LANGUAGE-MACHINE FOR DATA RECONFIGURATION

Introduction

    In NWG/RFC #80 we mentioned the needs for data reconfiguration along
    with a complier/executor version of a Form Machine to perform those
    manipulations.

    This note proposes a different approach to the Form Machine.
    Specifically, we describe a syntax-driven interpreter that operates
    on a grammar which is an ordered set of replacement rules. Following
    the interpreter description are some "real-world" examples of
    required data reconfigurations that must occur between RAND consoles
    and the Remote Job System on the UCLA 360/91. Lastly, we suggest
    that the Protocol Manager mentioned in NWG/RFC #80 can be simplified
    by using the Form Machine and two system forms (specified a priori in
    the code).

    Caveat: The Form Machine is not intended to be a general purpose
    programming language. Note the absence of declaration statements,
    etc.

THE FORM MACHINE

I. Forms

    A form is an ordered set of rules.

        F = {R1, ...,Rn}

    The first rule (R1) is the rule of highest priority; the last rule
    (Rn) is the rule of lowest priority.

    The form machine gets as input: 1) a list of addresses and lengths
    that delimit the input stream(s); 2) a list of addresses and lengths
    that delimit the output area(s); 3) a pointer to a list of form(s);
    4) a pointer to the starting position of the input stream; and 5) a
    pointer to the starting position of the output area. The Form
    Machine applies a form to the input string emitting an output string
    in the output area. The form is applied in the following manner:

        Step 1: R1 is made the current rule.

        Step 2: The current rule is applied to the input data.

        Step3:   a) If the rule fails, the rule of priority one lower is
                    made current.

                 b) If the rule succeeds, the rule of highest priority is
                    made current

                 c) When the rule of lowest priority fails, the form fails
                    and application of the form to the input data
                    terminates.

        Step 4: Continue at Step 2.

   In addition, during Step 2, if the remainder of the input string is
   insufficient to satisfy a rule, then that rule fails and partial
   results are not emitted. If a rule fills the output string,
   application of the form is terminated.

II. Rules

   A rule is a replacement operation of the form:

      left-hand-side -> right-hand-side

   Both sides of a rule consists of a series of zero or more _terms_
   (see below) separated by commas.

   The left-hand-side of the rule is applied to the input string at the
   current position as a pattern-match operation. If it exactly
   describes the input, 1) the current input position pointer is
   advanced over the matched input, 2) the right-hand-side emits data at
   the current position in the output string, and 3) the current output
   position pointer is advanced over the emitted data.

III. Terms

   A term is a variable that describes the input string to be matched or
   the output string to be emitted. A term has three formats.

Term Format 1
```
+-----------------------------------------------------------------+
|                                                                 |
|    name ( data   replication   .   value   :    length    )     |
|          type    expression    expression      expression       |
|                                                                 |
|_____|
```

   Any of the fields may be absent.

   The _name_ is a symbolic name of the term in the usual programming
   language sense. It is a single, lower-case alphabetic that is unique
   within a rule.

   The _data type_ describes the kind of data that the term represents.
   It is a member of the set:

        {D, O, X, A, E, B}

   Data types have the following meanings and implied unit lengths:

Char.	Meaning	Length
D	decimal number	1 bit
O	octal number	3 bits
X	hexadecimal number	4 bits
A	ASCII character	8 bits
E	EBCDIC character	8 bits
B	binary number	1 bit

   The _replication expression_ is a multiplier of the value expression.
   A replication expression has the formats.

   1) an arithmetic expression of the members of the set:

        {v(name), L(name) , numerals, programming variables}

   The v(name) is a value operator that generates a numeric value of
   the named data type and L(name) is a length operator that
   generates a numeric value of the named string length.

   The programming variable is described under term format three.
   Arithmetic operators are shown below and have their usual
   meanings.

        {*, /, +, -}

or 2) the terminal '#' which means an arbitrary multiple of the value
     expression.

The _value expression_ is the unit value of a term expressed in the
format indicated by the data type. The value expression is repeated
according to the replication expression. A value expression has the
format:

   1) same as part 1) of the replication expression where again
      v(name) produces a numeric value

or 2) a single member of the set

      {v(name), quoted literal}

      where v(name) produces a data type (E or A) value). (Note that
      concatenation is accomplished through multiple terms.)

The _length expression_ is the length of the field containing the
value expression as modified by the replication expression. It has
the same formats as a replication expression.

Thus, the term

   x(E(7.'F'):L(x)) is named x, is of type EBCDIC, has the value
   'FFFFFFF' and is of length 7.

The term

   y(A:8) on the left-hand-side of a rule would be assigned the next
   64 bits of input as its value; on the right-hand-side it would
   only cause the output pointer to be advanced 64 bit positions
   because is has no value expression (contents) to generate data in
   the output area.

