Network Working Group
RFC-5

4691
Jeff Rulifson
June 2, l969

DEL

:DEL, 02/06/69 1010:58 JFR ; .DSN=1; .LSP=0; ['=] AND NOT SP ; ['?]; dual transmission?

ABSTRACT

The Decode-Encode Language (DEL) is a machine independent language tailored to two specific computer network tasks:

accepting input codes from interactive consoles, giving immediate feedback, and packing the resulting information into message packets for network transmissin.

and accepting message packets from another computer, unpacking them, building trees of display information, and sending other information to the user at his interactive station.

This is a working document for the evolution of the DEL language. Comments should be made through Jeff Rulifson at SRI.

FORWARD

The initial ARPA network working group met at SRI on October 25-26, 1968.

It was generally agreed beforehand that the runmning of interactive programs across the network was the first problem that would be faced.

This group, already in agreement about the underlaying notions of a DEL-like approach, set down some terminology, expectations for DEL programs, and lists of proposed semantic capability.

At the meeting were Andrews, Baray, Carr, Crocker, Rulifson, and Stoughton.

A second round of meetings was then held in a piecemeal way.

Crocker meet with Rulifson at SRI on November 18, 1968. This resulted in the incorporation of formal co-routines.

and Stoughton meet with Rulifson at SRI on Decembeer 12, 1968. It was decided to meet again, as a group, probably at UTAH, in late January 1969.

The first public release of this paper was at the BBN NET meeting in Cambridge on February 13, 1969.

NET STANDARD TRANSLATORS

NST The NST library is the set of programs necessary to mesh efficiently with the code compiled at the user sites from the DEL programs it receives. The NST-DEL approach to NET interactive system communication is intended to operate over a broad spectrum.

The lowest level of NST-DEL usage is direct transmission to the server-host, information in the same format that user programs would receive at the user-host.

In this mode, the NST defaults to inaction. The DEL program does not receive universal hardware representation input but input in the normal fashion for the user-host.

And the DEL 1 program becomes merely a message builder and sender.

A more intermediate use of NST-DEL is to have echo tables for a TTY at the user-host.

In this mode, the DEL program would run a full duplex TTY for the user.

It would echo characters, translate them to the character set of the server-host, pack the translated characters in messages, and on appropriate break characters send the messages.

When messages come from the server-host, the DEL program would translate them to the user-host character set and print them on his TTY.

A more ambitious task for DEL is the operation of large, display-oriented systems from remote consoles over the NET.

Large interactive systems usually offer a lot of feedback to the user. The unusual nature of the feedback make it impossible to model with echo table, and thus a user program must be activated in a TSS each time a button state is changed.

This puts an unnecessarily large load on a TSS, and if the system is being run through the NET it could easily load two systems.

To avoid this double overloading of TSS, a DEL program will run on the user-host. It will handle all the immediate feedback, much like a complicated echo table. At appropriate button pushes, message will be sent to the server-host and display updates received in return.

One of the more difficult, and often neglected, problems is the effective simulation of one nonstandard console on another non-standard console.

We attempt to offer a means of solving this problem through the co-routine structure of DEL programs. For the complicated interactive systems, part of the DEL programs will be constructed by the server-host programmers. Interfaces between this program and the input stream may easily be inserted by programmers at the user-host site.

UNIVERSAL HARDWARE REPRESENTATION

To minimize the number of translators needed to map any facility’s user codes to any other facility, there is a universal hardware representation.

This is simply a way of talking, in general terms, about all the

hardware devices at all the interactive display stations in the initial network.

For example, a display is thought of as being a square, the mid-point has coordinates (0.0), the range is -1 to 1 on both axes. A point may now be specified to any accuracy, regardless of the particular number of density of rastor points on a display.

The representation is discussed in the semantic explanations accompanying the formal description of DEL.

INTRODUCTION TO THE NETWORK STANDARD TRANSLATOR (NST)

Suppose that a user at a remote site, say Utah, is entered in the AHI system and wants to run NLS.

The first step is to enter NLS in the normal way. At that time the Utah system will request a symbolic program from NLS.

REP This program is written in DEL. It is called the NLS Remote Encode Program (REP).

The program accepts input in the Universal Hardware Representation and translates it to a form usable by NLS.

It may pack characters in a buffer, also do some local feedback.

When the program is first received at Utah it is compiled and loaded to be run in conjunction with a standard library.

