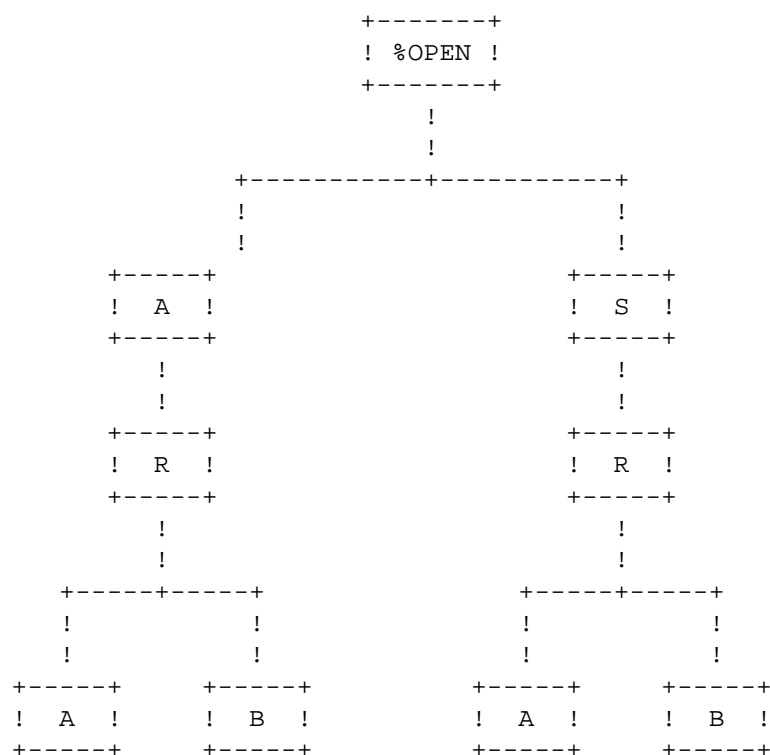
            Figure 9.1-3      The Context of %OPEN

     When a directory node is closed, the corresponding context is
     removed from the context of %OPEN.  When a node is opened, the
     associated context is added as the rightmost context subordinate
     to %OPEN.

9.2 Context Stack Maintenance

     The context stacks are always empty between requests.  The FOR-
     operator adds entries to the stacks.  FOR A adds the context of A
     to the input context stack.  FOR A, B ... adds the context of A to
     the output stack and the context of B to the input stack.

     When adding to an empty stack, FOR adds two contexts instead of
     one.  The second addition to the stack is the context of the
     looparg; the first addition is the context of the outermost
     container which encloses the looparg.

For example, given a context of %OPEN as in Figure 9.1-3, and
empty context stacks, the fragment 'FOR F.R ...' adds two
contexts: (1) the context of F, and (2) the context of F.R.

Contexts are removed from the stacks by the END matching the FOR
which added them.

10.  Miscellaneous Topics

10.1 The LIST-request

The LIST-request provides a means for the user to inspect system
data of interest to him.  The user can determine the contents of
the directory, the source or parsed form of any data description
in the system, and the mode and connection status of any open file
or port.

The LIST operator has a single operand, which can have any of
several forms.  The action of the operator is to output a list of
values on the Datalanguage output port.

To output a list of pathnames, the operand used is '%ALL'.  When
'%ALL' appears alone, all pathnames in the directory are listed.
When '%ALL' appears after the last ident in a directory pathname,
the full pathnames of all nodes subordinate to the named node are
listed.

To output one or more source descriptions, the operand '.%SOURCE'
is used.  '.%SOURCE' is preceded with one of (a) '%ALL', (b)
'%OPEN', or (c) the ident of an open outermost container.  The
output is either (a) all descriptions, (b) all open descriptions,
or (c) a particular open description.

To output a parsed description, the operand '.%DESC' is used
('%DESCRIPTION' is also accepted).  This operand is preceded
either with (a) '%OPEN', or (b) the ident of an open outermost
container.

Examples:

     Let P be the ident of an open PORT.  Let A.B.C be a
     directory pathname.
         LIST %ALL
         LIST A.B.C.%ALL
         LIST %OPEN
         LIST %ALL.%SOURCE
         LIST %OPEN.%SOURCE
         LIST P.%SOURCE
         LIST %OPEN.%DESC
         LIST P.%DESC

     Note that 'LIST A.B.C.%SOURCE' is not legal - '.%SOURCE'
     must be preceded with the ident of an open container, not a
     pathname.  A similar restriction applies to '.%DESC'.

