HGL 19-JAN-73 11:02 14020

Load File Problems?-- See (13782,)

I just read the message about load file problems. The symptoms are to vague to really track it down. Please keep a record for the next time it happens . Did this happen in DNLS or TNLS, what else was done before, etc. I notice, incidentally, that I get similar messages when I mistakenly type improper characters (spaces, etc) in the file name or am connected to the wrong directory. Could this be the problem? Logging out in those cases is not neccessary. Please call if this happens again while it is happening and maybe we can track it down. 1

HGL 19-JAN-73 11:02 14020

Load File Problems?-- See (13782,)

(J14020) 19-JAN-73 11:02; Title: Author(s): Lehtman, Harvey G./HGL;
Distribution: Stone, Duane L., Hopper, J. D., Kaye, Diane S., Lehtman,
Harvey G., Irby, Charles H., Kelley, Kirk E./dls bugs ;
Sub-Collections: SRI-ARC BUGS; Clerk: HGL;

HGL 19-JAN-73 11:08 14021

More on Continue and SNDMSG-- (13904,)

Another note about continue-- (13904,) and my reply. You can get around TENEX by using the GOTO EXEC command in NLS to get to the EXEC to use other subsystems, "Quit" at the EXEC level when finished, and be back in NLS at the correct place. This command may be dangerous if you are not careful (you can have several EXEC forks around and you may not log out of the lower levels) 1

HGL 19-JAN-73 11:08 14021

More on Continue and SNDMSG-- (13904,)

(J14021) 19-JAN-73 11:08; Title: Author(s): Lehtman, Harvey G./HGL;
Distribution: Stone, Duane L., Hopper, J. D., Kaye, Diane S., Lehtman,
Harvey G., Irby, Charles H., Kelley, Kirk E./dls bugs ;
Sub-Collections: SRI-ARC BUGS; Clerk: HGL;

JAKE 19-JAN-73 9:51 14022

Message to Don from Derek

JAKE 19-JAN-73 9:51 14022

Message to Don from Derek

(J14022) 19-JAN-73 9:51; Title: Author(s): Feinler, Elizabeth J. (Jake)/JAKE; Distribution: Wallace, Donald C. (Smokey)/dcw ;
Sub-Collections: SRI-ARC; Clerk: JAKE;

JLM 19-JAN-73 11:52 14023

ADP Technology Committee Agenda

This is the file I tried to send you yesterday--If the good Lord is willing and the creek doesn't rise; it will make it this time.

JLM 19-JAN-73 11:52 14023

ADP Technology Committee Agenda

Tentatively Set for 2nd week in February

AGENDA

ADP Technology Review Committee

1100 -1200	Arrive Griffiss AFB - check into VOQ	5
1200 - 1245	Lunch - Officer's Club	6
Feb 1973		7
1300 - 1315	IS Overview	8
1315 - 1400	Higher Order Languages + (EIS work)	9
1400 - 1430	Associative Processing	10
1430 - 1500	MULTICS + (Security)	11
1500 - 1530	Graphics	12
1545 - 1600	Demonstration DIA Graphics	13
1600 - 1700	Demonstration Displays	14
1800	Cocktails and Dinner	15
Feb 1973		16
0830 - 0915	Data Management System + (JTSA WWDH(MS - etc)	17
0915 - 0930	Augmented Human Intellect (AHI)	18
0930 - 1000	Demonstration of AHI	19
1015 - 1045	OLPARS w Demonstration - time	20
1100 - 1115	Microform Project	21
1115 - 1200	Open discussion	22
1200 - 1330	Lunch	23

24

JLM 19-JAN-73 11:52 14023

ADP Technology Committee Agenda

(J14023) 19-JAN-73 11:52; Title: Author(s): McNamara, John L./JLM;
Distribution: Bethke, William P., Stone, Duane L., Iuorno, Rocco F./WPB
DLS RFI; Sub-Collections: RADC; Clerk: MDP;
Origin: <MCNAMARA>ADPCOMMITTE.NLS;1, 11-JAN-73 8:46 JLM
;.sn=0;.rm=72;

KIRK 19-JAN-73 14:47 14024

Viewspec v just fucked me over again. It DOESN'T work. I was using it on a long list of invisibles replacing tabs with SP's when it messed me up by replacing characters in a visible on the other side of the screen. It worked in other cases previous to that but it seems to be ocasionally messing up randomly. 1

(J14024) 19-JAN-73 14:47; Author(s): Kelley, Kirk E./KIRK;
Distribution: Irby, Charles H., Hopper, J. D., Kaye, Diane S., Lehtman, Harvey G., Irby, Charles H., Kelley, Kirk E./chi bugs ;
Sub-Collections: SRI-ARC BUGS; Clerk: KIRK;

This is a request 1

LMM 18-JAN-73 22:28 14025

(J14025) 18-JAN-73 22:28; Author(s): Masinter, Larry M./LMM;
Distribution: Masinter, Larry M./LMM; Sub-Collections: NIC; Clerk: LMM;

NJN 19-JAN-73 9:02 14026

BBN REport

Susan-- It seems I was mistaken about BBN REport 2491. It is going to be published as the BBN quarterly report, and will not be ready until February. At that time we will be sending you about 150 copies to be distributed to the network. I think that should be sufficient. --Nancy 1

NJN 19-JAN-73 9:02 14026

BBN REport

(J14026) 19-JAN-73 9:02; Title: Author(s): Neigus, Nancy J./NJN; Distribution: Lee, Susan R./SRL; Sub-Collections: NIC; Clerk: NJN;

thank you for your response on time series analysis and forecasting. i am forwarding it to dr. noyer singpurwala at george washington university who will be most appreciative.

i would like to personally thank you for taking the time to resond to my request. such interaction cannot help but make the arpanet a valuable resource, the capabilities of which will be useful to people of a wide range of interests and disciplines.

sincerely,

ernie forman 1

(J14027) 19-JAN-73 13:15; Title: Author(s): Silberski, Robert/RS2; Distribution: Beaman, Kathy/KB; Sub-Collections: NIC; Clerk: RS2;

IMP/TIP Preventive Maintenance Schedule 1

This note presents the current IMP/TIP Preventive Maintenance schedule. Preventive Maintenance is scheduled on a monthly basis, and a given machine is normally scheduled at the same time each month; wherever possible the time is chosen to coincide with the PM time for the Host(s) at the site. 1a

It is sometimes necessary to reschedule PM's because of network connectivity considerations or for other reasons. Whenever such rescheduling occurs, we will announce the fact as soon as possible via two mechanisms - the TIP NEWS and a Journal note (NOT an RFC) to Technical Liaisons. Any permanent changes to the schedule will be announced via the RFC mechanism. 1b

All times shown below are Eastern Time. 1c

First Monday (of each month) 1c1

BBN IMP [0830-1130] 1c1a

First Tuesday 1c2

BBN TIP [0830-1130] 1c2a

UCSB [1130-1430] 1c2b

First Wednesday 1c3

CARNEGIE [0830-1130] 1c3a

SRI [1130-1430] 1c3b

XEROX [1500-1800] 1c3c

First Thursday 1c4

SAAC [0830-1130] 1c4a

UTAH [1030-1330] 1c4b

Second Monday 1c5

MIT [0830-1130] 1c5a

NOAA [1030-1330] 1c5b

IMP/TIP Preventive Maintenance Schedule

Second Tuesday	1c6
MITRE [0830-1130]	1c6a
USC [1130-1430]	1c6b
Second Wednesday	1c7
CCA [0830-1130]	1c7a
STANFORD [1130-1430]	1c7b
RADC [1430-1730]	1c7c
Second Thursday	1c8
BELVOIR [0830-1130]	1c8a
UCSD [1130-1430]	1c8b
PATRICK [0930-1230]	1c8c
Third Monday	1c9
HARVARD [0830-1130]	1c9a
UCLA [1000-1300]	1c9b
Third Tuesday	1c10
RAND [1130-1430]	1c10a
SDC [1530-1830]	1c10b
Third Wednesday	1c11
ILLINOIS [0930-1230]	1c11a
ARPA [0830-1130]	1c11b
AMES IMP [1130-1430]	1c11c
Third Thursday	1c12
ETAC [0830-1130]	1c12a
FNWC [1130-1430]	1c12b

IMP/TIP Preventive Maintenance Schedule

Fourth Monday	1c13
CASE [0830-1130]	1c13a
HAWAII [1330-1630]	1c13b
Fourth Tuesday	1c14
NBS [0830-1130]	1c14a
LBL [1130-1430]	1c14b
Fourth Wednesday	1c15
LINCOLN [0830-1130]	1c15a
AMES TIP [1130-1430]	1c15b
Fourth Thursday	1c16
GWC [0930-1230]	1c16a
ISI [1300-1600]	1c16b
ABERDEEN [0830-1130]	1c16c

NWG/RFC# 445 AAM 22-JAN-73 13:52 14028
IMP/TIP Preventive Maintenance Schedule

(J14028) 22-JAN-73 13:52; Title: Author(s): McKenzie, Alex A./AAM; Distribution: Martin, Reg E., Leichner, Gene, Falk, Gil, Iseli, John, Donnelley, Jed E., Kantrowitz, William, Wolfberg, Michael S., Feinroth, Yeshiah S., Hurt, James, Hearn, Anthony C., Stein, James H., Shoshani, Arie, Harslem, Eric F., Metcalfe, Robert M. (Bob), Reussow, Bradley A., Reins, E. R. (Dick), Kadunce, Daniel L., McCutchen, Samuel P., Petregal, George N., Madden, James M., Young, Michael B., Padlipsky, Michael A., Stevenson, Schuyler, Deutsch, L. Peter, Davidson, John, O'Sullivan, Thomas, Seroussi, Sol F., Bradner, Scott, Thomas, Robert H., Thomas, John C., Romanelli, Michael J., Stoughton, Ronald M., Owen, A. D. (Buz), Fink, Robert L., Meir, Jaacov, North, Jeanne B., Crocker, Steve D., Lawrence, Thomas F., McConnell, John W., Ollikainen, Ari A. J., White, James E. (Jim), Hathaway, A. Wayne, Foulk, Patrick W., Winter, Richard A., Van Zoeren, Harold R., McKenzie, Alex A., Winett, Joel M., Bhushan, Abhay K., Pyke, Thomas N., Wilber, B. Michael, Feigenbaum, Edward A., Braden, Robert T., Pepin, James M., Wessler, Barry D., Melvin, John T./NLG; Sub-Collections: NIC NLG; RFC# 445; Clerk: AAM;

KIRK 22-JAN-73 20:40 14029

TRANSMITTAL TO: Professor John Seely Brown
Department ICS
University of California
Irvine, Cal. 92664

FROM: Susan Lee (SRI-ARC)
Station Agent 1

At the request of Tom O'Sullivan, Raytheon, Coordinator of the CBI Group, your name has been added to the Group list. We are enclosing the CBI Notes which have been issued and you will receive future ones as they are published.