All input from the Utah console first goes to the NLS NEP. It is processed, parsed, blocked, translated, etc. When NEP receives a character appropriate to its state it may finally initiate transfers to the 940. The bits transferred are in a form acceptable to the 940, and maybe in a standard form so that the NLSW need not differentiate between Utah and other NET users.

ADVANTAGES OF NST

After each node has implemented the library part of the NST, it need only write one program for each subsystem, namely the symbolic file it sends to each user that maps the NET hardware representation into its own special bit formats.

This is the minimum programming that can be expected if console is used to its fullest extent.

Since the NST which runs the encode translation is coded at the user site, it can take advantage of hardware at its consoles to the fullest extent. It can also add or remove hardware features without requiring new or different translation tables from the host.

Local users are also kept up to date on any changes in the system offered at the host site. As new features are added, the host programmers change the symbolic encode program. When this new program is compiled and used at the user site, the new features are automatically included.

The advantages of having the encode translation programs transferred symbolically should be obvious.

Each site can translate any way it sees fit. Thus machine code for each site can be produced to fit that site; faster run times and greater code density will be the result.

Moreover, extra symbolic programs, coded at the user site, may be easily interfaced between the user's monitor system and the DEL program from the host machine. This should ease the problem of console extension (e.g. accommodating unusual keys and buttons) without loss of the flexibility needed for man-machine interaction.

It is expected that when there is matching hardware, the symbolic programs will take this into account and avoid any unnecessary computing. This is immediately possible through the code translation constructs of DEL. It may someday be possible through program composition (when Crocker tells us how??)

AHI NLS - USER CONSOLE COMMUNICATION - AN EXAMPLE

BLOCK DIAGRAM

The right side of the picture represents functions done at the user's main computer; the left side represents those done at the host computer.

Each label in the picture corresponds to a statement with the same name.

There are four trails associated with this picture. The first links (in a forward direction) the labels which are concerned only with network information. The second links the total information flow (again in a forward direction). The last two are equivalent to the first two but in a backward direction. They may be set with pointers t1 through t4 respectively.

[">tif:] OR I" >nif"]; ["<tif:] OR ["<nif"];

USER-TO-HOST TRANSMISSION

Keyboard is the set of input devices at the user's console. Input bits from stations, after drifting through levels of monitor and interrupt handlers, eventually come to the encode translator. [>nif(encode)]

Encode maps the semi-raw input bits into an input stream in a form suited to the serving-host subsystem which will process the input. [>nif(hrt)<nif(keyboard)]

The Encode program was supplied by the server-host subsystem when the subsystem was first requested. It is sent to the user machine in symbolic form and is compiled at the user machine into code particularly suited to that machine.

It may pack to break characters, map multiple characters to single characters and vice versa, do character translation, and give immediate feedback to the user.

1 dm Immediate feedback from the encode translator first goes to local display management, where it is mapped from the NET standard to the local display hardware.

A wide range of echo output may come from the encode translator. Simple character echoes would be a minimum, while command and machine-state feedback will be common.

It is reasonable to expect control and feedback functions not even done at the server-host user stations to be done in local display control. For example, people with high-speed displays may want to selectively clear curves on a Culler display, a function which is impossible on a storage tube.

Output from the encode translator for the server-host goes to the invisible IMP, is broken into appropriate sizes and labeled by the encode translator, and then goes to the NET-to-host translator.

Output from the user may be more than on-line input. It may be larger items such as computer-generated data, or files generated and used exclusively at the server-host site but stored at the user-host site.

Information of this kind may avoid translation, if it is already in server-host format, or it may undergo yet another kind of translation if it is a block of data.

hrp It finally gets to the host, and must then go through the host reception program. This maps and reorders the standard transmission-style packets of bits sent by the encode programs into messages acceptable to the host. This program may well be part of the monitor of the host machine. [>tif(net mode)<nif(code)]

HOST-TO-USER TRANSMISSION

decode Output from the server-host initially goes through decode, a translation map similar to, and perhaps more complicated than, the encode map. [>nif(urt)>tif(imp ctrl)<tif(net mode)]

This map at least formats display output into a simplified logical-entity output stream, of which meaningful pieces may be dealt with in various ways at the user site.

The Decode program was sent to the host machine at the same time that the Encode program was sent to the user machine. The program is initially in symbolic form and is compiled for efficient running at the host machine.

Lines of charaters should be logically identified so that different line widths can be handled at the user site.

