Network Working Group   Bob Anderson
Request for Comments: 166   Rand
NIC 6780   Vint Cerf
                                                              UCLA
                                                      Eric Harslem
                                                      John Haefner
                                                              Rand
                                                       Jim Madden
                                                   U. of Illinois
                                                     Bob Metcalfe
                                                              MIT
                                                    Arie Shoshani
                                                              SDC
                                                       Jim White
                                                             UCSB
                                                      David Wood
                                                            Mitre
                                                      25 May 1971

DATA RECONFIGURATION SERVICE -- AN IMPLEMENTATION SPECIFICATION

CONTENTS

I. INTRODUCTION

PURPOSE OF THIS RFC

The Purpose of this RFC is to specify the Data Reconfiguration
Service (DRS.)

The DRS experiment involves a software mechanism to reformat Network
data streams. The mechanism can be adapted to numerous Network
application programs. We hope that the result of the experiment will
lead to a future standard service that embodies the principles
described in this RFC.

MOTIVATION

Application programs require specific data I/O formats yet the
formats are different from program to program. We take the position
that the Network should adapt to the individual program requirements
rather than changing each program to comply with a standard. This
position doesn't preclude the use of standards that describe the
formats of regular message contents; it is merely an interpretation
of a standard as being a desirable mode of operation but not a
necessary one.

In addition to differing program requirements, a format mismatch
problem occurs where users wish to employ many different kinds of
consoles to attach to a single service program. It is desirable to
have the Network adapt to individual console configurations rather
than requiring unique software packages for each console
transformation.

One approach to providing adaptation is for those sites with
substantial computing power to offer a data reconfiguration service;
this document is a specification of such a service.

The envisioned modus operandi of the service is that an applications
programmer defines _forms_ that describe data reconfigurations. The
service stores the forms by name. At a later time, a user (perhaps a
non-programmer) employs the service to accomplish a particular
transformation of a Network data stream, simply by calling the form
by name.

We have attempted to provide a notation tailored to some specifically
needed instances of data reformatting while keeping the notation and
its underlying implementation within some utility range that is
bounded on the lower end by a notation expressive enough to make the
experimental service useful, and that is bounded on the upper end by
a notation short of a general purpose programming language.

II.   OVERVIEW OF THE DATA RECONFIGURATION SERVICE

ELEMENTS OF THE DATA RECONFIGURATION SERVICE

An implementation of the Data Reconfiguration Service (DRS) includes
modules for connection protocols, a handler of some requests that can
be made of the service, a compiler and/or interpreter (called the
Form Machine) to act on those requests, and a file storage module for
saving and retrieving definitions of data reconfigurations (forms).

This section describes connection protocols and requests. The next
section covers the Form Machine language in some detail. File
storage is not described in this document because it is transparent
to the use of the service an its implementation is different at each
DRS host.

CONCEPTUAL NETWORK CONNECTIONS

There are three conceptual Network connections to the DRS, see Fig.
1.

> 1) The control connection (CC) is between an originating user
>   and the DRS. Forms specifying data reconfigurations are
>   defined over this connection. The user indicates (once)
>   forms to be applied to data passing over the two
>   connections described below.
>
> 2) The user connection (UC) is between a user process and the
>   DRS.

        3) The server connection (SC) is between the DRS and the
            serving process.