1a

c: T. O'Sullivan 1b

NIC 9343
NIC 9608
NIC 9875
NIC 9876
NIC 9981
NIC 12741
NIC 12742
NIC 12755 1b1

(J14029) 22-JAN-73 20:40; Author(s): Kelley, Kirk E./KIRK;
Sub-Collections: SRI-ARC; Clerk: KIRK;

AAM 22-JAN-73 12:06 14030

Notice to AMES TIP users

The memory retrofit to the Ames TIP, previously announced for Wednesday, January 24, has been moved to Thursday, January 25. Due to the extent of the work required, it is likely that the TIP will also be down for some of Friday, January 26. 1

AAM 22-JAN-73 12:06 14030

Notice to AMES TIP users

(J14030) 22-JAN-73 12:06; Title: Author(s): McKenzie, Alex A./AAM; Distribution: Martin, Reg E., Leichner, Gene, Falk, Gil, Iseli, John, Donnelley, Jed E., Kantrowitz, William, Wolfberg, Michael S., Feinroth, Yeshiah S., Hurt, James, Hearn, Anthony C., Stein, James H., Shoshani, Arie, Harslem, Eric F., Metcalfe, Robert M. (Bob), Reussow, Bradley A., Reins, E. R. (Dick), Kadunce, Daniel L., McCutchen, Samuel P., Petregal, George N., Madden, James M., Young, Michael B., Padlipsky, Michael A., Stevenson, Schuyler, Deutsch, L. Peter, Davidson, John, O'Sullivan, Thomas, Seroussi, Sol F., Bradner, Scott, Thomas, Robert H., Thomas, John C., Romanelli, Michael J., Stoughton, Ronald M., Owen, A. D. (Buz), Fink, Robert L., Meir, Jaacov, North, Jeanne B., Crocker, Steve D., Lawrence, Thomas F., McConnell, John W., Ollikainen, Ari A. J., White, James E. (Jim), Hathaway, A. Wayne, Foulk, Patrick W., Winter, Richard A., Van Zoeren, Harold R., McKenzie, Alex A., Winett, Joel M., Bhushan, Abhay K., Pyke, Thomas N., Wilber, B. Michael, Feigenbaum, Edward A., Braden, Robert T., Pepin, James M., Wessler, Barry D., Melvin, John T./NLG; Sub-Collections: NIC NLG; Clerk: AAM;

AAM 22-JAN-73 12:14 14031

Notice to selected AMES TIP users

The memory retrofit to the Ames TIP, previously announced for Wednesday, January 24, has been moved to Thursday, January 25. Due to the extent of the work required, it is likely that the TIP will also be down for some of Friday, January 26 1

AAM 22-JAN-73 12:14 14031

Notice to selected AMES TIP users

(J14031) 22-JAN-73 12:14; Title: Author(s): McKenzie, Alex A./AAM; Distribution: Deutsch, L. Peter, English, William K., Fiala, Edward R., Kay, Alan C., Lampson, Butler W., McCreight, Edward M., Mitchell, James G., Payne, Adrienne M., Satterthwaite, Ed H., Sweet, Richard E., Teitelman, Warren, Metcalfe, Robert M. (Bob), Golding, Stan, Hart, James, Hathaway, A. Wayne, Linebarger, Robert N., Pucine, Gino, Rogallo, Sallie J./LPD WKE ERF ACK BWL EMM JGM AMP EHS RES WT RMM SG JH AWH RNL GINO SJR; Sub-Collections: NIC; Clerk: AAM;

AAM 22-JAN-73 12:17 14032

Notice to Ames TIP users

The memory retrofit to the Ames TIP, previously announced for Wednesday, January 24, has been moved to Thursday, January 25. Due to the extent of the work required, it is likely that the TIP will also be down for some of Friday, January 26. 1

AAM 22-JAN-73 12:17 14032

Notice to Ames TIP users

(J14032) 22-JAN-73 12:17; Title: Author(s): McKenzie, Alex A./AAM; Distribution: Jones, Diana L., Donnelley, Jed E., Basset, Margaret A. (Maggie), Colman, Harold, Neigus, Nancy J., Sack, Terry, McHale, Frances A. (Toni), Young, Helen D., Lee, Susan E., Gilliard, Lucille C. (Lucy), Falk, Gil, Collins, Ed J., Blunck, Gary, Heafner, John F., Beaman, Kathy, King, David J., Moody, C. Jane, Lemaro, Maria E., Shiffrin, James H., Pitkin, Sue, Fitzsimmons, Jerry, Hicks, Gregory P., Maxey, Gloria Jean, Peeler, Roberta J., Baxter, Faye, Fields, Craig, Benedict, James G., McCauley, Ermalee R., Iwamoto, Margaret, Larson, Dee, Doane, Robert E., Odom, Dan, Monroe, Brenda, Reynolds, Dorothy A., North, Jeanne B., Cutler, Pam J. Klotz, Barnett, Barbara, Forman, Ernest H., Golding, Stan, Chipman, Steve G., Barden, John P., Ginsberg, Martha A., Watkins, Shirley W., Connelly, Linda M., Troxel, Janet W., Rosewall, Connie D., Webster, Linda M., Coley, Anita L., Mostrom, Carol J./NSAG; Sub-Collections: NIC NSAG; Clerk: AAM;

AAM 22-JAN-73 6:27 14033

FTP Documentation-Improvement Meeting

I have called a meeting of representatives from MIT and BBN on Thursday, January 25 at 1:00 PM to attempt to generate a new FTP document from RFCs 354,385,414,418 and personal communications. It is my intent that this document should be released as a true "Request for Comments" and not as an "Official Specification". I intend to schedule a meeting of the entire FTP committee several weeks after the publication of this RFC, at which time we can consider release of an "Official Protocol". In spite of the fact that I think the above-mentioned meeting will not be making major changes, it is not my intent to exclude anyone from the FTP committee who wishes to attend. Therefore, if you haven't been personally invited but would like to come (or supply additional inputs) please call me at

(617) 491-1850 ext 441

for more information.

Alex McKenzie

1

AAM 22-JAN-73 6:27 14033

FTP Documentation-Improvement Meeting

(J14033) 22-JAN-73 6:27; Title: Author(s): McKenzie, Alex A./AAM; Distribution: Braden, Robert T., Chan, Arvolo, Harslem, Eric F., Heafner, John F., Hathaway, A. Wayne, Metcalfe, Robert M. (Bob), Postel, Jonathan B., Ryan, Neal D., Sundberg, Robert L., Tomlinson, Ray S., Watson, Richard W., White, James E. (Jim), Neigus, Nancy J., Bhushan, Abhay K., Bressler, Robert D. (Bob)/RTB AC EFH JFH AWH RMM JBP NDR RLS RST RWW JEW NJN AKB RDB2; Sub-Collections: NIC; Clerk: AAM;

CHI 22-JAN-73 9:44 14034

Some viewspec v nottes, Response to (Journal, 14024,)

Kirk, I think I understand your difficulty with viewspec v. Viewspec v merely deferes changing the image on the screen; it does not defer changing the file which is represented on the screen Consequently, if you do several edits to the same satement, your bug selections will get maped into characters other than those which were marked on the screen. For example, if you delete the first part of a statement then subsequent selections on that statement will get mapped to the wrong characters. To use viewspec v to advantage, delete things from statements from the end toward the begining and delete structures from the bottom toward the top. -- Charles. 1

CHI 22-JAN-73 9:44 14034

Some viewspec v nottes, Response to (Journal, 14024,)

(J14034) 22-JAN-73 9:44; Title: Author(s): Irby, Charles H./CHI;
Distribution: Kelley, Kirk E., Hopper, J. D., Kaye, Diane S., Lehtman,
Harvey G., Irby, Charles H., Kelley, Kirk E./kirk bugs ;
Sub-Collections: SRI-ARC BUGS; Clerk: CHI;

DIA 21-JAN-73 17:03 14035

A new version of the NLS call-return trace program

This describes a new tracing user program. It includes execution time and the trace of signals Find out how long your favorite routine takes

A new version of the NLS call-return trace program

A new wrinkle in the trace program: 1

A call-return trace can be obtained via user program in (andrews,ttrace,). This user program succeeds and surpasses (andrews,tracer,). 1a

The new one does basically what the old one did, but includes execution time as well as the trace information. Also, it traces signals (after a fashion) as well. 1a1

Execution times are given as elapsed time between "events" (where an event is a call or return or signal). The units are milliseconds. Also, execution times are summed for each procedure called and the sum is printed at the return from each procedure. 1a2

Signals are traced in the following form: 1a3

SIGNAL TO <routine>+<dsp>,<progname> <adr> OF (n,<adr>) 1a3a

Where <routine> and <progname> represent symbols, <dsp> is and octal displacement, <adr> is an octal address. 1a3b

This provides the fact that there was a signal, and <routine> was activated at <routine>+<dsp> (which is the <adr>), and the value of the signal was n (decimal number) and the message was at location <adr>. You have to look in the proper version of NLS to find the text of the message. 1a3c

The execution time is distorted somewhat by all this tracing. But not excessively so. I did some measuring: A simple procedure call and return (no arguments, no return value) is about 65 micro seconds. With the trace stuff on, It takes about 200 micro seconds. The time taken to do the PMAP JSYS (to get the trace data out to the file) is subtracted from the printed execution times. Hence the execution times are pretty accurate. The sum for a routine may be inflated somewhat if, in executing that routine, many calls and returns were done. Certainly compute bound loops and JSYS calls are timed quite accurately. 1a4

Some sample output and an explaination. 1b

Suppose one "branch" of output from the superwatch interpret trace command (which is used to print the trace.data file) looks like this: 1b1

The number following the procedure names and file names are octal addresses. The number in parens are elapsed times in ms. (decimal). The numbers in square brackets are total times in routines in ms. The total times are associated with the routine specified at the "source" structure wise. 1b1a

```
x,utilty 3456 (4)                                          1b1b
   y,utilty 3477 (2)                                       1b1b1
      qq,auxcod 3344 (0) R (1)                             1b1b1a
      zz,auxocd 3210 (4) R (0)                             1b1b1b
      R (3) [8]                                            1b1b1c
   R (0) [10]                                              1b1b2
SIGNAL TO mainctl+3,blah 56773 OF (2,334122) (0)           1b1c
   help,bloop 44567 (0)                                    1b1c1
      etc.                                                 1b1c1a
```

This reads as follows: 1b2

Procedure x in utilty was called. Whoever called him computed for 4 ms. between the time he was called and the time he called x. X called y 2 ms. later. Zero ms. later, y called qq in auxcod. QQ returned a millisecond later (or so) and y called zz four milliseconds after that return. ZZ didnt use much time and returned to y. Y then computed for 3 ms. and returned. The total time burned up by y is given as 8 ms. Then x returned and the total for x is given as 10 ms. Then a signal occurred in whoever called x. The value was 2 and the message text at 334122. Control was passed to mainctl+3 in file blah. He called help in bloop, etc. 1b2a

Elapsed times that are one ro two milliseconds are not significant except to say that the execution time was "small". Execution times that are a sum of many "small" elapsed times (say 5 or 10 ms.) are accurate within 1 or 2 ms. however. 1b3

DIA 21-JAN-73 17:03 14035

A new version of the NLS call-return trace program

(J14035) 21-JAN-73 17:03; Title: Author(s): Andrews, Don I./DIA; Distribution: Victor, Kenneth E. (Ken), White, James E. (Jim), Dornbush, Charles F., Michael, Elizabeth K., Vallee, Jacques F., Mitchell, James G., Deutsch, L. Peter, Kaye, Diane S., Andrews, Don I., Bass, Walt, Hopper, J. D., Irby, Charles H., Lehtman, Harvey G., Deutsch, L. Peter, Watson, Richard W., Wallace, Donald C. (Smokey), Victor, Kenneth E. (Ken)/NPG LPD RWW DCW KEV; Sub-Collections: SRI-ARC NPG; Clerk: DIA; Origin: <ANDREWS>BLURB.NLS;1, 21-JAN-73 16:23 DIA ;

DSK 22-JAN-73 14:02 14036

Viewspec "v" Reported Problem

Viewspec v either works or it doesn't work. If you have it on, and edit or jump, and it fails to recreate display, it WORKED. That's all it promises to do. Let me know if it allows the display to recreate.