Some form of logical line identification must also be made. For example, if a straight line is to be drawn across the display this fact should be transmitted, rather than a series of 500 short vectors.

As things firm up, more and more complicated structural display information (in the manner of LEAP) should be sent and accommodated at user sites so that the responsibility for real-time display manipulation may shift closer to the user.

imp ctrl The server-host may also want to send control information to IMPs. Formatting of this information is done by the host decoder. [>tif(urt) <tif(decode)]

The other control information supplied by the host decoder is message break up and identification so that proper assembly and sorting can be done at the user site.

From the host decoder, information does to the invisible IMP, and directly to the NET-to-user translator. The only operation done on the messages is that they may be shuffled.

urt The user reception translator accepts messages from the user-site IMP 1 and fixes them up for user-site display. [>nif(d ctrl)>tif(prgm ctrl)<tif(imp ctrl)<nif(decode)]

The minimal action is a reordering of the message pieces.

dctrl For display output, however, more needs to be done. The NET logical display information must be put in the format of the user site. Display control does this job. Since it coordinates between (encode) and (decode) it is able to offer features of display management local to the user site. [>nif(display)<nif(urt)]

prgmctrl Another action may be the selective translation and routing of information to particular user-site subsystems. [>tif(dctrl)<tif(urt)]

For example, blocks of floating-point information may be converted to user-style words and sent, in block form, to a subsystem for processing or storage.

The styles and translation of this information may well be a compact binary format suitable for quick translation, rather than a print-image-oriented format.

(display) is the output to the user. [<nif(d ctrl)]

USER-TO-HOST INDIRECT TRANSMISSION

(net mode) This is the mode where a remote user can link to a node indirectly through another node. [<nif(decode)<tif(hrt)]

DEL SYNTAX

NOTES FOR NLS USERS

All statements in this branch which are not part of the compiler must end with a period.

To compile the DEL compiler:

Set this pattern for the content analyzer ((symbol for up arrow)P1 SE(P1) <-"-;). The pointer "del" is on the first character of pattern.

Jump to the first statement of the compiler. The pointer "c" is on this statement.

And output the compiler to file (’/A-DEL’). The pointer "f" is on the name of the file for the compiler output -

PROGRAMS

SYNTAX

```
      -meta file (k=100.m=300,n=20,s=900)

      file = mesdecl $declaration $procedure "FINISH";

      procedure =

        procname (

           (

              type "FUNCTION" /

              "PROCEDURE" ) .id (type .id / -empty)) /

           "CO-ROUTINE") ' /

        $declaration labeledst $(labeledst ';) "endp.";

      labeledst = ((left arrow symbol).id ': / .empty) statement;

      type = "INTEGER" / "REAL" ;

      procname = .id;
```

Functions are differentiated from procedures to aid compilers in better code production and run time checks.

Functions return values.

Procedures do not return values.

Co-routines do not have names or arguments. Their initial envocation points are given the pipe declaration.

It is not clear just how global declarations are to be??

DECLARATIONS

SYNTAX

```
      declaration = numbertype / structuredtype / label / lcl2uhr /
      uhr2rmt / pipetype;

      numbertype = : ("REAL" / "INTEGER") ("CONSTANT" conlist /
      varlist);

      conlist =

         .id '(left arrow symbol)constant

         $('. .id '(left arrow symbol)constant);

      varlist =

         .id ('(left arrow symbol)constant / .empty)

         $('. .id('(left arrow symbol)constant / .empty));

      idlist = .id $('. .id);

      structuredtype = (tree" / "pointer" / "buffer" ) idlist;
```

```
      label = "LABEL1" idlist;

      pipetype = PIPE" pairedids $(', pairedids);

      pairedids = .id .id;

      procname = .id;

      integerv = .id;

      pipename = .id;

      labelv = .id;
```

Variables which are declared to be constant, may be put in read-only memory at run time.

The label declaration is to declare cells which may contain the machine addresses of labels in the program as their values. This is not the B5500 label declaration.

In the pipe declaration the first .ID of each pair is the name of the pipe, the second is thke initial starting point for the pipe.

ARITHMETIC

SYNTAX

```
      exp = "IF" conjunct "THEN" exp "ELSE" exp;

      sum = term (

         '+ sum /

         '- sum /

         -empty);

      term = factor (

         '* term /

         '/ term /

         '(up arrow symbol) term /

         .empty);

      factor = '- factor / bitop;

      bitop = compliment (

         '/' bitop /

         '/'\ bitop /

         '& bitop / (

         .empty);

      compliment = "--" primary / primary;
```

(symbol for up arrow) means mod. and /\ means exclusive or.