    Since the goal is to adapt the Network to user and server processes,
    a minimum of requirements are imposed on the UC and SC.

```
+------------+        +------+        +---------+
| ORIGINATING|   CC   | DRS  |   SC   | SERVER  |
| USER       |--------|      |--------| PROCESS |
+------------+        +------+        +---------+
             ^          /   ^                ^
             |      UC / <-----\             |
             |        /         \            |
             |    +------------+  \          |
TELNET ------+    | USER       |   \|  +-- Simplex or Duplex
Protocol          | PROCESS    |       Connections
Connection        +------------+
```
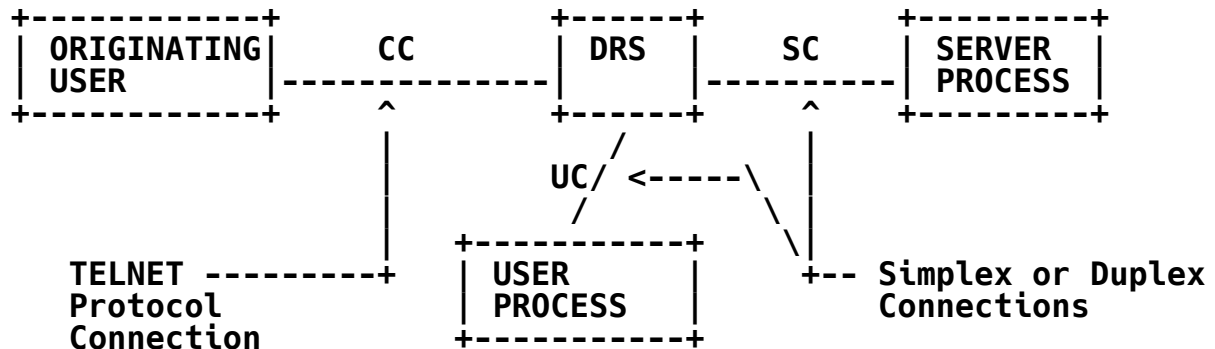
              Figure 1. DRS Network Connections

CONNECTION PROTOCOLS AND MESSAGE FORMATS

    Over a control connection the dialog is directly between an
    originating user and the DRS. Here the user is defining forms or
    assigning predefined forms to connections for reformatting.

    The user connects to the DRS via the standard initial connection
    protocol (ICP). Rather than going through a logger, the user calls
    on a particular socket on which the DRS alway listens. (Experimental
    socket numbers will be published later.) DRS switches the user to
    another socket pair.

    Messages sent over a control connection are of the types and formats
    specified for TELNET. (The data type code should specify ASCII --
    the default.) Thus, a user at a terminal should be able to connect
    to a DRS via his local TELNET, for example, as shown in Fig. 2.

```
                    +---------+   CC  +---------+
          +---------| TELNET  |-------|   DRS   |
          |         +---------+       +---------+
   +-----------------------+
   |        USER           |
   | (TERMINAL OR PROGRAM) |
   +-----------------------+
```
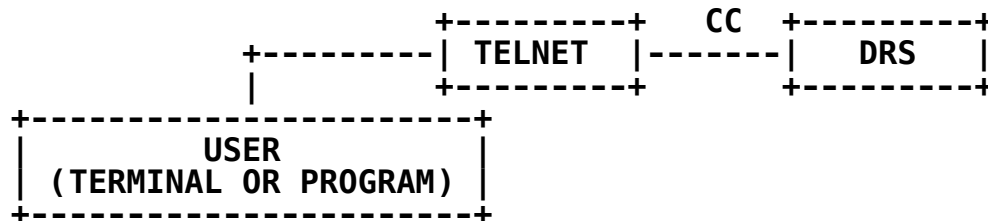
              Figure 2. A TELNET Connection to DRS

When a user connects to DRS he supplies a six-character user ID (UID) as a qualifier to guarantee the uniqueness of his form names. He will initially have the following commands:

    1. DEFFORM (form)
    2. ENDFORM (form)

        These two commands define a form, the text of which is chronologically entered between them. The form is stored in the DRS local file system.

    3. PURGE (form)

        The named form, as qualified by the current UID, is purged from the DRS file system.

    4. LISTNAMES (UID)

        The unqualified names of all forms assigned to UID are returned.

    5. LISTFORM (form)

        The source text of a named form is returned.

    6. DUPLEXCONNECT (user site, user receive socket, user method, server site, server receive socket, server method, user-to-server form name, server-to-user form name)

        A duplex connection is made between two processes using the receive sockets and the sockets one greater. Method is defined below. The forms define the transformations on these connections.

    7. SIMPLEXCONNECT (user site, user socket, user method, server site, server socket, server method, form)

        A simplex connection is made between the two sockets as specified by method.

    8. ABORT (site, receive socket)

        The reconfiguration of data is terminated by closing both the UC and SC specified in part in the command.

Either one, both, or neither of the two parties specified in 6 or 7 may be at the same host as the party issuing the request. Sites and sockets specify user and server for the connection. Method indicates

the way in which the connection is established.

The following rules apply to these commands:

1) Commands may be abbreviated to the minimum number of
   characters to identify them uniquely.

2) All commands should be at the start of a line.