1

DSK 22-JAN-73 14:02 14036

Viewspec "v" Reported Problem

(J14036) 22-JAN-73 14:02; Title: Author(s): Kaye, Diane S./DSK; Distribution: Kelley, Kirk E./kirk ; Sub-Collections: SRI-ARC; Clerk: DSK;

DVN 22-JAN-73 20:25 14037

Output to COM, Current State / Visitor Log: Mark Brown &Paul Johnson of DDSI

Mark Brown and Paul Johnson of DDSI passed briefly through today (January 18) on their way from Marin County to Los Angeles, 1

Walter Bass, Dick Watson, and I met with them. 2

At one point we summarized the problems that we (ARC) saw outstanding as follows: 3

(1) Proportional Spacing -- Paul asserts that proportional spacing is now working. We handed him a tape that contains proportional spacing and we will see what returns. 3a

(2) When we specify that the type face in a single line switch back between slanted and unslanted more than twice, the switch does not necessarily occur. 3b

Paul was ignorant of that problem and says he will look into it. 3b1

(3) The printing in the painted fonts is too heavy looking, particular at point sizes less than ten. Paul said he was aware of the problem and that they were going to improve it. 3c

Specifically he said they planned to reduce both the size and intensity (that is reduce the total number of photons) in the beam which painted the characters and to reduce it diferentially according to type size. 3c1

(4) Character sets -- Walt had asked for a complete ASCII character set. He subsequently discovered that our character set is 1962 ASCII which includes back arrow and up arrow (essential to L-10), but their ASCII is 1968 ASCII which has a circumflex and something or other at those spots. 3d

Paul and Walter worked out a solution. 3d1

We discussed briefly the future problems of large character sets. 3d2

Paul pointed out that in their stick fonts have a wide array of mathematical characters. 3d2a

(5) Halftones -- We agreed that we would attempt to incorporate halftones in our documents by ARC supplying to DDSI files on tape that included photo numbers and space where halftones or line drawings were supposed to occur and supplying to them with the tape the photos and line drawings

themselves cropped with instructions as to their size on the finished printed page. 3e

Paul asserted that they were subcontracting to have some other company working at their shop transfer images directly from microfilm to paper offset plates, rather than to an intermediate glossy paper step. He said that subcontractors would insert the halftones and line drawings on the plates at this step in the way normal to offset printing. 3e1

Hitherto they had been sending an intermediate paper step to their printer who made plates from those glossies, 3e2

I had the strong impression thathey were sending microfilm to Diline. 3e2a

(6) Turn around time -- They spoke of 3 or 4 days when we made the contract. Turn around has been running ten days to two weeks. Paul and Mark asserted that it has been faster this year. I intend to track more carefully turn around in the next few weeks. I'll make an arrangement with Kay. I also intend to talk to the SRI mail room about whether the tapes are moving to DDSI in the fastest possible way. 3f

Paul spoke with pride of a time of about 20 hours from the arrival of a tape in his hands until a roll of exposed film left his hand to their film processing laboratory. 4

In general, we mentioned to them future possibilities arising from our relationship. 5

We gave them a copy of Doug's paper (13445,). Dick Watson mentioned that there had been talk at the ARPA Principal Investigators' Meeting of some central COM available to anyone on the NET, and Dick and I mentioned that there had been talk of an academic journal for the Network which would probably be published through COM. 5a

DVN 22-JAN-73 20:25 14037

Output to COM, Current State / Visitor Log: Mark Brown &Paul Johnson of DDSI

(J14037) 22-JAN-73 20:25; Title: Author(s): Van Nouhuys, Dirk H./DVN; Distribution: Engelbart, Douglas C., Meyer, N. Dean, Watson, Richard W., Bass, Walt, Norton, James C., Byrd, Kay F./dce ndm rww wlb jcn kfb ; Sub-Collections: SRIARC DPCS ; Clerk: DVN; Origin: <VANNOUHUYS>JDDSI.NLS;2, 22-JAN-73 20:21 DVN ;

KIRK 22-JAN-73 17:51 14038

I have changed the default links for locator to be NIC. If there is a reason not to do this, let me know . (It's easy to do: Execute Ownership) 1

(J14038) 22-JAN-73 17:51; Author(s): Kelley, Kirk E./KIRK;
Distribution: Van Nouhuys, Dirk H./dvn ; Sub-Collections: SRI-ARC;
Clerk: KIRK;

MDK 22-JAN-73 15:01 14039

Exec-Level Journal Commands

Charles ... Regarding your recommendation for EXEC-level journal commands for Journal submission and Journal printout: 1

This is a good idea. As you know, several persons who use our system over the network have been asking for this for some time, so it will be a plus for us if we can implement it soon. 1a

The NIC feels a special responsibility in this area, since most of the complaints and requests come through us, and we will have to explain the use of the new command. Consequently, we would like to participate in the design of the syntax and semantics of this new command. 1b

MDK 22-JAN-73 15:01 14039

Exec-Level Journal Commands

(J14039) 22-JAN-73 15:01; Title: Author(s): Kudlick, Michael D./MDK ; Distribution: Hoffman, Carol B., Lee, Susan R., Michael, Elizabeth K., Dornbush, Charles F., ARC, Guest O., Feinler, Elizabeth J. (Jake), Handbook, Augmentation Research, Kelley, Kirk E., Meyer, N. Dean, Byrd, Kay F., Prather, Ralph, White, James E. (Jim), Vallee, Jacques F., Kaye, Diane S., Rech, Paul, Kudlick, Michael D., Ferguson, Ferg R., Lane, Linda L., Auerbach, Marilyn F., Bass, Walt, Engelbart, Douglas C., Hardeman, Beauregard A., Hardy, Martin E., Hopper, J. D., Irby, Charles H., Jernigan, Mil E., Lehtman, Harvey G., North, Jeanne B., Norton, James C., Page, Cindy, Paxton, William H., Peters, Jeffrey C., Ratliff, Jake, Van De Riet, Edwin K., Van Nouhuys, Dirk H., Victor, Kenneth E. (Ken), Wallace, Donald C. (Smokey), Watson, Richard W., Andrews, Don I./sri-arc ; Sub-Collections: SRI-ARC; Clerk: MDK;
Origin: <KUDLICK>JOURNAL.NLS;3, 22-JAN-73 14:48 MDK ;

MFA 22-JAN-73 15:46 14040

Response to some NLS language questions (13902,)

In response to some of your language questions - 1

I'm not that sensitive to entity defaulting - your proposal sounds fine 1a

I think making the jump to ... stuff like any other command a great idea, it seems to get in my way more that it helps me 1b

I would hate to see the jump to end of... stuff go away. It's not like I use it all the time, but I have certain regular jobs where it really comes in handy and saves me a few command steps. 1c

About CDOT, I agree that it should be changed as you indicate - CDOT seems to mostly unusable as it operates now in any command except insert. 1d

MFA 22-JAN-73 15:46 14040
Response to some NLS language questions (13902,)

(J14040) 22-JAN-73 15:46; Title: Author(s): Auerbach, Marilyn F./MFA; Distribution: Irby, Charles H./CHI; Sub-Collections: SRI-ARC; Clerk: MFA;

NJN 22-JAN-73 6:48 14041

sockets

Jon-- Sorry about last journal message; apology coming by mail. I got a note this morning from one of my favorite nuisances at Harvard who asked about the possibility of a network information socket that would list the names, nicknames, calling names for accessing sites on the net. It mighnt be a nice feature but I'm not really sure it is either necessary or worth a network socket. If you have any feeling for this let me know and I can take it up further with Tenex people. (I also tend to think they wouldn't be interested either.) Ciao --Nancy 1

NJN 22-JAN-73 6:48 14041

sockets

(J14041) 22-JAN-73 6:48; Title: Author(s): Neigus, Nancy J./NJN;
Distribution: Postel, Jonathan B./JBP; Sub-Collections: NIC; Clerk: NJN;

DVN 23-JAN-73 10:00 14042

Plans for Command Changes that May be of Interest to People on the Net.

You might be interested in two journal documents that plan changes in certain commands that will be important to people on the net:(journal,14006) and (journal,14039). 1

I'm sure Charles and Mike would be glad to hear your thoughts on these matters. 2

DVN 23-JAN-73 10:00 14042

Plans for Command Changes that May be of Interest to People on the Net.

(J14042) 23-JAN-73 10:00; Title: Author(s): Van Nouhuys, Dirk H./DVN; Distribution: Irby, Charles H., Kudlick, Michael D., Neigus, Nancy J., Forman, Ernest H., Crocker, David H./chi mdk njn ehf dhc ; Sub-Collections: SRI-ARC; Clerk: DVN;

LINKS 1

A "link" is a specially formatted text string that gives the location of any desired item of on-line information. 1a

A link can occur anywhere in a file. The information that it refers to can occur anywhere in that file, or in any other file that is on-line. 1a1

The use of a link is an important part of the NLS concept of files and information-handling. Specific beneficial aspects of links are: 1b

Using links enables the user to embed precise cross references in a file for subsequent on-line reading. 1b1

The link allows access to be made to individual statements, not merely to entire files. 1b2

With it, one can easily access information, and return easily to files that were previously accessed. 1b3

It removes the necessity of typing the file name or other identifying data every time one wants to access some information. 1b4

It allows the "view" of information to be specified at the same time that it is accessed. 1b5

Note, however, that although the address and viewspecs control the way a file appears when a link is first executed, the user may change any of these parameters once he has accessed a file using a link. 1b5a

The syntax of the link is: 1c

(directory, filename, address: viewspecs) 1c1

where 1c2

directory = the name of the directory containing the file named "filename". 1c2a

If the directory name is omitted, the directory to which the user is currently connected is the default value. 1c2a1

When the directory name is omitted, the delimiting comma following it should also be omitted. 1c2a1a

filename = the name of the file to be accessed. 1c2b

If the filename is omitted, the system assumes that the link refers to a location in the file that the link occurs in. 1c2b1

The filename cannot be omitted unless the directory name is also omitted. 1c2b2

When the filename is omitted, the delimiting comma following it should also be omitted. 1c2b2a

address = the number or name of the statement in the file that is to be accessed. 1c2c

If "address" is not specified, the system assumes the address is that of the origin statement of the file. 1c2c1

When the address is omitted, the delimiting colon following it should NOT be omitted. 1c2c1a

viewspecs = a series of specifications, or format codes, which control the way the file will appear when accessed through the link. 1c2d

If "viewspecs" are not specified, the system uses the viewspecs that are in effect when the link is executed. 1c2d1

Examples: 1d

(jones,summary,100:wm) 1d1

This link specifies statement number "100" in a file named "summary", contained in a directory named "jones". 1d1a

When the link is executed, statement numbers will be visible, because of viewspec m. Also, all levels and all lines will be visible, because of viewspec w. 1d1b

Any other viewspecs in effect at the time the link is executed will also control the appearance of the file. 1d1c

(myfile,:n) 1d2

This link specifies the file named "myfile", belonging to the current user. When the link is executed, the file will be positioned at statement 0. 1d2a

Statement numbers will not be visible (viewspec n). 1d2b

Any other viewspecs in effect at the time the link is executed will also control the appearance of the file. 1d2c

(200b) 1d3

This link specifies statement 200b in the file that contains the link. 1d3a

Any viewspecs in effect at the time the link is executed will control the appearance of the file. 1d3b

Further Notes on Link Syntax: 1e

If the directory name is omitted, the default is the directory of the file in which the link exists. 1e1

This convention allows the use of links that are in files created in other directories, even when the directory name has been omitted in those links. 1e1a

It also allows the user to omit the directory name from links between files that belong to a common directory. 1e1b