Term Format 2
```
+-------------------------------------------------------------------+
|                                                                   |
|              name (label)                                         |
|                                                                   |
+-------------------------------------------------------------------+
```

    The _label_ is a symbolic reference to a previously named term in the
    rule. It has the same value as the term by that name.

    The identity operation below illustrates the use of the _label_
    notation.

        a(A:10) -> (a)

    The (a) on the right-hand side causes the term a to be emitted in the
    output area. It is equivalent to the rule below.

        a(A:10) -> (Av(a):L(a))

Term Format 3
```
+-------------------------------------------------------------------+
|                                                                   |
|    name      ( programming     connective        operand  )       |
|                variable                          expression       |
|                                                                   |
+-------------------------------------------------------------------+
```

    A _programming variable_ is a user-controlled data item that does not
    explicitly appear in the input/output streams. Its value can be
    compared to input data, to constants, and used to generate output
    data. Programming variables are single, lower case Greek symbols.

    They are used: to generate indices, counters, etc. in the output
    area; to compare indices, counters, etc. in the input area, and; to
    bind replacement rules where the data is context sensitive (explained
    later).

    A _connective_ is a member of the set:

        {<-, =, !=, >=, <=, <, >}

    The left arrow denotes replacement of the left part by the right
    part; the other connectives are comparators.

The _operand expression_ is an arithmetic expression of members of
the set:

        {programming variables, v(name), l(name), numerals}

For example, if the programming variable [alpha] has the value 0 and
the rule

    a(H[alpha]:1) -> (a), ([alpha]<-[alpha]+1), (H[alpha]:1)

is applied exhaustively to string of hexadecimal digits

    0 1 2 3 4 5

the output would be the hexadecimal string

    0 1 1 2 2 3 3 4 4 5 5 6 .

Note: the above rule is equivalent to

    a(B[alpha]:4) -> (a), ([alpha]<-[alpha]+1), (B[alpha]:4)

IV. Restrictions and Interpretations of Term Functions

When a rule succeeds output will be generated. In the rule

    a(A:#),(A'/':1)->(Ev(a):74),(E'?':1)

the input string is searched for an arbitrary number of ASCIIs
followed by a terminal '/'. The ASCIIs (a) are converted to EBCDIC
in a 74-byte field followed by a terminal '?'. This brings out three
issues:

    1. Arbitrary length terms must be separated by literals since the
       data is not type-specific.

    2. The # may only be used on the left-hand-side of a rule.

    3. A truncation padding scheme is needed.

The truncation padding scheme is as follows:

   a. Character to Character (types: A, E)

      Output is left-justified with truncation or padding (with
      blanks) on the right.

   b. Character to Numeric (A, E to D, O, H, B)

   c. Numeric to Character (D, O, H, B to A, E)

   d. Numeric to Numeric (D, O, H, B)

      Output is right-justified with padding or truncation on the
      left. Padding is zeros if output is numeric.

EXAMPLES OF SOME DATA RECONFIGURATIONS

   The following are examples of replacement rule types for specifically
   needed applications.

   Literal Insertion

      To insert a literal, separate the left-hand-side terms for its
      insertion on the right.

         a(A:10),b(A:70)->(a),(E'LIT':3),(b)

      The 80 ASCII characters are emitted in the output area with the
      EBCDIC literal LIT inserted after the first 10 ASCII characters.

   Deletion

      Terms on the left are separated so that the right side may omit
      unwanted terms.

         (B:7),a(A:10)->(Ev(a):L(a))

      Only the 10 ASCII characters are emitted (as EBCDIC) in the output
      area, the 7 binary digits are discarded.

   Spacing in the Output Buffer

      Where a pre-formatted output buffer exists (typically a display
      buffer) spacing can be realized by omitting the replication and
      value functions from a term on the right.

```
     a(A:74)->(E:6),(Ev(a):74)
```

   The (E:6) causes 48 bit positions to be skipped over in the output
   area, then the 74 ASCII characters are converted to EBCDIC and
   emitted at the current output position.

   Arbitrary Lengths

      Some devices/programs generate a variable number of characters per
      line and it is desirable to produce fixed-length records from
      them.

```
     a(A:#) -> (Ev(a):74)
```

      The ASCII characters are truncated or padded as required and
      converted to EBCDIC in a 74 character field.

   Transposition

      Fields to be transposed should be isolated as terms on the left.

```
     a(X:2),b(A:#)->(Ev(b):L(b)),(a)
```

   String Length Computation

      Some formats require the string length as part of the data stream.
      This can be accomplished by the length function.

```
     a(E:10),b(X'FF':2)->(BL(a)+L(b)+8:8),(Av(a):L(a)),(b)
```

      The length term is emitted first, in a 8 bit field. In this case
      the length includes the length field as well as the ASCII
      character field.