3) Parameters are enclosed in parentheses and separated by
   commas.

4) Imbedded blanks are ignored.

5) The parameters are:

   form name   1-6 characters
   UID   1-6 characters
   Site   1-2 characters specifying
                       the hexadecimal host number
   Socket   1-8 characters specifying the
                       hexadecimal socket number
   Method   A single character

6) Method has the following values:

   C   The site/socket is already connected
            to the DRS as a dummy control connection
            (should not be the real control connection).
   I   Connect via the standard ICP (does not
            apply to SIMPLEXCONNECT).
   D   Connect directly via STR, RTS.

   The DRS will make at least the following minimal
   responses to the user:

   1) A positive or negative acknowledgement after
       each line (CR/LF)
   2) If a form fails or terminates
   TERMINATE, ASCII Host # as hex, ASCII Socket # as hex,
           ASCII Return Code as decimal
   thus identifying at least one end of the connection.

EXAMPLE CONNECTION CONFIGURATIONS

    There are basically two modes of DRS operation: 1) the user wishes to
    establish a DRS UC/SC connection(s) between the programs and 2) the
    user wants to establish the same connection(s) where he (his
    terminal) is at the end of the UC or the SC. The latter case is
    appropriate when the user wishes to interact from his terminal with
    the serving process (e.g., a logger).

    In the first case (Fig. 1, where the originating user is either a
    terminal or a program) the user issues the appropriate CONNECT
    command. The UC/SC can be simplex or duplex.

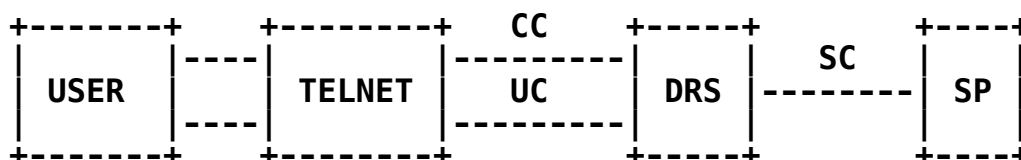   The second case has two possible configurations, shown in Figs. 3 and
    4.

```
+-------+     +--------+  CC    +-----+        +----+
|       |----|        |--------|     |   SC   |    |
| USER  |    | TELNET | UC     | DRS |--------| SP |
|       |----|        |--------|     |        |    |
+-------+     +--------+        +-----+        +----+
```

              Figure 3.   Use of Dummy Control Connection

```
            +---------+
+------+   /| USER    |  CC    +-----+
|      |--/ | SIDE    |--------|     |   SC   +----+
| USER |    +---------+ UC     | DRS |--------| SP |
|      |--\ | SERVING |--------|     |        +----+
+------+   \| SIDE    |        +-----+
            +---------+
```
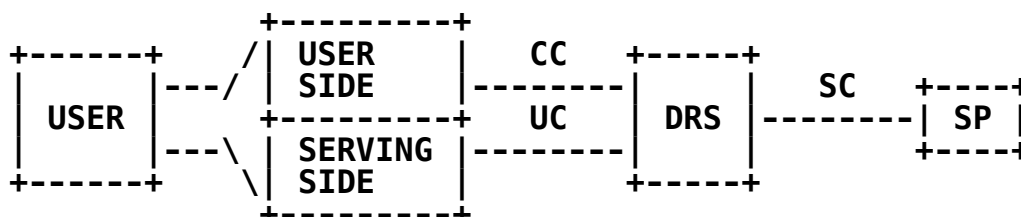
              Figure 4.   Use of Server TELNET

    In Fig. 3 the user instructs his TELNET to make two duplex
    connections to DRS. One is used for control information (the CC) and
    the other is a dummy. When he issues the CONNECT he references the
    dummy duplex connection (UC) using the "already connected" option.

    In Fig. 4 the user has his TELNET (user side) call the DRS. When he
    issues the CONNECT the DRS calls the TELNET (server side) which
    accepts the call on behalf of the console. This distinction is known
    only to the user since to the DRS the configuration Fig. 4 appears
    identical to that in Fig. 1. Two points should be noted:

        1) TELNET protocol is needed only to define forms and direct
            connections. It is not required for the using and serving

               processes.
    2) The using and serving processes need only a minimum of
        modification for Network use, i.e., an NCP interface.