However, there is one exception to this default convention. 1e2

If the filename is a number (digits only), NLS first checks an NLS Journal directory to find the current location of a Journal document with that number. 1e2a

If no such journal file exists, the system then takes the default directory name as described above. 1e2b

MDK 23-JAN-73 9:15 14043

link description

(J14043) 23-JAN-73 9:15; Title: Author(s): Kudlick, Michael D./MDK ; Distribution: Kudlick, Michael D./mdk ; Sub-Collections: SRI-ARC; Clerk: MDK;
Origin: <KUDLICK>LINKS.NLS;2, 5-JAN-73 10:21 MDK ;

NJN 23-JAN-73 11:09 14044

nls trouble

Dirk-- I noticed another problem with nls this morning. It no longer allows me to update or Output File my initial file. In addition when i first tried updating it I got a message saying: 1

Update unnecessary: no changes <BBN-NET>NJN.NLS;18. 1a

I know I made changes in it yesterday, and did not update when I was through. However, I did not get any journal mail after that. So I added some junk statements and then tried to update and got: 2

You cannot write on <BBN-NET>NJN.NLS;19. 2a

Note different version number. 2b

Bob Bressler tried to update his initial file after receiving and deleting his journal mail, but got the same negative results. What has happened? 3

I tried to send this to you via SNDMSG from SRI (i.e. user to user on the same host) and i got 4

VANNOUHUYS -- can't. 4a

I then tried to send from BBN and got queued even though your host was up. That means your sndmsg processor is not delivering any messages. See Smokey; it is your system that is at fault. 5

SRI is the only Tenex that still has the old version of sndmsg. 5a

Got your letter. Am getting myself together to answer it. 6

--Nancy 7

NJN 23-JAN-73 11:09 14044

nls trouble

(J14044) 23-JAN-73 11:09; Title: Author(s): Neigus, Nancy J./NJN; Distribution: Van Nouhuys, Dirk H./DVN; Sub-Collections: NIC; Clerk: NJN;

Tree Meta Report -- Preliminary Draft

<LEHTMAN>1TMR.NLS;10, 8-APR-71 16:44 HGL ;

Tree Meta--
A Metacompiler for the Augmentation Research Center

D. I. Andrews
H. G. Lehtman
W. H. Paxton

25 March 1971

Tree Meta - ABSTRACT

Tree Meta is a compiler-compiler system for context-free languages developed at the Augmentation Research Center (ARC) of the Stanford Research Institute in Menlo Park, California. The parsing statements of the metalanguage resemble Backus-Naur Form with embedded tree-building directives. Unparsing rules include extensive tree-scanning and code-generation constructions. All compilers produced by the system are single pass compilers that produce loadable binary files. 1b1

This is a preliminary draft of a more complete report due to be completed shortly. The finished version will include a detailed example of a compiler for a small algebraic language; excerpts from that example will be incorporated into the semantic descriptions of the language. The Program Environment section will be cleaner and hopefully more easy to understand. There will be a description of the use of Tree Meta to bootstrap itself from one machine to another. 1b2

The present report, while essentially new, makes use of some sections from a report on a more primitive system which appeared in (ENGLEBART1) in 1968. Tree Meta is a constantly evolving programming system; thus some of the more recent additions may not yet be incorporated into this text. This document describes Tree Meta as of 1 March 1971. 1b3

Because this is a draft, please don't quote it without permission. Comments, questions and suggestions will be appreciated. They should be addressed to Harvey Lehtman at ARC. 1b4

1c

Tree Meta - CONTENTS

Terms such as "metalanguage" and "metacompiler" have a variety of meanings. Their use within this report, however, is well defined. 1d1

"Language," without the prefix "meta," means any formal computer language such as ALGOL or FORTRAN. Any metalangauge is also a language. 1d1a

A compiler is a computer program that reads a formal-language program as input and translates that program into instructions that may be executed by a computer. The term "compiler" also means a listing of the instructions of the compiler. 1d1b

A language that can be used to describe other languages is a metalanguage. English is an informal, general metalanguage that can describe any formal language. Backus-Naur Form or BNF (NAUR1) is a formal metalanguage used to define ALGOL. BNF is weak, for it describes only the syntax of ALGOL, and says nothing about the semantics or meaning. English, on the other hand, is powerful. Its informality, however, prohibits its easy translation into computer programs. 1d1c

A metacompiler, in the most general sense of the term, is a program that reads a metalanguage program as input and translates that program into a set of instructions. If the input program is a complete description of a formal language, the result of the translation is a compiler for the language. 1d1d

Tree Meta is built to deal with a specific set of languages and an even more specific set of users. There is no attempt to design universal languages, or machine independent languages, or to achieve any of the other goals of many compiler-compiler systems. 1d2

Tree Meta is intended to parse context-free languages with limited backup. There is no intent or desire on the part of the users to deal with such problems as the FORTRAN "continue" statement, the PL/I "enough ends to match," or the ALGOL "is it procedure or is it a variable" question. Tree Meta is only one part of the system-building technique used at ARC. The ARC system is flexible at all levels and the design philosophy has been to take the easy way out rather than fight old problems. 1d3

The metacompilers are expressible in their own language,

hence the prefix "meta". Tree Meta makes use of a common top down parsing algorithm which is described below. 1d4

The following formal discussion of top-down parsing algorithms relies heavily on definitions and formalisms that are standard in the literature. For a language L, with vocabulary V, nonterminal vocabulary N, productions P, and start symbol S, the top-down parse of a string u in L starts with S and looks for a sequence of productions such that S=>u (S produces u). 1d4a

Let 1d4a1

V = [E, T, F, +, *, (,), X] 1d4a1a

N = [E, T, F] 1d4a1b

P = 1d4a1c

[E ::= T / T + F, 1d4a1c1

T ::= F / F * T, 1d4a1c2

F ::= X / (E)] 1d4a1c3

L = (V,N,P,E) 1d4a1d

The following intentionally incomplete ALGOL procedures will perform a top-down analysis of strings in L. 1d4a2

```
boolean procedure E;                                        1d4a2a

   E := if T then (                                         1d4a2a1

      if issymbol('+') then F else true) else               1d4a2a1a

      false;                                                1d4a2a1b

   comment issymbol (arg) is a Boolean procedure
   that compares the next symbol in the input
   string with its argument, arg.  If they match,
   the input stream is advanced;                            1d4a2a2

boolean procedure T;                                        1d4a2b

   T := if F then (                                         1d4a2b1
```

```
        if issymbol('*') then T else true)          1d4a2b1a
        else false;                                 1d4a2b1b
    boolean procedure F;                            1d4a2c
      F := if issymbol('X') then                    1d4a2c1
        true                                        1d4a2c1a
        else if issymbol('(') then (                1d4a2c1b
          if E then (                               1d4a2c1b1
            if issymbol(')') then                   1d4a2c1b1a
              true                                  1d4a2c1b1a1
              else false)                           1d4a2c1b1a2
            else false)                             1d4a2c1b1b
          else false;                               1d4a2c1b2
```

The left-recursion problem in top-down, goal-oriented recognizers can readily be seen by a slight modification of L. Change the first production to "E ::= T / E + T" and the procedure for E in the corresponding manner to "E := if T then true else if...." 1d4a3

Parsing the string "X+X", the procedure E will call T, which calls F, which tests for "X" and gives the result "true." E is then true but only the first element of the string is in the analysis, and the parse stops before completion. If the input string is not a member of the language, T is false and E loops infinitely. This is because of the implicit goal orientaion of the top-down parsing schemes. When E becomes a new sub-goal, the first step is to create again a new sub-goal E; in other words, E calls itself recursively. 1d4a3a

The solution to the problem used in Tree Meta is the arbitrary number operator. In Tree Meta the first production could be similar to the rule E ::= T$("+" T) where the dollar sign and the

parentheses indicate that the quantity can be repeated any number of times including 0. 1d4a3b

Tree Meta makes no check to insure that the compiler it is producing lacks syntax rules containing left recursion. Including instances of left recursion is one of the more common mistakes made by inexperienced metalanguage programmers. 1d4a3c

The input language to the metacompiler closely resembles BNF. The primary difference between a BNF rule

```
<go to> -> go to <label>
```

and a metalanguage rule

```
goto = "GO" "TO" .ID;
```

is that the metalanguage has been designed to use a computer-oriented character set and simply delimited basic entities. 1d4b

The arbitrary-number operators and parentheses construction of the metalanguage are lacking in BNF. For example:

```
TERM = FACTOR $(("*" / "/" / "↑")
FACTOR);
```

is a metalanguage rule that would replace the 2 BNF rules:

```
<term> ::= <factor1>
<factor1> ::= <factor> /
              <factor> * <factor1> /
              <factor> "/" <factor1> /
              <factor> ↑ <factor1>
```

(Note that in the second rule we have enclosed the slash (/) in quotes to avoid confusion with the BNF alternative choice metasymbol.) 1d4c

The ability of the compilers to be expressed in their own language has resulted in the proliferation of metacompiler systems. Each one is easily bootstrapped from a more primitive version, and complex compilers are built with little programming or debugging effort. 1d4d

A history of compiler-compiler systems may be found in the article on Translator Writing Systems by Feldman and Gries (FELDMAN1). Tree Meta owes much to earlier systems. 1d5

A brief history of the lineage of Tree Meta will be included here in the completed report. 1d5a

1e

Contained here is a tutorial introduction to the syntax and operation of meta programs and Tree Meta in particular. Only the essential features appear in the examples in order to make the basic functioning clear. 1e1

A metaprogram is a set of metalanguage rules. Each rule has the form of a modified BNF rule with compiling directives embedded in the syntactic description. 1e2

The Tree Meta compiler converts each of the rules to a set of instructions for the computer which function like a recursive subroutine. 1e2a

As the rules (acting as instructions) compile a program, they read an input stream of characters one character at a time. Each new character is subjected to a series of tests until an appropriate syntactic description is found for that character. The next character is then read, and the rule testing moves forward through the input. 1e2b

The following four rules illustrate the basic constructions in the system. They will be referred to later by the reference numbers R1A through R4A. 1e3

```
R1A exp = term $("+" exp / "-" exp ) ;
```
1e3a

```
R2A term = factor $("*" factor / "/" factor);
```
1e3b

```
R3A factor = "-" factor / prim;
```
1e3c

```
R4A prim = .ID / .NUM / "(" exp ")";
```
1e3d

The identifier to the left of the initial equal sign names the rule. This name is used to refer to the rule from other rules. The name of rule R1A is "exp". 1e3e

The right part of the rule-- everything between the initial equal sign and the trailing semicolon-- is the part of the rule which effects the scanning of the input. Several basic types of entities may occur in a right part. Each of the entities represents some sort of a test which results in success or failure. 1e3f

A string of characters between quotation marks (") represents a literal string. These literal strings are tested against the input stream as characters are read. 1e3f1

Rule names may also occur in a right part. Such an occurence means that the named rule is invoked. R3A defines a "factor" as being either a minus sign followed by a "factor" or just a "prim". 1e3f2

The right part of the rule "factor" has just been defined as "a string of elements," "or" "another string of elements." The "or's" are indicated by slash marks (/) and each individual string is called an alternative. Thus, in the above example, the minus sign and the rule name "factor" are two elements in R3A. These two elements make up an alternative of the rule. The rule has two alternatives, and the second one is simply a "prim". 1e3f3

The dollar sign is the arbitrary number operator in the metalanguage. A dollar sign must be followed by a single element, and it indicates that this element may occur an arbitrary number of times (including zero). Parentheses may be used to group a set of elements into a single element as in R1A and R2A. 1e3f4

A single element enclosed in square brackets, "[" and "]", is optional and is not necessary to the syntax. The element may be a grouped set. 1e3f4a