Notice that the uniary minus is allowable, and parsed so you can write x*-y.

Since there is no standard convention with bitwise operators, they all have the same precedence, and parentheses must be used for grouping.

Compliment is the l’s compliment.

It is assumed that all arithmetic and bit operations take place in the mode and style of the machine running the code. Anyone who takes advantage of word lengths, two’s compliment arithmetic, etc. will eventually have problems.

PRIMARY

SYNTAX

```
primary =

   constant /

   builtin /

   variable / (

   block /

   ’( exp ’);

variable = .id (

   ’(symbol for left arrow) exp /

   ’( block ’) /

   .empty);

constant =  integer / real / string;

builtin =

   mesinfo /

   cortnin /

   ("MIN" / "MAX") exp $(’. exp) ’/ ;
```

parenthesized expressions may be a series of expressions. The value of a series is the value of the last one executed at run time.

Subroutines may have one call by name argument.

Expressions may be mixed. Strings are a big problem? Rulifson also wants to get rid of real numbers!!

CONJUNCTIVE EXPRESSION

SYNTAX

```
conjunct = disjunct ("AND" conjunct / .empty);
```

```
disjunct = negation ("OR" negation / .empty);

negation = "NOT" relation / relation;

relation =

   '( conjunct ') /

   sum (

     "<=" sum /

     ">=" sum /

     '< sum /

     '> sum /

     '= sum /

     '" sum /

     .empty);
```

The conjunct construct is rigged in such a way that a conjunct which is not a sum need not have a value, and may be evaluated using jumps in the code. Reference to the conjunct is made only in places where a logical decision is called for (e.g. if and while statements).

We hope that most compilers will be smart enough to skip unnecessary evaluations at run time. I.e a conjunct in which the left part is false or a disjunct with the left part true need not have the corresponding right part evaluated.

ARITHMETIC EXPRESSION

SYNTAX

```
statement = conditional / unconditional;

unconditional = loopst / cases / cibtrikst / uist / treest /
block / null / exp;

conditional = "IF" conjunct "THEN" unconditional (

   "ELSE" conditional /

   .empty);

block = "begin" exp $('; exp) "end";
```

An expressions may be a statement. In conditional statements the else part is optional while in expressions it is mandatory. This is a side effect of the way the left part of the syntax rules are ordered.

SEMI-TREE MANIPULATION AND TESTING

SYNTAX

```
treest = setpntr / insertpntr / deletepntr;
```

```
      setpntr = "set" "pointer" pntrname "to" pntrexp;

      pntrexp = direction pntrexp / pntrname;

      insertpntr = "insert" pntrexp "as"

         (("left" / "right") "brother") /

         (("first" / "last: ) "daughter") "of" pntrexp;

      direction =

         "up" /

         "down" /

         "forward" /

         "backward: /

         "head" /

         "tail";

      plantree = "replace" pntrname "with" pntrexp;

      deletepntr = "delete: pntrname;

      tree = '( tree1 ') ;

      tree1 = nodename $nodename ;

      nodename = terminal / '( tree1 ');

      terminal = treename / buffername / point ername;

      treename = id;

      treedecl = "pointer" .id / "tree" .id;
```

Extra parentheses in tree building results in linear subcategorization, just as in LISP.

FLOW AND CONTROL

```
   controlst = gost / subst / loopstr / casest;
```

GO TO STATEMENTS

```
      gost = "GO" "TO" (labelv / .id);

         assignlabel = "ASSIGN" .id "TO" labelv;
```