                        III.   THE FORM MACHINE

INPUT/OUTPUT STREAMS AND FORMS

    This section describes the syntax and semantics of forms that specify
    the data reconfigurations.   The Form Machine gets an input stream,
    reformats the input stream according to a form describing the
    reconfiguration, and emits the reformatted data as an output stream.

    In reading this section it will be helpful to envision the
    application of a form to the data stream as depicted in Fig. 5.   An
    input stream pointer identifies the position of data (in the input
    stream) that is being analyzed at any given time by a part of the
    form.   Likewise, an output stream pointer locates data being emitted
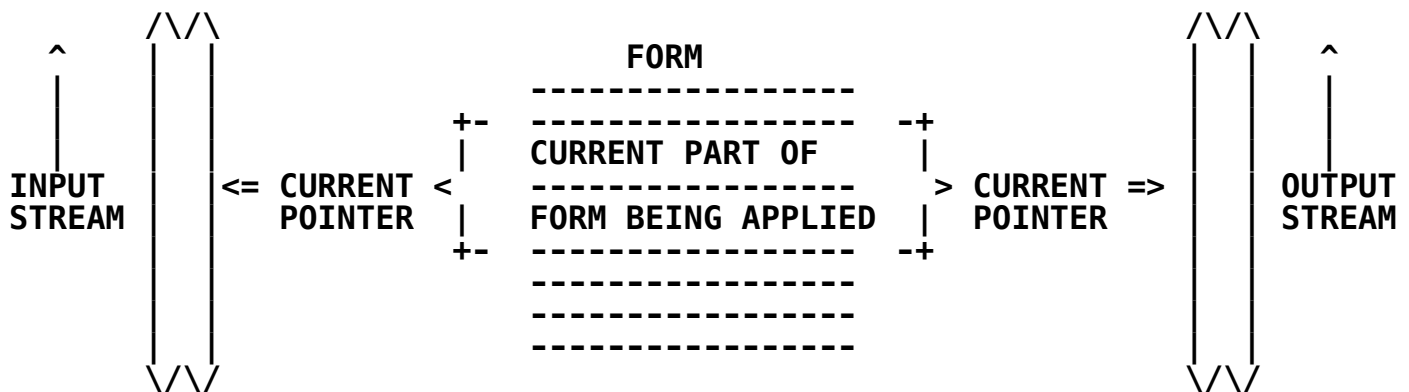   in the output stream.

```
                                        FORM
   ^       /\/\                    ------------------                     /\/\        ^
   |       |  |                +-  ------------------  -+                  |  |        |
   |       |  |                |   CURRENT PART OF      |                  |  |        |
INPUT      |  |  <= CURRENT <  ------------------  >  CURRENT =>          |  |    OUTPUT
STREAM     |  |     POINTER  |   FORM BEING APPLIED   |   POINTER         |  |    STREAM
   |       |  |                +-  ------------------  -+                  |  |        |
   |       |  |                    ------------------                     |  |        |
   |       |  |                    ------------------                     |  |        |
   |       |  |                    ------------------                     |  |        |
           \/\/                                                           \/\/
            Figure 5.   Application of Form to Data Streams
```

FORM MACHINE BNF SYNTAX

```
form             ::=  rule | rule form

rule             ;;=  label  inputstream  outputstream ;

label            ::=  INTEGER | <null>

inputstream      ::=  terms | <null>

terms            ::=  term | terms , term

outputstream     ::=  : terms | <null>

term             ::=  identifier | identifier  descriptor |
                      descriptor | comparator

identifier       ::=  an alpha character followed by 0 to 3
                      alphanumerics

descriptor       ::=  (replicationexpression , datatype ,
                      valueexpression , lengthexpression  control)

comparator       ::=  (value  connective  value  control)  |
                      (identifier  *<=*  control)

replicationexpression  ::=  # | arithmeticexpression | <null>

datatype         ::=  B | O | X | E | A

valueexpression  ::=  value | <null>

lengthexpression  ::=      arithmeticexpression | <null>

connective       ::=  .LE. | .LT. | .GE. | .GT. | .EQ. | .NE.

value            ::=  literal | arithmeticexpression

arithmeticexpression  ::=  primary | primary operator
                           arithmeticexpression

primary          ::=  identifier | L(identifier) | V(identifier) |
                      INTEGER

operator         ::=  + | - | * | /

literal          ::=  literaltype "string"
```

```
literaltype    ::=  B | O | X | E | A

string         ::=  from 0 to 256 characters

control        ::=  :  options | <null>

options        ::=  S(where) | F(where) | U(where) |
                    S(where) , F(where) |
                    F(where) , S(where)

where          ::=  arithmeticexpression | R(arithmeticexpression)
```