The number sign, "#", is used before an element which may occur any number of times equal to or greater than one. 1e3f4b

If the "$" or "#" is followed by a construction enclosed by angle brackets, "<" and ">", followed by a single item, a series of those items (including a zero length series for "$") separated by the enclosed construction may occur. For example, the rule

```
term = factor $( "*" factor );
```

is equivalent to

```
term = #<"*">factor; .
```

1e3f4c

Some basic entities may be seen in rule R4A. They are the basic recognizers of the metacompiler system. A basic recognizer is a program in the Tree Meta library described in the Program Environment section below that may be called to test the input stream for an occurrence of a particular entity. In Tree Meta three basic recognizers are "identifier" as .ID, "number" as .NUM, and "string" as .SR. 1e3f5

An identifier is a string of lower case letters and digits that begins with a lower case letter. It may be any length. 1e3f5a

A .SR is a string of characters that does not contain a quote (") and is bounded by quotes. The string may contain carriage returns, tabs etc. It may be any length also. 1e3f5b

A .NUM is a string of digits, with an optional "B" at end to indicate the base 8, a "D" to indicate base 10, or an "M" to indicate that the number is a mask. Numbers without a base charcter are taken to the base 10. If the base is indicated, a number may follow as a scale factor. For example 51D3 is 51000; the octal number 77000B may also be specified as 77B3. 1e3f5c

**If the base character is an "M" the number to the left indicates the number of bits to be turned on in the mask; the optional number to the right is an octal scale factor. If the scale factor is absent the mask is right justified. Thus 18M is an 18 bit mask on the right half of a 36-bit PDP-10 word; 18M6 is an 18 bit mask on the left half word. 1e3f5d

In addition to the three basic recognizers .ID, .NUM, and .SR, others are occasionally very useful. 1e3f5e

The symbol .UID indicates an identifier composed entirely of upper case letters. 1e3f5e1

The symbol .SR1 indicates a single character string as a single character preceded by a single quote ('). 1e3f5e2

The symbol .CHR indicates any character. In the parse rules, .CHR causes the next character on the input stream to be taken as input regardless of what it is. The character is stored in a special way, and hence references to it are not exactly the same as for the other basic recognizers. In node testing, if one wishes to check for the occurrence of a particular character that was recognized by a

.CHR, one uses the single quote-character construction. 1e3f5e3

Suppose that the input stream contains the string x+y when the rule "exp" is invoked during a compilation. 1e4

"exp" first calls rule "term", which calls "factor", which tests for a minus sign. This test fails. Then, as an alternative, "factor" calls "prim" which tests for an identifier. The character x is an identifier; it is recognized and the input stream advances one character. 1e4a

"prim" returns a value of "true" to "factor", which in turn returns to "term". "term" tests for an asterisk and fails. It then tests for a slash and fails. The dollar sign in front of the parenthesized group in "term", however, means that the rule has succeeded because "term" has found a "factor" followed by zero occurrences of "asterisk 'factor'" or "slash 'factor'". Thus "term" returns a "true" value to "exp". "exp" now tests for a plus sign and finds it. The input stream advances another character. 1e4b

"exp" then calls on itself. All necessary information is saved for return to the right place. In calling on itself, it goes through the sequence just described until it recognizes the y. 1e4c

If the plus in the input were replaced by a minus, "exp", after recognizing the identifier x, would start on the first alternative in the parenthesized group and would fail upon checking for a plus sign. It would start on the second alternative in a similar manner and succeed. If the input stream contained only the identifier x, "exp" would still succeed because the elements following the "term" are enclosed in square brackets and thus are optional. 1e4d

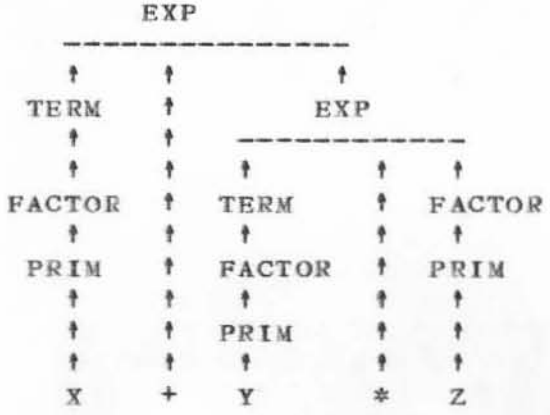
Thinking of the rules in this way is confusing and tedious. It is best to think of each rule separately. For example: one should think of R2A as defining a "term" to be a series of "factor"s separated by asterisks and slashes and not attempt to think of all the possible things a "factor" could be. 1e4e

Tree Meta generally builds a parse tree of the input stream before producing any output. Before we describe the

syntax of node generation, let us first discuss parse trees. 1e5

A parse tree is a structural description of the input stream in terms of the given grammar. 1e5a

Using the four rules above, the input stream
X+Y*Z
might generate the following parse tree (this tree is formed by making a node each time a rule is called)



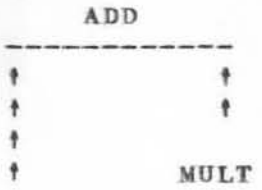
1e5a1

In this tree each node is either the name of a rule or one of the primary entities recognized by the basic recognizer routines. 1e5a2

In this tree there is much substructure. For example, y is a "prim" which is a "factor", which is the left member of a "term". This degree of substructure is generally unnecessary and often undesirable. 1e5a3

A tree produced by a metacompiler program is generally simpler than the one above, yet it contains sufficient information to complete the compilation. 1e5b

The parse tree produced by the following example (R1B-R4B) is



1e5b1

In this tree the names of the nodes are not the rule names of the syntactic definitions, but rather the names of rules that will be used to generate the code from the tree. 1e5b2

The rules that produce the above tree are the same as the four previous rules with new syntax additions to perform the appropriate node generation. The complete rules are: 1e5b3

```
R1B exp = term $("+" :add[2] / "-" exp :sub[2] ) ;
```
1e5b3a

```
R2B  term = factor $("*" factor :mult[2]/ "/"
factor :divd[2] );
```
1e5b3b

```
R3B factor = "-" factor :minus[1] / prim;
```
1e5b3c

```
R4B  prim = .ID / .NUM / "(" exp ")";
```
1e5b3d

As these rules scan an input stream, they perform just like the first set. As the entities are recognized, however, they are stored on a push-down stack until the node-generation elements remove them to make trees. We will step through these rules with the same sample input stream:

```
X+Y*Z
```
1e5c

EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the X. The input stream moves forward and the X is put on a stack. 1e5c1

PRIM returns to FACTOR, which returns to TERM, which returns to EXP. The plus sign is recognized and EXP is again called. Again EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the Y. The input stream is advanced, and Y is put on the push-down stack. The stack now contains Y and X, and the next character on the input stream is the asterisk. 1e5c2

PRIM returns to FACTOR, which returns to TERM. The

asterisk is recognized and the input is advanced another character. The asterisk is thrown away since only basic entities are put on the stack. 1e5c3

The rule TERM now calls FACTOR, which calls PRIM, which recognizes the Z, advances the input stream, and puts the Z on the push-down stack. 1e5c4

The :MULT is now processed. The MULT names the next node to be put in the tree. Later we will see that in a complete metacompiler program there will be a rule named MULT which will be processed when the time comes to produce code from the tree. Next, the [2] is processed. This tells the system to construct a portion of a tree. The branch is to have two nodes, and they are to be the last two entities recognized (they are on the stack). The name of the branch is to be MULT, since that was the last name given. The branch is constructed and the top two items of the stack are replaced by a reference to the new node of the tree, which we will indicate by (MULT). 1e5c5

The stack now contains 1e5c5a

(MULT) 1e5c5a1

X 1e5c5a2

The parse tree is now

1e5c5b

Notice that the nodes are assembled left-to-right, and that the original order of recognition is retained. 1e5c5c

Rule TERM now returns to EXP which names the next node by executing the :ADD-- i.e., names the next node for the tree. The [2] is now executed. A branch of the tree is generated containing the top two items of the stack and whose name is ADD. The top two items of the stack are replaced by a reference to the ADD node which is the top of the tree. The tree is now complete, as first shown, and all the input has been passed over. All that remains

is to invoke the unparse rules to scan the tree and produce output. 1e5c6

The unparse rules have two functions: they test the tree in much the same way as the parsing rules test the input stream and they produce output. This testing of the tree allows the output to be based on the deep structure of the input, and hence more efficient code may be produced. 1e6

Before we discuss the node-testing features, let us first describe the various types of output that may be produced. The following list of output-generation features in the metacompiler system is enough to understand most examples. 1e6a

The output is instruction oriented. An instruction is formed by writing any number of entities or values needed to make up that instruction and then closing the instruction. A comma is usually written to close an instruction. 1e6a1

Computer operation codes for the output are generally predefined as all-upper-case identifiers. Symbols used in the output are written as lower case identifiers; nodes on the tree may be referenced and put out. 1e6a2

As we shall see, one may declare certain quantities to be opcodes or set values. Additionally one may declare in the initialization header various identifiers to represent fields in the output words. These field identifiers serve as qualifiers on other variables which are to be put into output words and indicate that the value of the variable is to go into the position specified by the field. In the output words opcodes are ORd, fields are replaced and ordinary values added into place. 1e6a3

As can be seen in the last example of a tree, a node of the tree may be either the name of an unparse rule, such as ADD, or one of the basic entities recognized during the parse, such as the identifier X. 1e6a4

Suppose that the expression X+Y*Z has been parsed and the program is in the ADD unparse rule processing the ADD node (later we will see how this state is reached). To put the identifier X

into the output stream, one writes "*1" (meaning "the first node below") as an element. For example, to generate an instruction with the operation code ADD and the operand field X, one would write: 1e6a4a

```
ADD *1,                                                    1e6a4a1
```

If the node referenced (e.g., *1 or *2) is not a terminal node, as for example the right side (indicated by *2) of the ADD node, then the unparse rule named in the node below is invoked. In this case the MULT unparse rule would be invoked, and would eventually return. The ADD rule would continue immediately after the "*2." 1e6a4b

Generated labels are handled automatically. As each unparse rule is entered, a new set of labels is generated. A label is referred to by a number sign followed by a number (e.g. #2). Every time a label is mentioned during the execution of an unparse rule, the value of the label is added to the instruction being formed. If another rule is invoked in the middle of a rule, all the labels are saved and new ones generated (i.e., generated labels are local to a particular rule). When a return is made, the previous labels are restored. 1e6a5

As trees are being built during the parse phase, a time comes when it is necessary to generate code from the tree. To do this one writes an asterisk as an element of a parse rule--for example,

```
R5B  program = "PROGRAM" $(st *) "END";
```

which generates code for each statement after it has been entirely parsed. When the asterisk is executed, control of the program is transferred to the unparse rule whose name is the root (top node or last generated node) of the tree. When return is finally made to the rule which initiated the output, the entire tree is cleared and the generation process begins anew. 1e7

An unparse rule is a rule name followed by a series of output rules. Each output rule begins with one or more tests of nodes. The series of output rules make up a set of highest level alternatives. When an unparse rule is called, the tests for the first output rule is made. If it is satisfied, the remainder of the alternative is executed; if it is false, the next alternative output

rule test is made. This process continues until either a successful test is made or all the alternatives have been tried. If a test is successful, the alternative is executed and a "return true" is made from the unparse rule. If no test is successful, a "return false" is made. 1e7a

The simplest test checks that the correct number of nodes emanate from the node being processed. The ADD rule may begin

add [-,-] =>

The string within the brackets is known as an out-test. Individual items of the out-test are separated by commas. The hyphens are individual items of the out-test. Each item tests a node. All that the hyphen requires is that a node be present. There are, of course, more complicated node testing elements. Descriptions of some follow: 1e7b