10.2 Constants in Datalanguage

     A constant of type STRING can be included in a Datalanguage
     request by enclosing it in single quotes:
         'ABC'
     A single quote is included in a constant by preceding it with a
     double quote:
         'FATHER"'S'
     Likewise, a double quote is included by preceding it with a double
     quote:
         'JOHN SAID ""HELLO""'

     Such constants can be used on the right of comparison operators
     and of assignment operators which reference strings.

     Eventually, Datalanguage will contain facilities for the inclusion
     of constants of all datatypes; such constants are simply a special
     case of the Datalanguage expression and will be permitted wherever
     such expressions are permitted.

10.3 Character Set

     Internally, Version 0/9 of the Datacomputer software operates in
     7-bit ASCII characters.  Its output to the ARPANET is converted to
     8-bit ASCII.  On input from the ARPANET, it expects 8-bit
     characters, which it converts to 7-bit characters.

To convert from 7- to 8-bit characters, a '0' bit is prefixed.  To
convert from 8- to 7-bit characters, the high-order bit is
checked.  If the high-order bit is a '0', the bit is discarded and
the character is accepted as a 7-bit character.  If the high-order
bit is a '1', then the character is discarded.

(In the following discussion, as elsewhere in this memo, all
references to numerical character codes are in decimal).

The remainder of this section discusses treatment of codes 0
through 127, when they appear in Datalanguage requests.

In general, printing characters are acceptable in requests, while
control characters are not.  There are some exceptions, which are
detailed below.  The printing characters are codes 32-126.  The
control characters are codes 0-31 and 127.

Certain control characters are accepted:

   Tab(9) is accepted freely in requests.  It functions as a
   separator (explained below).

   EOL(31), meaning end-of-line, is accepted in requests,
   functioning both as a separator and an activator (a).  EOL has
   a special meaning in data, and should not be introduced into
   STRING constants(***).

   Control-L(12) is an activator and a high-level request
   delimiter.  It terminates the test of any request being
   processed when it is encounter in the input stream.  It is
   useful in Datacomputer-user program synchronization.

   Control-Z(26) means end-of-session when encountered in
   Datalanguage.  It has the properties of control-L, and in
   addition, causes the Datacomputer to execute an end-of-session
   procedure, which results in closing the Datalanguage
   connections, closing any open files or ports, etc.  The effect
   is equivalent to a [LOGOUT(which] does not exist yet).

   The two-character sequence <carriage return(13), line feed(10)>
   is equivalent to EOL (and is translated to EOL on input from
   the network).  The reverse sequence, as well as either
   character alone, is treated simply as other control characters
   (ignored).

All other control characters are ignored.

The printing characters are further divided into four groups:
special characters, uppercase letters, lower case letters, and
digits (the membership of these groups is defined in section 11).

Corresponding upper and lower case letters are equivalent in
requests, except with quoted strings.

Certain special characters have a lexical function, which is
either break or separator.  A break character terminates the
current lexical item and returned itself as the next item.  A
separator character terminates the current item but does not begin
a new item (i.e., its only function is to separate items).
Multiple separators are equivalent to a single separator.  A
separator can always be inserted before or after a break
character, without altering the meaning of the request.

The separators are tab(9), space(32), and end-of-line(31).

The break characters are left parenthesis(40), right
parenthesis(41), equals(61), semicolon(59), period(46), comma(44),
quote(39), and slash(47).

(a)    An activator character causes the Datacomputer to process
       whatever has been received since the previous activator or
       the beginning of the request.  The meaning of a request is
       independent of the presence/absence of activators.  However,
       a request will not be executed until an activator beyond the
       end of the request is received.

While Version 0/9 defines (carriage return, linefeed) in terms of
EOL, network users should not think in terms of sending EOL's to
the Datacomputer over the network.  EOL is not part of the network
ASCII character set, and has no definite permanent place in
Datacomputer implementation plans.

10.4 Comments

Comments can be included in Datalanguage requests.  A comment is
begun with the two-character sequence '/*', and ended with the
two-character sequence '*/'.  Since slash is a break character, a
comment does cause a lexical break; its overall effect is that of
a separator.

10.5 Reserved Identifiers

      Certain identifiers are reserved in Datalanguage, and should not
      be used to name containers or directory nodes.  These are:

      AND
      APPEND
      AT
      CLOSE
      CONNECT
      CREATE
      DELETE
      DISCONNECT
      END
      EQ
      FILE
      FOR
      GE
      GT
      LE
      LIST
      LT
      NODE
      NE
      NOT
      OPEN
      OR
      PORT
      READ
      STR
      STRUCT
      TO
      WITH
      WRITE

      More reserved identifiers will be added in Version 0/10.