ALTERNATE SPECIFICATION OF FORM MACHINE SYNTAX

$$\text{form} \quad ::= \quad \{\text{rule}\}_1^{\text{infinity}}$$

$$\text{rule} \quad ::= \quad \{\text{INTEGER}\}_0^1 \quad \{\text{terms}\}_0^1 \quad \{:\text{terms}\}_0^1 ;$$

$$\text{terms} \quad ::= \quad \text{term} \{,\text{term}\}_0^{\text{infinity}}$$

$$\text{term} \quad ::= \quad \text{identifier} \mid \{\text{identifier}\}_0^1 \quad \text{descriptor}$$
$$\mid \text{comparator}$$

$$\text{descriptor} \quad ::= \quad (\{\text{arithmeticexpression}\}_0^1 , \text{datatype} ,$$
$$\{\text{value}\}_0^1 , \{\text{lengthexpression}\}_0^1 \{:\text{options}\}_0^1$$

$$\text{comparator} \quad ::= \quad (\text{value} \ \text{connective} \ \text{value} \{:\text{options}\}_0^1) \mid$$
$$(\text{identifier} .<=. \text{value} \{:\text{options}\}_0^1)$$

```
connective     ::=  .LE. | .LT. | .GE. | .GT. | .EQ. | .NE.

lengthexpression ::= # | arithmeticexpression

datatype       ::=  B | O | X | E | A

value          ::=  literal | arithmeticexpression
```

arithmeticexpression   ::=   primary {operator primary}$_0^{infinity}$

operator   ::= + | - | * | /

primary   ::=   identifier | L(identifier) |
                              V(identifier) | INTEGER

literal   ::=   literaltype "{CHARACTER}$_0^{256}$ "

literaltype   ::=   B | O | X | A | E

options   ::=   S(where) {,F(where)}$_0^1$ |

                              F(where) {,S(where)}$_0^1$ | U(where)

where   ::=   arithmeticexpression |
                              R(arithmeticexpression)

identifier   ::=   ALPHABETIC {ALPHAMERIC}$_0^3$

FORMS

   A form is an ordered set of rules.

        form ::= rule | rule form

   The current rule is applied to the current position of the input
   stream. If the (input stream part of a) rule fails to correctly
   describe the contents of the current input then another rule is made
   current and applied to the current position of the input stream. The
   next rule to be made current is either explicitly specified by the
   current term in the current rule or it is the next sequential rule by
   default. Flow of control is more fully described under TERM AND RULE
   SEQUENCING.

   If the (input stream part of a) rule succeeds in correctly describing
   the current input stream, then some data may be emitted at the
   current position in the output stream according to the rule. The
   input and output stream pointers are advanced over the described and
   emitted data, respectively, and the next rule is applied to the now
   current position of the input stream.

   Application of the form is terminated when an explicit return
   (R(arithmeticexpression)) is encountered in a rule. The user and

server connections are closed and the return code
(arithmeticexpression) is sent to the originating user.

RULES

A rule is a replacement, comparison, and/or an assignment operation
of the form shown below.

        rule ::= label inputstream outputstream

A label is the name of a rule and it exists so that the rule may be
referenced elsewhere in the form for explicit rule transfer of
control. Labels are of the form below.

        label ::= INTEGER | <null>

The optional integer labels are in the range 0 >= INTEGER >= 9999.
The rules need not be labeled in ascending numerical order.

TERMS

The inputstream (describing the input stream to be matched) and the
outputstream (describing data to be emitted in the output stream)
consist of zero or more terms and are of the form shown below.

        inputstream   ::= terms | <null>
        outputstream   ::= :terms | <null>
        terms   ::= term | terms , term

Terms are of one of four formats as indicated below.

        term ::= identifier | identifier descriptor |
                  descriptor | comparator

Term Format 1

The first term format is shown below.

        identifier

The identifier is a symbolic reference to a previously identified
term (term format 2) in the form. It takes on the same attributes
(value, length, type) as the term by that name. Term format 1 is
normally used to emit data in the output stream.