SUBROUTINES

```
      subst = callst / returnst / cortnout;

         callst = "CALL" procname (exp / .emptyu);

         returnst = "RETURN" (exp / .empty);

         cortnout = "STUFF" exp "IN" pipename;
```

```
      cortnin = "FETCH" pipename;

      FETCH is a builtin function whose value is computed by envoking
      the named co-routine.

   LOOP STATEMENTS

      SYNTAX

         loopst = whilest / untilst / forst;

         whilest = "WHILE" conjunct "DO" statement;

         untilst = "UNTIL" conjunct "DO" statement;

         forst = "FOR" integerv '- exp ("BY" exp / .empty) "TO" exp

         "DO" statements;

      The value of while and until statements is defined to be false
      and true (or 0 and non-zero) respectively.

      For statements evaluate their initial exp, by part, and to part
      once, at initialization time.  The running index of for
      statements is not available for change within the loop, it may
      only be read.  If, some compilers can take advantage of this
      (say put it in a register) all the better.  The increment and
      the to bound will both be rounded to integers during the
      initialization.

CASE STATEMENTS

   SYNTAX

      casest = ithcasest / condcasest;

      ithcasest = "ITHCASE" exp "OF" "BEGIN" statement $(';
      statement) "END";

      condcasest = "CASE" exp "OF" "BEGIN" condcs $('; condcs)
      "OTHERWISE" statement "END";


      condcs = conjunct ': statement;

   The value of a case statement is the value of the last case executed.

EXTRA STATEMENTS

   null = "NULL";

I/O STATEMENTS

   iost = messagest / dspyst ;

   MESSAGES

      SYNTAX

         messagest = buildmes / demand;

            buildmest = startmes / appendmes / sendmes;
```

```
            startmes = "start" "message";

            appendmes = "append" "message" "byute" exp;

            sendmes = "send" "message";


         demandmes = "demand" "Message";

      mesinfo =

         "get" "message" "byte"

         "message1" "length" /

         "message" empty: ’?;

      mesdecl = "message" "bytes" "are" ,byn "bits" long" ’..
```

DISPLAY BUFFERS

SYNTAX

```
      dspyst = startbuffer / bufappend / estab;

      startbuffer - "start" "buffer";

      bufappend = "append" bufstuff $(’& bufstuff);

      bufstuff = :

         "parameters" dspyparm $(’. dspyparm) /

         "character" exp /

         "string"1 strilng /

         "vector" ("from" exp ’:exp / .empty) "to" exp ’. exp /

         "position" (onoff / .empty) "beam" "to" exp ’= exp/

         curve" ;

      dspyparm F :

         "intensity" "to" exp /

         "character" "width" "to" exp /

         "blink" onoff /

        "italics" onff;

      onoff = "on" / "off";

      estab = "establish" buffername;
```

LOGICAL SCREEN

The screen is taken to be a square. The coordinates are normalized from -1 to +1 on both axes.

Associated with the screen is a position register, called

PREG. The register is a triple <x.y.r> where x and y specify a point on the screen and r is a rotation in radians, counter clockwise, from the x-axis.

The intensity, called INTENSITY, is a real number in the range from 0 to 1. 0 is black, 1 is as light as your display can go, and numbers in between specify the relative log of the intensity difference.

Character frame size.

Blink bit.

BUFFER BUILDING

The terminal nodes of semi-trees are either semi-tree names or display buffers. A display buffer is a series of logical entities, called bufstuff.

When the buffer is initilized, it is empty. If no parameters are initially appended, those in effect at the end of the display of the last node in the semi-tree will be in effect for the display of this node.

As the buffer is built, the logical entities are added to it. When it is established as a buffername, the buffer is closed, and further appends are prohibited. It is only a buffername has been established that it may be used in a tree building statement.

LOGICAL INPUT DEVICES

Wand

Joy Stick

Keyboard

Buttons

Light Pens

Mice

AUDIO OUTPUT DEVICES

.end

SAMPLE PROGRAMS

Program to run display and keyboard as tty.

```
to run NLS

   input part

   display part

      DEMAND MESSAGE;

      While LENGTH " O DO
```

```
            ITHCASE GETBYTE OF Begin

            ITHCASE GETBYTE OF %file area uipdate% BEGIN

               %literal area%

               %message area%

               %name area%

               %bug%

               %sequence specs%

               %filter specs%

               %format specs%

               %command feedback line%

               %filer area%

               %date time%

               %echo register%

            BEGIN %DEL control%
```

DISTRIBUTION LIST

Steve Carr
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
Phone 801-322-7211 X8224

Steve Crocker

Boelter Hall
University of California
Los Angeles, California
Phone 213-825-4864

Jeff Rulifson

Stanford Research Institute
333 Ravenswood
Menlo Park, California 94035
Phone 415-326-6200 X4116

Ron Stoughton

Computer Research Laboratory
University of California
Santa Barbara, California 93106
Phone 805-961-3221

Mehmet Baray

Corey Hall
University of California

Berkeley, California 94720
Phone 415-843-2621