   Expansion and Compression of repeated Symbols

      The following rule packs repeated symbols.

```
     a(E:1), b(E#*v(a):L(b)) -> (BL(b)+1:8),(a)
```

      Given the input string below, three successive applications of the
      rule will emit the output string shown.

         Input: XXXXYYZZZZZZZ

         Output: 4X2Y7Z

APPLICATION OF THE FORM MACHINE TO PROGRAM PROTOCOLS

The Protocol Manager mentioned in NWG/RFC #80 needs several
interesting features that are properties of the above Form Machine.

In certain instances during a protocol dialog it might be acceptable
to get either an accept on connection A or an allocation on connect
B, that is, the order is sometimes unimportant. The defined
procedure for applying rules allows for order independence.

A logger might send us a socket number embedded in a regular message
-- the socket number is intended to be the first of a contiguous set
of sockets that we can use to establish connections with some
program. We wish to extract the socket number field from the regular
message, perhaps convert it to another format, and add to it to get
the additional socket names. As a result of the regular message we
wish to emit several INIT system calls that include the socket
numbers that we have computed. The value operator and the arithmetic
operators of the Form Machine can do this.

A third property of the Form Machine that is applicable to protocols
is inter- and intra-rule binding to resolve context sensitive
information. In general we wish rules to be order independent but in
certain cases we wish to impose an ordering. Using the logger in
NWG/RFC #66 as an example, the close that is sent by the logger can
have two different meanings depending upon its context. If the close
is sent before the regular message containing the socket number then
it means call refused. If the regular message precedes the close
then the call is accepted. Since the close has contextual meaning,
we must bind it to the regular message to avoid introducing IF and
THEN into the Form Machine language.

Assume for a moment that we can express system calls in Form Machine
notation. (The notation below is for _illustration only_ and is not
part of the Form Machine language.) We have two ways to bind the
regular message to the close. By intra-rule binding we insist that
the close be preceded by a regular message.

    Reg. Msg , Close ->

Now assume for a moment that the remote party must have an echo after
each transmission. Since we must emit an echo after receiving the
regular message and before the close is sent, then we must use
inter-rule binding. This can be accomplished with the programming
variable. It is assigned a value when the regular message is
received and the value is tested when the close is received.

    Reg. Msg -> Echo , ([lambda]+1)

       Close, ([lambda]=1) ->

    To illustrate inter-rule binding via the programming variable the
    connection protocol in NWG/RFC #66 could be represented by passing
    the following form to a protocol manager. (The notation below is for
    illustration only and is not part of the Form Machine language).

       1. ->INIT(parameters) , ([alpha]<-0)

    Send an INIT(RTS).

       2. INIT(parameters) -> ALLOCATE(parameters)

    Send an allocate in response to the connection completion (an STR
    received).

       3. Reg. Msg (parameters) -> ([alpha]<-1)

    When the messages bearing link numbers is received, set an
    internal indicator. (The extraction of the link is not
    illustrated.)

       4. CLOSE(parameters),([alpha]=1) ->
                            INIT(parameters),INIT(parameters)

    When the close is received following the regular message [2] is
    checked to see that the regular message was received before
    establishing the duplex connection. If the close is received with
    no regular message preceding it (call refused) the form will fail
    (since no rules is satisfied).

    This protocol can be handled via a single form containing four
    replacement rules. We have examined similar representations for more
    complex protocol sequences. Such protocol sequences, stored by name,
    are an asset to the user; he can request a predefined sequence to be
    executed automatically.

Two System Forms to Handle Protocol Statements

   Assume that we have a Protocol Manager that manages protocol
   sequences between consoles and the Network. The consoles generate
   and accept EBCDIC character strings and the Network transmits binary
   digits. The console user has a language similar to system calls in
   which he can create and store protocol sequences via Protocol
   Manager, and at the same time he can indicate which commands are
   expected to be sent and which are to be received. Upon command the
   Protocol Manager can execute this sequence with the Network,
   generating commands and validating those received. Assume also that
   the Protocol Manager displays the dialog for the console user as it
   progresses.

   In order to translate between console and Network for generating,
   comparing, and displaying commands, the Protocol Manager can use the
   Form Machine. Two system forms are needed, see Fig. 1. One is a
   console-to-Network set of rules containing EBCDIC to binary for all
   legal commands; the other is a mirror image for Network-to-console.