Identifiers are formed by an alpha character followed by 0 to 3
alphanumeric characters.

Term Format 2

    The second term format is shown below.

        identifier descriptor

    Term format 2 is generally used as an input stream term but can be
    used as an output stream term.

    A descriptor is defined as shown below.

        descriptor ::= (replicationexpression, datatype,
                        valueexpression, lengthexpression
                        control)

    The identifier is the symbolic name of the term in the usual
    programming language sense. It takes on the type, length, value, and
    replication attributes of the term and it may be referenced elsewhere
    in the form.

    The replication expression, if specified, causes the unit value of
    the term to be generated the number of times indicated by the value
    of the replication expression. The unit value of the term (quantity
    to be replicated) is determined from the data type, value expression,
    and length expression attributes. The data type defines the kind of
    data being specified. The value expression specifies a nominal value
    that is augmented by the other term attributes. The length
    expression determines the unit length of the term. (See the IBM SRL
    Form C28-6514 for a similar interpretation of the pseudo instruction,
    defined constant, after which the descriptor was modeled.)

    The replication expression is defined below.

        replicationexpression ::= # | arithmeticexpression | <null>
        arithmeticexpression ::= primary | primary operator
                            arithmeticexpression
        operator ::= + | - | * | /
        primary ::= identifier | L(identifier) | V(identifier) |
                    INTEGER

    The replication expression is a repeat function applied to the
    combined data type value, and length expressions. It expresses the
    number of times that the nominal value is to be repeated.

    The terminal symbol # means an arbitrary replication factor. It must
    be explicitly terminated by a match or non-match to the input stream.
    This termination may result from the same or the following term.

A null replication expression has the value of one. Arithmetic
expressions are evaluated from left-to-right with no precedence.

The L(identifier) is a length operator that generates a 32-bit binary
integer corresponding to the length of the term named. The
V(identifier) is a value operator that generates a 32-bit binary
integer corresponding to the value of the term named. (See
Restrictions and Interpretations of Term Functions.) The value
operator is intended to convert character strings to their numerical
correspondents.

The data type is defined below.

           datatype ::= B | O | X | E | A

The data type describes the kind of data that the term represents.
(It is expected that additional data types, such as floating point
and user-defined types, will be added as needed.)

Data Type	Meaning	Unit Length
B	Bit string	1 bit
O	Bit string	3 bits
X	Bit string	4 bits
E	EBCDIC character	8 bits
A	Network ASCII character	8 bits

The value expression is defined below.

           valueexpression ::= value | <null>
           value ::= literal | arithmeticexpression
           literal ::= literaltype "string"
           literaltype ::= B | O | X | E | A

The value expression is the nominal value of a term expressed in the
format indicated by the data type. It is repeated according to the
replication expression.

A null value expression in the input stream defaults to the data
present in the input stream. The data must comply with the datatype
attribute, however.

A null value expression generates padding according to Restrictions
and Interpretations of Term Functions.

The length expression is defined below.

         lengthexpression ::= arithmeticexpression | <null>

The length expression states the length of the field containing the
value expression.

If the length expression is less than or equal to zero, the term
succeeds but the appropriate stream pointer is not advanced.
Positive lengths cause the appropriate stream pointer to be advanced
if the term otherwise succeeds.

Control is defined under TERM AND RULE SEQUENCING.

Term Format 3

Term format 3 is shown below.

        descriptor

It is identical to term format 2 with the omission of the identifier.
Term format 3 is generally used in the output stream. It is used in
the input stream where input data is to be passed over but not
retained for emission or later reference.

Term Format 4

The fourth term format is shown below.

        comparator   ::= (value connective value control) |
                            (identifier *<=* value control)
        value   ::= literal | arithmeticexpression
        literal   ::= literaltype "string"
        literaltype   ::= B | O | X | E | A
        string   ::= from 0 to 256 characters
        connective   ::= .LE. | .LT. | .GE. | .GT. | .EQ. | .NE.