The nodes emanating from the node being evaluated are referred to as *1, *2, etc., counting from left to right. To test for equality between nodes, one merely writes *i for some i as the desired item in an out-test. For example, to see if node 2 is the same as node 1, one could write either [-,*1] or [*2,-]. To see if the third node is the same as the first,

**one could write [-,-,*1]. Note that [*1,-] always succeeds if [-,-] succeeds and thus the *1 in this situation is redundant. 1e7b1

One may test to see if a node is an element that was generated by a basic recognizer by mentioning the name of the recognizer. Thus to see if the node is an identifier one writes .ID. To test whether the first node emanating from the ADD is an identifier and if the second node exists, one writes [.ID,-]. 1e7b2

An .ID is represented as a literal string on a node. To check for a literal string on a node one may write a string as an item in an out-test. The construction [-,"x"] tests to be sure that there are two nodes and that the second node is the identifier x. The second node will have been recognized by the .ID basic recognizer during the parse phase. 1e7b3

One checks for a number in the node simply by writing that number as an element of the out-test. For example [23] will ensure that there is one node and

that it contains the number 23. The number may have been recognized by the .NUM recognizer. 1e7b4

To see if a node is a generated label one may write a number sign followed by a number or identifier. If the node is not a generated label the test fails. If it is a generated label the test is successful and the label is associated with the number or identifier following the number sign. To refer to the label in the unparse rule, one writes the number sign followed by the number or identifier. One may also write the symbol "LABL" to make the test without being able to refer to the label in the rule. 1e7b5

One may test to see if the node has a specified name and form. Suppose that one had generated a node named STORE. The left node emanating from it is the name of a variable and on the right is the tree for an expression. An unparse rule may begin as follows:

```
store[ -,add[ *1,1 ]] =>  AOS  *1, ;
```

The *1 as an item of the ADD refers to the left node of the STORE. 1e7b6

Only a tree such as

1e7b6a

would satisfy the test; the two identifiers must be the same or the test fails. An expression such as X ← X + 1 meets all the requirements. The code generated (for the PDP- 10) would be the single instruction AOS X, which increments the cell X by one. 1e7b6b

If one wishes to eliminate the test at the head of the out-rule, one may omit the out-test. A rule which has no out-test may have only one output rule. The rule

```
mt  => TRUE;
```

is frequently used in place of an optional item which is absent. It may be tested in other unparse rules but it always does nothing itself and returns. 1e7b7

Output rules may also deliver values to their caller if the "=>" is replaced by a ":=". The final value in the output expression is assigned to the value of the identifier. 1e7b8

Each output expression (genexp), or highest-level alternative, in an unparse rule is made up of alternatives. These alternatives are separated by slashes, as were the alternatives in the parse rules. 1e7c

The alternatives of the output expression are made up of output tests (otests) and output generation and control elements (genelts). If an alternative contains a test, the test is made. If it fails and the test was the first element in the alternative, the next alternative in the output rule is tried. If the test is successful, control proceeds to the next element of the alternative. When the alternative is done, a return is made from the unparse rule. If the alternatives are exhausted because a non-optional test failed in the first position of every alternative of the genexp or if a non-optional test not in the first position of an alternative fails, the whole unparse rule fails and a compiler error is indicated. 1e7c1

The otest in a genexp resembles the node tests for the output rule and the syntax tests. There are several types of otest. 1e7c2

Any non-terminal node in the tree may be invoked by referring to its position in the tree rather than its name. For example, *2 would invoke the second node from the left. During operation of that rule, all tree references are relative to that node. If the rule returns "true" the genexp proceeds to the next element. If it returns "false" and this is the first element of the genexp the next alternative of the genexp is tried. 1e7c2a

Another type of test is made by invoking another unparse rule by name. To call another unparse rule from an genexp one writes the name of that rule followed by an argument list enclosed in brackets. This is, of course, similar to the node construction element in a parse rule. The argument list is a list of nodes in the tree.

(Arguments in this context may also be indications that new nodes are to be created.) These nodes are used to build a portion of a tree, and when the call is made the rule being called sees the argument list as its set of nodes to analyze. 1e7c2b

For example:

```
add[minus[-],-] => sub[*2,*1:1];
```

Here node ADD is being evaluated; the leftmost sub-node, *1, is tested for a MINUS node with exactly one sub-node emanating from it; a second node must also be present. MINUS is the name of another unparse rule which is tested for presence but is not invoked. 1e7c2b1

Special types of arguments may be specified in the argument list. Nodes are written as *1, *2, etc. To reach other nodes of the tree one may write such things as *1:2, which means "the second node emanating from the first node emanating from the node being evaluated." Referring to the tree for the expression X+Y*Z, if ADD is being evaluated, *2:1 is Y. 1e7c2b2

If a generated label is written as an argument, it is generated at that time and passed to the called unparse rule so that that rule may use it or pass it on to other rules. The generated label is written just as it is in an output element-- a number sign followed by a number or identifier. 1e7c2b3

Number values can be passed as a node by writing an equals sign-number, for example =7. 1e7c2b4

One may also test for attributes on nodes, flags, skip returns from calls on library routines, and various other relational checks described below. 1e7c2c

The calls on other unparse rules may occur anywhere in an genexp. This use of extra rules helps to make the output rules more concise because the invocation of an unparse rule generally causes output. 1e7c3

The rest of an genexp is made up of output elements to produce instructions and controls to control execution, as discussed before. 1e7c4

Genexps may be nested, using parentheses, in the same way as the alternatives of the parse rules. 1e7d

There are a few features of Tree Meta that are not essential but do make programming easier for the user: 1e8

As the parse rules proceed through the input stream they may come to the middle of a parse alternative and fail. Failure may occur for two reasons: backup is necessary to parse the input, or there is a syntax error in the input. Backup will not be covered in this introductory chapter. If a syntax error occurs the system prints out a portion of the input stream up to the character which cannot be parsed. The system then executes the error recovery rule specified in the header of the compiler. 1e8a

Comments may be inserted anywhere in a metalanguage program where blanks may occur. A comment begins and ends with a percent sign and may contain any character except a percent sign. 1e8b

&SRI-ARC 23-JAN-73 14:00 14045
Tree Meta Report -- Preliminary Draft

(J14045) 23-JAN-73 14:00; Title: Author(s): Stanford Research Institute/&SRI-ARC ; Distribution: Auerbach, Marilyn F./MFA; Sub-Collections: SRI-ARC; Clerk: MFA;

Tree Meta Report -- Preliminary Report, Formal Description

<LEHTMAN>2TMR.NLS;9, 7-APR-71 14:03 HGL ; 1

Tree Meta - FORMAL DESCRIPTION--
Detailed example

This section gives as an example a Tree Meta description of a small, simple language which Tree Meta will compile into a compiler for source code written in that language. The example is explained in detail to illustrate how the metalanguage constructs of Tree Meta are used. 1a1

Detailed example-- Not included in preliminary draft. 1a2

1b

Tree Meta - FORMAL DESCRIPTION--
Syntax and semantics

The following formal description of the tree meta language is based on the syntax definition of the Tree Meta program itself. Actually some syntax definition rules have been changed slightly for clarity. (Note that these changes make the syntax presented unusable as the syntax of a meta-compiler.) 1b1

The formal description may be somewhat difficult to read since the syntax used is the same as the syntax being defined, but accompanying explanations should help overcome this problem. 1b1a

A reference table of the complete syntax of Tree Meta as of this writing may be found in the the following section. 1b1b

Compiler header 1b2

program = 1b2a

There are two types of input files recognized by the Tree Meta system. 1b2a1

"FILE" .ID ["CHECK"] "META" .ID #headelt #rule
"END" / 1b2a2

The first represents a meta compiler program; it has header elements followed by a body made up of rules and is terminated by an "END" symbol. 1b2a2a

The "FILE" .ID construction corresponds to the TITLE statement of the PDP-10 Macro Assembler. The ID appears as the name of the assembled program and is used when debugging with the Dynamic Debugging Technique (DDT) to gain access to the program's symbol table. 1b2a2a1

The optional "CHECK" may be useful in checking out new compilers. It causes the check flag to be set. 1b2a2a2

At present this flag is used to produce code to check whether or not a node reference is a terminal node in a context in which one is needed. If not, a compiler error is noted. If the "CHECK" is removed, the check flag is not set, the code to make the test is not

put out, and, if the compiler is not perfect, bad things will happen; if it is perfect, however, it will go a bit faster. 1b2a2a2a

A user may use this construction in conjunction with new library routines to make other checks on his compilers during debugging. 1b2a2a2b

The mandatory identifier after the META is the name of the rule that is first invoked and which will start the parse. When the end of that rule is reached, compilation will end. If it fails, compilation will end immediately with a syntax error. (See the discussion of initialization in the Program Environment section.) 1b2a2a3

"LIBRARY" $.ID ; 1b2a3

The other input file possibility, the word "LIBRARY" followed by a list of IDs, sets the library attribute on each of the IDs listed. The Tree Meta system itself would then know these IDs were defined in the library. This input file is used to list conflicts between the meta-compiler and the library with which it will be loaded. 1b2a3a

headelt = 1b2b

Heading elements must appear in a group before rules. Although it is not indicated in the syntax, the error and size constructions are required in every compiler. If they are omitted, correct compilation will result, but some undefined symbols will show up at load time. Of course, it is entirely possible that the omissions are irrelevant to the user's compiler. 1b2b1

"ERROR:" scn exp '; / 1b2b2

ERROR defines a rule which will be invoked for recovery whenever an error occurs in compilation; it must begin with a scan (scn) which is discussed below. 1b2b2a

```
"SIZE:"      $( ('K /'N /'M /'G /'S /'L) '=  .NUM )
';  /                                                    1b2b3
```

SIZE defines the sizes of the various storage areas for the compiler being created. The storage areas are designated by K, N, M, G, S and L. If any are missing, they will be undefined at load time. The actual sizes needed for a compiler must be found by experiment. The amount of each area used by the compiler is written out after each time the compiler is used. At this writing Meta itself uses K = 75, N = 900, M = 110, G = 50, S = 1400, L = 250. 1b2b3a

The K-stack contains the symbols recognized by the basic recognizers.
The N-stack contains nodes in the tree being constructed.
The M-stack is used to hold call return locations.
The G-stack contains generated labels.
The S-storage area is used for symbol storage.
The L-storage area contains literals. 1b2b3a1

A more detailed discussion of the use of these areas is in the Program Environment section. 1b2b3a2

```
"FLAGS:"     $.ID ';    /                                 1b2b4
```

FLAGS declares the names of flags which may be set, reset and tested by the compiler through the use of the &FS &FR and ?F constructions. 1b2b4a

```
"ATTRIBUTES:" $.ID ';    /                               1b2b5
```

ATTRIBUTES defines new attributes to be used in the compiler. Each symbol table item recognized by the compiler (ID, UID, SR, SR1) has these attributes as well as a standard set (all initialized to off or 0); they may be set, reset and tested by @S @R and ?@. 1b2b5a