11.  Datalanguage Syntax Expressed in BNF

11.1 Requests

```
     11.1.01  <request>  ::=  ;
     11.1.02  <request>  ::=  <create>
     11.1.03  <request>  ::=  OPEN <pn>  ;
     11.1.04  <request>  ::=  OPEN <pn> <mode>  ;
     11.1.05  <request>  ::=  CLOSE <ident>  ;
     11.1.06  <request>  ::=  CONNECT <ident> TO <address>  ;
     11.1.07  <request>  ::=  DISCONNECT <ident>  ;
     11.1.08  <request>  ::=  MODE <ident> <mode>  ;
     11.1.09  <request>  ::=  DELETE <pn>  ;
     11.1.10  <request>  ::=  LIST <listarg>  ;
     11.1.11  <request>  ::=  <sr-request>  ;
```

11.2 Data Description and Creation

```
     11.2.01  <create>  ::=  CREATE <pn>
     11.2.02  <create>  ::=  CREATE <pn> <ftn> LIST <desc>
     11.2.03  <create>  ::=  CREATE <pn> <ftn> LIST <size> <desc>

     11.2.04  <desc>  ::=  <ident> <attributes>

     11.2.05  <attributes>  ::=  LIST <size> <desc>
     11.2.06  <attributes>  ::=  STRUCT <descs> END
     11.2.07  <attributes>  ::=  STR <size>
     11.2.08  <attributes>  ::=  STR <size> ,I=D

     11.2.09  <descs>  ::=  <desc>
     11.2.10  <descs>  ::=  <descs> <desc>

     11.2.11  <ftn>  ::=  PORT
     11.2.12  <ftn>  ::=  FILE
     11.2.13  <ftn>  ::=  TEMP PORT
     11.2.14  <ftn>  ::=  TEMPORARY PORT

     11.2.15  <size>  ::=  (  <integer constant>  )
```

11.3 Data Storage and Retrieval

```
     11.3.01  <sr-request>  ::=  <assign>
     11.3.01  <sr-request>  ::=  <loop>

     11.3.03  <assign>  ::=  <pn> = <object>

     11.3.04  <loop>  ::=  FOR <looparg> <loopbody> END

     11.3.05  <looparg>  ::=  <exp>
     11.3.06  <looparg>  ::=  <pn> , <exp>

     11.3.07  <loopbody>  ::=  <sr-request>
     11.3.08  <loopbody>  ::=  <loopbody1> <sr-request>
     11.3.09  <loopbody>  ::=  <loopbody1>

     11.3.10  <loopbody1>  ::=  <sr-request>  ;
     11.3.11  <loopbody1>  ::=  <loopbody1> <sr-request>  ;
```

11.4 Expressions

```
     11.4.01  <exp>  ::=  <pn>
     11.4.02  <exp>  ::=  <pn> WITH <boolexp>

     11.4.03  <boolexp>  ::=  <pn> <comop> <string constant>
     11.4.04  <boolexp>  ::=  ( <boolexp> )
     11.4.05  <boolexp>  ::=  NOT <boolexp>
     11.4.06  <boolexp>  ::=  <boolexp> AND <boolexp>
     11.4.07  <boolexp>  ::=  <boolexp> OR <boolexp>

     11.4.08  <comop>  ::=  EQ
     11.4.09  <comop>  ::=  NE
     11.4.10  <comop>  ::=  GT
     11.4.11  <comop>  ::=  LT
     11.4.12  <comop>  ::=  GE
     11.4.13  <comop>  ::=  LE
```

11.5 Miscellaneous

```
11.5.01  <address>  ::=  <quote> <TENEX file designator>
         <quote>
11.5.02  <address>  ::=  <socket> AT <host>
11.5.03  <address>  ::=  <socket>

11.5.04  <socket>  ::=  <integer constant>  //INTERPRETED AS
         DECIMAL

11.5.05  <host>  ::=  <integer constant>  //INTERPRETED AS
         DECIMAL
11.5.06  <host>  ::=  *****  TENEX host names *****

11.5.07  <object>  ::=  <pn>
11.5.08  <object>  ::=  <string constant>

11.5.09  <mode>  ::=  READ
11.5.10  <mode>  ::=  APPEND
11.5.11  <mode>  ::=  WRITE

11.5.12  <listarg>  ::=  %ALL
11.5.13  <listarg>  ::=  <pn>.%ALL
11.5.14  <listarg>  ::=  %OPEN
11.5.15  <listarg>  ::=  %ALL.%SOURCE
11.5.16  [<LISTARG>]  ::=  <IDENT>.%SOURCE
11.5.17  <listarg>  ::=  %OPEN.%SOURCE
11.5.18  <listarg>  ::=  %OPEN.%DESC
11.5.19  <listarg>  ::=  <ident>.%DESC

11.5.20  <pn>  ::=  <ident>
11.5.21  <pn>  ::=  <pn>.<ident>

11.5.22  <ident>  ::=  <letter>
11.5.23  <ident>  ::=  <ident> <letter>
11.5.24  <ident>  ::=  <ident> <digit>

11.5.25  <integer constant>  ::=  <digit>
11.5.26  <integer constant>  ::=  <integer constant> <digit>

11.5.27  <string constant>  ::=  <quote> <string conbody>
         <quote>

11.5.28  <string conbody>  ::=  <nonquote>
11.5.28  <string conbody>  ::=  <string conbody> <nonquote>
```