The fourth term format is used for assignment and comparison.

The assignment operator *<=* assigns the value to the identifier.
The connectives have their usual meaning. Values to be compared must
have the same type and length attributes or an error condition arises
and the form fails.

The Application of a Term

The elements of a term are applied by the following sequence of
steps.

        1. The data type, value expression, and length expression
            together specify a unit value, call it x.

2. The replication expression specifies the number of times x
    is to be repeated. The value of the concatenated xs
    becomes y of length L.

3. If the term is an input stream term then the value of y of
    length L is tested with the input value beginning at the
    current input pointer position.

4. If the input value satisfies the constraints of y over
    length L then the input value of length L becomes the value
    of the term.

In an output stream term, the procedure is the same except that the
source of input is the value of the term(s) named in the value
expression and the data is emitted in the output stream.

The above procedure is modified to include a one term look-ahead
where replicated values are of indefinite length because of the
arbitrary symbol, #.

Restrictions and Interpretations of Term Functions

1. Terms having indefinite lengths because their values are
    repeated according to the # symbol, must be separated by some
    type-specific data such as a literal. (A literal isn't
    specifically required, however. An arbitrary number of ASCII
    characters could be terminated by a non-ASCII character.)

2. Truncation and padding is as follows:
    a) Character to character (A <-> E) conversion is left-
        justified and truncated or padded on the right with blanks.
    b) Character to numeric and numeric to numeric conversions are
        right-justified and truncated or padded on the left with
        zeros.
    c) Numeric to character conversions is right-justified and
        left-padded with blanks.

3. The following are ignored in a form definition over the control
    connection.
    a) TELNET control characters.
    b) Blanks except within quotes.
    c) /* string */ is treated as comments except within quotes.

4. The following defaults prevail where the term part is omitted.

    a) The replication expression defaults to one.
    b) # in an output stream term defaults to one.
    c) The value expression of an input stream term defaults to

                   the value found in the input stream, but the input stream
                   must conform to the data type and length expression. The
                   value expression of an output stream term defaults to
                   padding only.
              e)   The length expression defaults to the size of the quantity
                   determined by the data type and value expression.
              f)   Control defaults to the next sequential term if a term is
                   successfully applied; else control defaults to the next
                   sequential rule. If _where_ evaluates to an undefined
                   label the form fails.

     5.   Arithmetic expressions are evaluated left-to-right with no
            precedence.

     6.   The following limits prevail.

              a)   Binary lengths are <= 32 bits
              b)   Character strings are <= 256 8-bit characters
              c)   Identifier names are <= 4 characters
              d)   Maximum number of identifiers is <= 256
              e)   Label integers are >= 0 and <= 9999
     7.   Value and length operators product 32-bit binary integers. The
            value operator is currently intended for converting A or E type
            decimal character strings to their binary correspondents. For
            example, the value of E'12' would be 0......01100. The value
            of E'AB' would cause the form to fail.

TERM AND RULE SEQUENCING

   Sequencing may be explicitly controlled by including control in a
   term.

```
        control ::=  :options | <null>
        options ::=  S(where) | F(where) | U(where)
                     S(where) , F(where) |
                     F(where) , S(where)

        where   ::=  arithmeticexpression | R(arithmeticexpression)
```

   S, F, and U denote success, fail, and unconditional transfers,
   respectively. _Where_ evaluates to a _rule_ label, thus transfer can
   be effected from within a rule (at the end of a term) to the
   beginning of another rule. R means terminate the form and return the
   evaluated expression to the initiator over the control connection (if
   still open).

   If terms are not explicitly sequenced, the following defaults
   prevail.

1) When a term fails go to the next sequential rule.
2) When a term succeeds go to the next sequential
    term within the rule.
3) At the end of a rule, go to the next sequential
    rule.

Note in the following example, the correlation between transfer of
control and movement of the input pointer.

```
1    XYZ(,B,,8:S(2),F(3)) : XYZ ;
2    . . . . . . .
3    . . . . . . .
```

The value of XYZ will never be emitted in the output stream since
control is transferred out of the rule upon either success or
failure. If the term succeeds, the 8 bits of input will be assigned
as the value of XYZ and rule 2 will then be applied to the same input
stream data. That is, since the complete left hand side of rule 1
was not successfully applied, the input stream pointer is not
advanced.