REQUEST

   Since language design is not our forte, we would like comments from
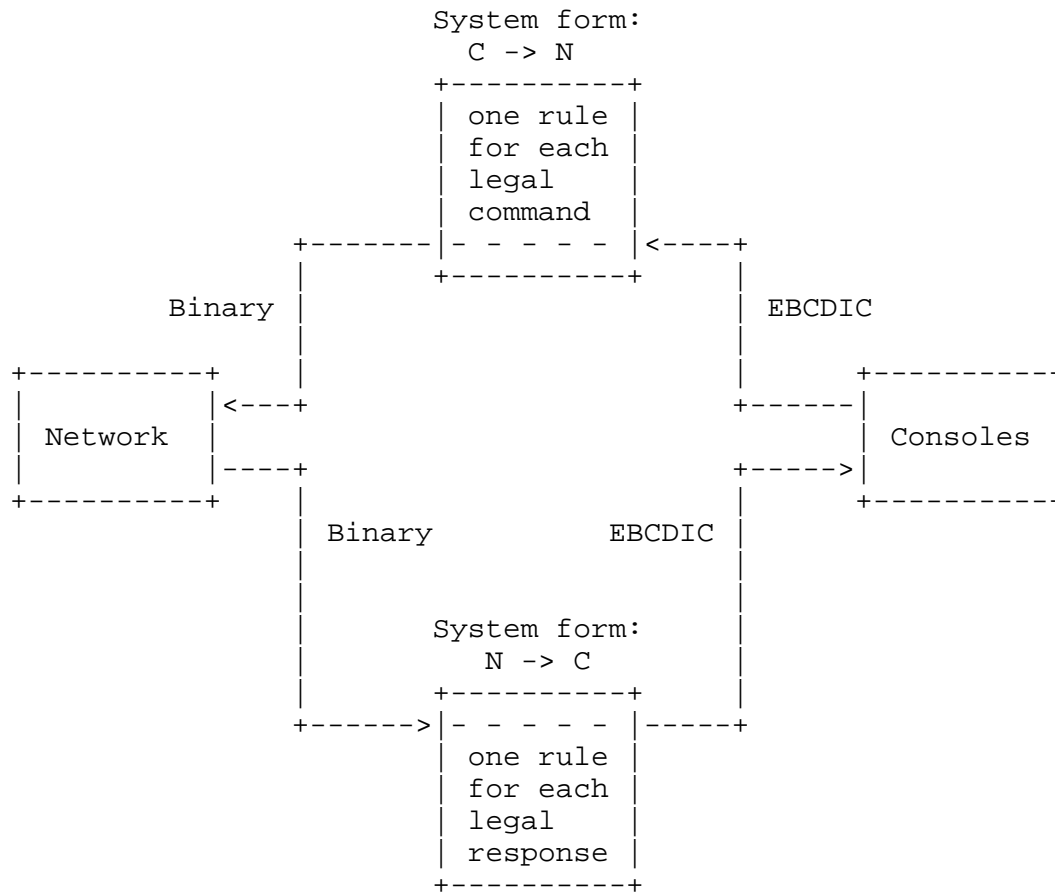   those with more experience than we.

```
                         System form:
                           C -> N
                         +----------+
                         | one rule |
                         | for each |
                         | legal    |
                         | command  |
                +-------|- - - - - |<----+
                |        +----------+      |
          Binary |                         | EBCDIC
                |                         |
  +----------+  |                         |     +----------+
  |          |<---+                  +------|     |          |
  | Network  |                         | Consoles |
  |          |----+                  +----->|     |          |
  +----------+  |                         |     +----------+
                | Binary          EBCDIC |
                |                         |
                |                         |
                |   System form:          |
                |      N -> C             |
                |   +----------+          |
                +------>|- - - - - |-----+
                    | one rule |
                    | for each |
                    | legal    |
                    | response |
                    +----------+
```

Figure 1 -- Application of System Form for Protocol Management

Distribution List

     Alfred Cocanower - MERIT
     Gerry Cole - SDC
     Les Earnest - Stanford
     Bill English - SRI
     James Forgie - Lincoln Laboratory
     Jennings Computer Center - Case
     Nico Haberman - Carnegie-Melon
     Robert Kahn - BB&N
     Peggy Karp - MITRE
     Benita Kirstel - UCLA
     Tom Lawrence - RADC/ISIM
     James Madden - University of Illinois
     George Mealy - Harvard
     Thomas O'Sullivan - Raytheon
     Larry Roberts - ARPA
     Ron Stoughton - UCSB
     Albert Vezza- MIT
     Barry Wessler - Utah

     [The original document included non-ASCII characters. The Greek
     letters Alpha and Lambda have been spelled out and enclosed in
     square brackets "[ ]". A curly "l" character
     has been replaced by capital L. Left and right arrows have been
     replaced by "<-" and "->" respectively. RFC-Editor]

          [This RFC was put into machine readable form for entry]
          [into the online RFC archives by Lorrie Shiota, 10/01]