Six standard attributes are included in every Meta compiler (their use is described in the Program Environment section): 1b2b5a1

defned-- defined=1, undefined=0.

```
            reloca-- relocatable=1, absolute=0.
            opcode-- opcode=1, other=0.
            extern-- external=1, local=0.
            linked-- linked or referenced=1,
            unlinked=0=unreferenced.
            noddt--  noddt=1, to be passed to DDT=0.    1b2b5a1a

        The user is permitted to define up to 10
        additional attributes.  If more are specified,
        an error is indicated.                          1b2b5a2

"DUMMY:"        $.ID ';     /                           1b2b6

    DUMMY  Dummy IDs are usually unparse rule names
    that are used only as node names and are never
    called as an unparse rule.  A compiler error
    results if the rule is ever called.                 1b2b6a

"OPCODES:"      $( .UID '= .NUM ) ';  /                 1b2b7

    OPCODES   defines a set of UIDs as opcodes which
    have not been already pre-defined in Tree Meta
    (the compiler being used to compile this program).
    They will be used in the text of this program.
    The declared opcodes are placed into the meta
    compiler's symbol table (to recognize ops in this
    program), not in the symbol table of the compiler
    being created.  They will not be passed to DDT.     1b2b7a

        When used in the formation of output
        instruction words, UIDs set as opcodes in the
        header are OR'd into the word being formed.     1b2b7a1

        A list of predefined opcodes for the 10 system
        appears in Appendix --.                         1b2b7a2

"SET:"          $( .UID '= .NUM ) ';  /                 1b2b8

    SET  Set IDs have their values preset to the
    indicated numbers in the meta-compiler's symbol
    table.                                              1b2b8a

        They are very much like opcodes except for the
        way in which they are treated when they are
        used to form output: they are added into the
        output word being formed.                       1b2b8a1
```

Set IDs are primarily used to specify register cells or constants. 1b2b8a2

"FIELDS:" $(.UID "= [" .NUM ': .NUM ']) '; ; 1b2b9

FIELDS A field is an upper case identifier which designates a group of contiguous bits in output words being formed in a manner described in the section on output generation below. The number on the left of the colon in the definition indicates the number of bits in the field; the number on the right indicates the number of bits to the right of the field. 1b2b9a

If a field is specified on a value in an output word being formed (see pvalue below), any previous contents of the field are replaced by the specified value. If the value is a number of bits greater than N where N is the size of the field, the rightmost N bits of the value will be placed in the output word field. 1b2b9a1

Rules 1b3

rule = synrul / unprul ; 1b3a

There are two types of rules, syntax (parse) rules and unparse rules. 1b3a1

synrul = .ID '= exp '; ; 1b3b

The ID to the left of the equal sign is the name of the syntax rule. The syntax expression (exp) to the right is responsible for controling the parsing of the input through various tests and alternations, the building of nodes of the tree through stack manipulations, and the passing of control to other parse rules and to unparse or output rules. Each syntax rule ends with a semi-colon. 1b3b1

unprul = 1b3c

There are three types of unparse rules. 1b3c1

```
.ID  #( #nodtst  "=>"  genexp '; ) /
.ID  #( #nodtst  ":="  valexp '; ) /
```
1b3c2

In the more elaborate types of unparse rule the name of the rule, the ID, is followed by one or more tree testing parts (nodtst) each followed by either a "=>" symbol followed by an output generation expression (genexp) or a ":=" followed by a value expression (valexp) closed by a semi-colon. The genexp permits (among other things) more testing and alternations, passing of control to other unparse rules, and generation of output code. 1b3c2a

The symbol ":=" following the nodtst indicates that the rule will return a value. At some point during its execution, the value expression must create a value. If more than one value is present in the valexp, the last one is returned. 1b3c2a1

A valexp is similar to a genexp; it does not, however, permit controls or output generation. Only testing, alternations, and value assignments are permitted. 1b3c2a2

The unparse rules with "=>" symbols do not return values. 1b3c2a3

One may not mix modes that assign values with modes that do not assign values in one unparse rule. 1b3c2a4

.ID "=>" genexp '; ; 1b3c3

The more simple form of unparse rule does not permit node testing and is used when no subnode testing is required. 1b3c3a

Syntax expressions and subexpressions 1b4

exp = # <'/>subexp; 1b4a

An expression (exp) is the right part of a syntax rule. It contains any number of subexpressions (subexps) separated by slashes. The subexps are alternate possibilities to be read from from left to right. If all fail, the rule as a whole fails. If one succeeds the rule succeeds, and only the successful subexp is executed. 1b4a1

subexp = ['←] #(stest / ntest); 1b4b

A subexp with no leading "←" is a series of syntax testing (stest) or non-testing (ntest) elements. If the subexp begins with a "←", backup will be invoked within the subexp if any of its elements fail. 1b4b1

If the first subexp element is an stest in a rule in which backup has not been invoked and it fails, execution will pass to the next alternation in the exp if the alternations have not all been exhausted. In any other case in which backup is not used, if a failing stest is not the first element a syntax error will result. 1b4b2

If backup has been invoked within a subexp such a failure will cause the subexp to fail and everything (including the input marker position but excluding any output which may have been produced) will be restored to the condition present upon entry into the subexp. Then the next alternation in the exp will be tried (or the rule will return false if all alternatives have been exhausted). Following the "←" is a series of syntax testing (stest) and non-testing (ntest) elements. 1b4b3

It is not wise to do any unparsing inside expressions in which backup has been invoked because the output generated is not backed up. 1b4b3a

Testing syntax elements 1b5

stest = 1b5a

These syntax elements may fail. 1b5a1

'. uid / 1b5a2

The '. uid represents a call on a basic recognizer. If successful, the recognizer will add a reference to the recognized symbol onto the K-stack. 1b5a2a

.ID / 1b5a3

An identifier indicates a call on another parse

rule. The called rule may add new items to the K-stack. 1b5a3a

```
.SR       /
.SR1      /                                                    1b5a4
```

The two types of strings (SR, SR1) test the input stream for the given string. They fail if that string is not the very next thing to be read. An SR string may be of arbitrary length. Nothing on the K-stack is changed. 1b5a4a

```
'( exp ') /                                                    1b5a5
```

Parentheses may be used to nest expressions. 1b5a5a

The parenthesized expression is taken to be a single stest entity no matter how complicated it may be. 1b5a5a1

```
"?F" .ID  /                                                    1b5a6
```

The ?F fails if the flag specified by the ID is false. 1b5a6a

```
"?@" .ID  /                                                    1b5a7
```

The ?@ fails if the indicated attribute of the most recently recognized symbol is not set. 1b5a7a

```
( "+'" / "-'" ) .CHR;                                          1b5a8
```

A plus or minus sign followed by a single quote followed by a character checks for the presence or absence of the given character as the next item in the input stream without advancing the input marker. 1b5a8a

This thus serves effectively as a one character lookahead; if the test fails, back-up is not necessary to restore the input stream. 1b5a8a1

This type of test is extremely fast in execution. 1b5a8a2

```
uid       = "UID" / "ID" / "NUM" / "SR" / "SR1" / "CHR" ;      1b5b
```

These are the basic identifiers. They were described in the Overview section of this report. 1b5b1

Non-testing elements 1b6

ntest = 1b6a

These syntax elements cannot fail. 1b6a1

': nodcon / 1b6a2

A parse rule node construction element (nodcon) following a colon builds a tree node. It replaces a specified number of items on the K-stack with a reference to a newly built node and its structure. The N-stack will contain the new node and its subnodes. 1b6a2a

'* / 1b6a3

An asterisk in this context passes control to the unparse rule at the top node or root of the tree (really a pointer at the top of the K-stack that refers to a node and its subnode chain on the N-stack). The N-stack is marked and a call made to the proper rule. Upon a successful return the branch of the tree is removed from the N- and K-stacks. A failing return results in a compiler error. (See the Program Environment section for a more detailed explanation.) 1b6a3a

".CHR" / 1b6a4

The ".CHR" reads one character from the input and adds an item to the K-stack but cannot fail. 1b6a4a

'[exp '] / 1b6a5

Expressions inside brackets are optional. For example, if they contain an stest which fails, the entire subexp does not necessarily fail. 1b6a5a

contrl / 1b6a6

Controls (contrls) perform various operations involved in the control of a compilation by the

user's compiler. They are discussed in detail below. 1b6a6a

scn / 1b6a7

Discussed below. 1b6a7a

":[" genexp "] / 1b6a8

A colon followed by an output generation expression in square brackets passes immediate control to the enclosed genexp. Control by the genexp permits outputting and other unparse rule operations directly from parse rules and is useful for simple parts of the compilation in which it would be cumbersome and unnecessary to build a parse tree node and then immediately unparse it. 1b6a8a

' opcd instr / 1b6a9

This construction permits insertion of assembly code instructions in the compiler. The opcd is an uppercase opcode which was either declared in the header or was present in the initialization file. The rest of the instruction (instr) appears in the section on Assembly Code below. The assembly code feature has been used so infrequently that at present it is only optionally present in the meta-compiler. 1b6a9a

' idseq "(exp ") / 1b6a10

This construction permits the identification of local variables. The ID sequence (idseq) is a list of IDs (flags and variables but not attributes) separated by commas whose values are saved before execution of the exp enclosed by the parentheses. Upon completion of the execution (during which the values of these IDs may have been changed), the original saved values are restored. Recursive identification of local variables is permitted. 1b6a10a

'& ("MARK" / "UNMARK") / 1b6a11

The "&MARK" and "&UNMARK" cause the N-stack to be bracketed. In effect, the portion of a tree built

up to the mark is saved. After unparse execution the tree is collapsed to the mark. Another tree may then be built and unparsed while keeping trees previously built . This bracketing may be nested permitting the "*" process of unparsing a tree to be nested. 1b6a11a

"&LABEL" / 1b6a12

The "&LABEL" construction causes the library to use the most recently recognized ID in all error messages as a reference for the user to a point in the input program. 1b6a12a

These messages are written loc+n where n lines have been read since the symbol loc was designated by an "&LABEL" command in the compiler. 1b6a12a1

"&DISCARD" / 1b6a13

The "&DISCARD" construction removes the top item on the K-stack and is used to throw away an unwanted UID, ID, etc. 1b6a13a

('$ / '# / ".$" / ".#") ['< exp '>] stest ; 1b6a14

The dollar sign, "$", is the arbitrary number or sequence operator. The single stest following the optional expression enclosed in angle brackets may occur an arbitrary number of times (including zero). Parentheses may be used to group a set of elements into a single element. 1b6a14a

The number sign, "#", is used before elements which may occur any number of times equal to or greater than one. 1b6a14b

If "$" or "#" preceeds an expression enclosed by angle brackets, a series of the stest items (including zero for "$", at least one for "#") separated by the enclosed expression must occur. The final element in the series must be one of the items, not one of the bracketed delimiters. 1b6a14c

If a period occurs before the "$" or "#" the K-stack will be marked before the start of the

recognition of the arbitrarily long series. This requires that the "$" construction be used to build a sequence of subnodes at some subsequent point. All items recognized after the mark from the ".$" or ".#" will be included in the sequence. (See nodcon below.) 1b6a14c1

If some other specification of the subnodes is made there may be trouble in removing the marker (which is a pointer to the first cell in the sequence) from the K-stack. 1b6a14c1a

scn = "->" stest; 1b6b

The scan construction causes the input stream to be advanced to the first occurrence of the given stest. It is used in the error construction and is invoked after syntax errors. 1b6b1

opcd = .UID; 1b6c

The UID must have been declared an opcode in the header or in the initialization file. 1b6c1

idseq = # <',>.ID ; 1b6d

This is a series of IDs (flags or variables but not attributes) separated by commas. 1b6d1

Node construction and execution in parse rules 1b7

nodcon = 1b7a

A parse rule node construction element (nodcon) following a colon builds a tree node; as a node element (see below) it creates a sub-node on a node. 1b7a1