IV. EXAMPLES

REMARKS

The following examples (forms and also single rules) are simple
representative uses of the Form Machine. The examples are expressed
in a term-per-line format only to aid the explanation. Typically, a
single rule might be written as a single line.

FIELD INSERTION

To insert a field, separate the input into the two terms to allow the
inserted field between them. For example, to do line numbering for a
121 character/line printer with a leading carriage control character,
use the following form.

```
(NUMB*<=*1);          /*initialize line number counter to one*/
1 CC(,E,,1:F(R(99))),  /*pick up control character and save
                          as CC*/
                          /*return a code of 99 upon exhaustion*/
LINE(,E,,121 : F(R(98)))  /*save text as LINE*/
:CC,                  /*emit control character*/
(,E,NUMB,2),          /*emit counter in first two columns*/
(,E,E".",1),          /*emit period after line number*/
(,E,LINE,117),        /*emit text, truncated in 117 byte field*/
(NUMB*<=*NUMB+1:U(1));   /*increment line counter and go to
                            rule one*/;;
```

DELETION

   Data to be deleted should be isolated as separate terms on the left,
   so they may be omitted (by not emitting them) on the right.

   (,B,,8),   /*isolate 8 bits to ignore*/
   SAVE(,A,,10)   /*extract 10 ASCII characters from
                           input stream*/
   :(,E,SAVE,);   /*emit the characters in SAVE as EBCDIC
                           characters whose length defaults to the
                           length of SAVE, i.e., 10, and advance to
                           the next rule*/

   In the above example, if either input stream term fails,
   the next sequential rule is applied.

VARIABLE LENGTH RECORDS

   Some devices, terminals and programs generate variable
   length records. The following rule picks up variable length
   EBCDIC records and translates them to ASCII.

   CHAR(#,E,,1),   /*pick up all (an arbitrary number of)
                           EBCDIC characters in the input stream*/
   (,X,X"FF",2)   /*followed by a hexadecimal literal,
                           FF (terminal signal)*/
   :(,A,CHAR,),   /*emit them as ASCII*/
   (,X,X"25",2);   /*emit an ASCII carriage return*/

STRING LENGTH COMPUTATION

   It is often necessary to prefix a length field to an arbitrarily long
   character string. The following rule prefixes an EBCDIC string with
   a one-byte length field.

   Q(#,E,,1),   /*pick up all EBCDIC characters*/
   TS(,X,X"FF",2)   /*followed by a hexadecimal literal, FF*/
   :(,B,L(Q)+2,8),   /*emit the length of the characters
                           plus the length of the literal plus
                           the length of the count field itself,
                           in an 8-bit field*/
   Q,   /*emit the characters*/
   TS,   /*emit the terminal*/

TRANSPOSITION

   It is often desirable to reorder fields, such as the following
   example.

   Q(,E,,20), R(,E,,10) , S(,E,,15), T(,E,,5) : R, T, S, Q ;

   The terms are emitted in a different order.

CHARACTER PACKING AND UNPACKING

   In systems such as HASP, repeated sequences of characters are packed
   into a count followed by the character, for more efficient storage
   and transmission. The first form packs multiple characters and the
   second unpacks them.

```
   /*form to pack EBCDIC streams*/
   /*returns 99 if OK, input exhausted*/
   /*returns 98 if illegal EBCDIC*/
   /*look for terminal signal FF which is not a legal EBCDIC*/
   /*duplication count must be 0-254*/
   1 (,X,X"FF",2 : S(R(99))) ;
   /*pick up an EBCDIC char/*
   CHAR(,E,,1) ;
   /*get identical EBCDIC chars/*
   LEN(#,E,CHAR,1)
   /*emit the count and the char/*
   : (,B,L(LEN)+1,8), CHAR, (:U(1));
   /*end of form*/;;

   /*form to unpack EBCDIC streams*/
   /*look for terminal*/
   1 (,X,X"FF",2 : S(R(99))) ;
   /*emit character the number of times indicated*/
   /*by the count, in a field the length indicated*/
   /*by the counter contents*/
   CNT(,B,,8), CHAR(,E,,1) : (CNT,E,CHAR,1:U(1));
   /*failure of form*/
   (:U(R(98))) ;;
```