11.6 Character Set

```
     11.6.01  <separator>  ::=   //SPACE(32)
     11.6.02  <separator>  ::=   //TAB(9)
     11.6.03  <separator>  ::=   <eol>

     11.6.04  <special>  ::=  <quote>
     11.6.05  <special>  ::=  <superquote>
     11.6.06  <special>  ::=  <special1>

     11.6.07  <letter>  ::=  A
     11.6.08  <letter>  ::=  B
      ...............
     11.6.09  <letter>  ::=  Z

     11.6.10  <letter>  ::=  a
     11.6.11  <letter>  ::=  b
      ...............
     11.6.12  <letter>  ::=  z

     11.6.13  <digit>  ::=  0
     11.6.14  <digit>  ::=  1
      ...............
     11.6.15  <digit>  ::=  9

     11.6.16  <nonquote>  ::=  <letter>
     11.6.17  <nonquote>  ::=  <digit>
     11.6.18  <nonquote>  ::=  <superquote> <quote>
     11.6.19  <nonquote>  ::=  <superquote> <superquote>
     11.6.20  <nonquote>  ::=  <special1>
     11.6.21  <nonquote>  ::=  <separator>

     11.6.22  <eol>  ::=  //EOL (31)
     11.6.23  <eol>  ::=  <carriage return> <line feed>

     11.6.24  <carriage return>  ::=  //CARRIAGE RETURN (13)
     11.6.25  <line feed>  ::=  //LINE FEED (10)

     11.6.26  <quote>  ::=  ' //SINGLE QUOTE(44)

     11.6.27  <superquote>  ::=  " //DOUBLE QUOTE(34)

     11.6.28  <special1>  ::=  ! //EXCLAMATION POINT(33)
     11.6.29  <special1>  ::=  # //NUMBER SIGN(35)
     11.6.30  <special1>  ::=  $ //DOLLAR SIGN(36)
     11.6.31  <special1>  ::=  % //PERCENT SIGN(37)
     11.6.32  <special1>  ::=  & //AMPERSAND(38)
     11.6.33  <special1>  ::=  ( //LEFT PARENTHESIS(40)
```

```
11.6.34  <special1>  ::=  ) //RIGHT PARENTHESIS(41)
11.6.35  <special1>  ::=  * //ASTERISK(42)
11.6.36  <special1>  ::=  + //PLUS SIGN(43)
11.6.37  <special1>  ::=  , //COMMA(44)
11.6.38  <special1>  ::=  - //MINUS SIGN(45)
11.6.39  <special1>  ::=  . //PERIOD(46)
11.6.40  <special1>  ::=  / //SLASH(47)
11.6.41  <special1>  ::=  : //COLON(58)
11.6.42  <special1>  ::=  ; //SEMICOLON(59)
11.6.43  <special1>  ::=  < //LEFT ANGLE BRACKET(60)
11.6.44  <special1>  ::=  = //EQUAL SIGN(61)
11.6.45  <special1>  ::=  > //RIGHT ANGLE BRACKET(62)
11.6.46  <special1>  ::=  ? //QUESTION MARK(63)
11.6.47  <special1>  ::=  @ //AT-SIGN(64)
11.6.48  <special1>  ::=  [ //LEFT SQUARE BRACKET(91)
11.6.49  <special1>  ::=   //BACK SLASH(92)
11.6.50  <special1>  ::=  ] //RIGHT SQUARE BRACKET(93)
11.6.51  <special1>  ::=  ^ //CIRCUMFLEX(94)
11.6.52  <special1>  ::=  _ //UNDERBAR(95)
11.6.53  <special1>  ::=  ` //ACCENT GRAVE(96)
11.6.54  <special1>  ::=  { //LEFT BRACE(123)
11.6.55  <special1>  ::=  | //VERTICAL BAR(124)
11.6.56  <special1>  ::=  } //RIGHT BRACE(125)
11.6.57  <special1>  ::=  ~ //TILDE(126)

<EOF>
```

[ This RFC was put into machine readable form for entry ]
[ into the online RFC archives by Walter Benton 12/98 ]