.ID ['[$ <',>nodel1 ']] / 1b7a2

The given ID will be the name of the generated node; the optional parse mode node elements (nodel1) separated by commas inside the brackets will be the sub-nodes emanating from it. Building a node may involve removing items from the K-stack to produce it and will always involve adding one item to the top of the K-stack -- a pointer or

reference to the node just built which will reside with its sub-nodes on the N-stack. 1b7a2a

'= value / 1b7a3

The equals-value is used to build a node or sub-node of any 36-bit value. 1b7a3a

opcd / 1b7a4

If an UID opcode appears in this context, the value of the opcode will be added as a node. The opcode must be defined. 1b7a4a

glabel / 1b7a5

A generated label (glabel) will appear as a node or sub-node on the tree if it occurs in this context. It is explained more fully below. 1b7a5a

.SR / 1b7a6

A string as a node building element enters the specified string into symbol storage and uses it as a node as if it were recognized from the input stream by SR. 1b7a6a

.SR1 ; 1b7a7

SR1 builds a single character sub-node as though it were recognized from the input stream by SR1 or CHR. 1b7a7a

node11 = 1b7b

These node elements will be subnodes of the node being constructed. 1b7b1

.NUM / 1b7b2

If a number occurs as a node element, that number of items will be removed from the K-stack and used as subnodes of the node being generated. They will appear in the order of recognition. 1b7b2a

'$ / 1b7b3

A dollar sign in this context indicates that a sequence of items recognized from the input with the marked sequence operators, ".$" or ".#", is to be built into the node in the order recognized. If any items are recognized after the sequence operator, then the "$" node building construction will include them as if they were in the sequence. 1b7b3a

nodref / 1b7b4

In this context the node reference (nodref) permits direct access of K-stack items by the user. The number in the nodref will indicate the order of recognition of the item starting with the first item recognized at the start of the parse rule as 1, the second as 2, and so on. Access beyond the start of a particular parse rule is undefined. 1b7b4a

When used here the K-stack item is not popped--the "&DISCARD" construction must be used to remove items from the stack. It removes items from the top of the stack so be careful. (See Controls below.) 1b7b4a1

Node references which appear in the more usual context of unparse rules are described below. 1b7b4a2

nodcon ; 1b7b5

A nodcon here is used to create (possibly elaborate) subnode structure on the node being generated. 1b7b5a

glabel = '# gref / "G#" gref ; 1b7c

A generated label (glabel) indicates that a generated label should be added as a sub-node. The "G" is used to force a new label to be generated. If a label has not been previously referenced and is referenced in the glabel construction without the presence of the "G", it is generated as if the "G" were there. 1b7c1

gref = '1 / '2 / .ID ; 1b7d

The only label numbers are 1 and 2. By regenerating labels any number of labels may be used by one rule,

but only two may be referenced at once. The labels 1 and 2 remain local to the rule. Provision is made for assigning a generated label to an ID (see asgn in Controls below). Assignment of generated labels to IDs allows global generated labels. 1b7d1

Controls 1b8

Controls alter the flow of the meta-compiler, give information to the loader, define and assign values to variables, flags and attributes, and send messages to the user. 1b8a

contrl = 1b8b

'& 'F ('S/'R) idseq / 1b8b1

The constructions "&FS" and "&FR" respectively set and reset the flags declared in the header and specified by the idseq. 1b8b1a

'@ ('S/'R) idseq [nodref] / 1b8b2

The attribute set and reset constructions ("@S" and "@R" respectively) set or reset the attributes, declared in the header and specified by the idseq, on the most recently recognized K-stack item in a parse rule without a nodref, on the K-stack item specified by the nodref in a parse rule, or on the node item specified by the nodref in an unparse rule. 1b8b2a

Note that the nodref is legitimately optional in a parse rule, but is manditory in an unparse rule. The nodref must be a terminal node in an unparse rule. A test is made to check whether or not a nodref is a terminal node if the check flag, which was set in the header by the inclusion of the word "CHECK", is on. 1b8b2a1

'& call / 1b8b3

Permits a call on a library routine. 1b8b3a

'& asgn / 1b8b4

Permits assignment to variables. 1b8b4a

"&NAME" [nodref] / 1b8b5

This construction is used to generate the name of the output file to be used by the loader on the PDP-10. If the optional nodref is absent, the most recently recognized ID is used as the title name; if it is present, the terminal node (in an unparse rule) or the K-stack item (in a parse rule) to which it refers is used. 1b8b5a

This was used to implement the "FILE" ID construction in the meta-compiler. 1b8b5a1

"&BSS" value ', / 1b8b6

This increments the location counter by the given value and sends a control word to the loader to leave a block of zeros when loading. 1b8b6a

"&TABLES" / 1b8b7

This calls on the library routine which outputs the symbol tables and dumps the literal storage table for the loader. (Symbols with the noddt attribute set are not passed to the loader. They include opcodes set in the header as well as numbers.) The symbol table required by DDT is described in the Program Environment section. 1b8b7a

"&FAIL" / 1b8b8

"&FAIL" forces a rule failure. It may be used anywhere in a parse rule; in an unparse rule it may not occur in the middle of a "$" iteration described in outunit below. 1b8b8a

"TRUE" / 1b8b9

"TRUE", legal only in unparse rules, forces success of a rule. It is effectively the "empty" rule. 1b8b9a

"&G#" gref / 1b8b10

Forces generation of the label referred to by the generated label reference (identical to the action

of a glabel in a parse rule except a tree node is not generated.) 1b8b10a

'> def / 1b8b11

Definitions (def) define values in identifiers and subnodes. See the Defintion section below. 1b8b11a

'< $mesgelt '> ; 1b8b12

The message elements (mesgelts) between angle brackets are all written on the message file. This construct is handy for communicating with the user; error messages, warnings, or values of parameters within the compiler can be written out. 1b8b12a

call = .ID; 1b8c

The ID is the name of the library routine called in the '& call construction. 1b8c1

asgn = .ID '← (value / glabel); 1b8d

The ID is the name of the variable to which the value or generated label is assigned in the '& asgn construction. If a generated label is assigned to an ID one can use global generated labels. If it has not been declared in a def, the named cell is usually declared in a dummy heading statement. 1b8d1

mesgelt = 1b8e

.SR / 1b8e1

A string as a mesgelt causes the given string to be written on the message file. 1b8e1a

"LOC" / 1b8e2

The symbol "LOC" causes the location in the input program to be written out in the form of a label plus the number of lines past that label. The "&LABEL" construct is used to designate the labels at the time they are read from the input program. 1b8e2a

"LINE" / 1b8e3

The symbol "LINE" causes the line number past the label set by the last "&LABEL" to be written. 1b8e3a

```
nodref      /                                          1b8e4
```

A nodref is used to write out a symbol in the tree from an unparse rule or on the K-stack from an parse rule. 1b8e4a

```
value ['B];                                            1b8e5
```

A value without the optional "B" causes a number to be written out to the base ten. This can be used to write compiler parameters at the end of compilation. With a "B" an octal number is written. 1b8e5a

Definitions 1b9

```
def       = ['↑ def1 [ '← value ];
def1      = nodref / .ID / .SR / glabel;               1b9a
```

Definitions permit the assignment of values to string storage items specified by the definition item (def1). The optional "↑" causes the external attribute of the item to to be set. At load time these symbols may be used to satisfy undefined symbol references from within other files. A flag is set permitting the loader to fix up the necessary links between the references in the various files. Links of this type are used in Tree Meta to create links between items referenced in both the Tree Meta program and its library routines. 1b9a1

If no value assignment part ("←" followed by a value) is present in the def, the item is given the current value of the location counter. The item may be a nodref in which case an immediate subnode (in an unparse rule) or a K-stack item (in a parse rule) is defined. If def1 is an ID or SR it is entered in symbol storage if not already present. Glabels are always defined to the current location; their external attributes may not be set and hence an "↑" is not permitted. 1b9a2

Output rules 1b10

unprul = 1b10a

.ID "=>" genexp '; / 1b10a1

.ID #(#nodtst "=>" genexp ';) / 1b10a2

.ID #(#nodtst ":=" valexp ';) ; 1b10a3

As noted above the ID is the name of the unparse rule. 1b10a4

Control passes to unparse rules from parse rules when an asterisk occurs. 1b10a4a

Control passes from one parse rule to another when, as we shall see, the name of the invoked rule occurs as a referenced subnode or is constructed in the calling rule. 1b10a4b

Nodtsts appearing after the unparse rule names permit the testing of the substructure of the node. 1b10a5

The nodtsts are alternatives; each is tried in order until one succeeds. The output rule fails if none of the nodtsts succeed. Once one is successful, the corresponding output generation expression (genexp) or value assignment expression (valexp) is executed. At this level only this alternative of the unprul is executed. The genexp permits more alternations and testing and may in fact fail completely. The valexp also permits alternations and testing and may also fail. In such a case a compiler error is generated. 1b10a5a

Node testing 1b11

nodtst = 1b11a

A nodtst appearing after the name of an unparse rule tests the substructure of the named branch. 1b11a1

"[$]" / 1b11a2

A dollar sign indicates that no test is to be made-- any number of nodes of any type may follow as immediate subnodes of the named branch. 1b11a2a

'[$ <',>item '] ; 1b11a3

The nodtst may be a series of nodtst items separated by commas. The position in the list indicates which immediate subnodes are being checked. If no items are in the list there must be zero subnodes. 1b11a3a

item = 1b11b

Nodtst items represent tests made on immediate subnodes or groups of subnodes. The position of the item in the nodtst determines which subnodes are tested. 1b11b1

'- / 1b11b2

The dash checks for the existence of a subnode in the position indicated by the position of the dash in the item list. 1b11b2a

'. udtst / 1b11b3

The period udtst item succeeds if the subnode is a symbol recognized by the indicated basic recognizer. See below. 1b11b3a

.NUM / 1b11b4

A number treats the subnode as a 36-bit number and checks for equality with the given number. 1b11b4a

opcd / 1b11b5

An opcd is similar to a number item. The subnode is tested for equality with the the value of the given opcode. 1b11b5a

'# gref / 1b11b6

A generated label item tests for a label and copies it into the label given. The label may then be used in the genexp part. 1b11b6a

(.ID / '?) $nodtst / 1b11b7

The ID with an optional nodtst construction tests

for a subnode with that name and allows for testing the substructure of that node. The sequence of nodtsts are alternatives which are tested in order until one succeeds. A question mark instead of the ID permits one to check the substructure of a subnode without specifying the name of the node. (Its position on the tree is, of course, specified by the position of the item in the nodtst.) 1b11b7a

nodref / 1b11b8

A nodref permits one to check the presence of a particular substructure or subnode specified by the node reference. See nodref below. 1b11b8a

.SR ; 1b11b9

A string tests only to see that the subnode is a symbol, and is the same as the string given. It may gave been recognized by UID, ID or SR. 1b11b9a

udtst = 1b11c

"UID" / "ID" / "SR" / "SR1" / "CHR" / "LABL" ; 1b11c1

These are the basic recognizers permitted in the '. udtst item. Note that "NUM" is not among them because all nodes are numbers. (One could either use a dash to check for the existence of the node or a number to check for the value on the node.) The "LABL" permits checking for the presence of a generated label on the node without copying the label over to permit its use in the genexp. 1b11c1a

Output expressions 1b12

genexp = 1b12a

#(otest / genelt) $('/ #(otest / genelt)) ; 1b12a1

Output generation expressions (genexp) allow another level of alternation. Alternatives are made up of any number of output test elements (otests) and output generation and control elements (genelts). Several alternatives may be

provided in a genexp. Just as in a syntax expression, they are separated by slashes. 1b12a1a

A genexp is executed in an output rule only if the corresponding nodtst had succeeded. Failure is determined just as for syntax alternatives without backup. If the first element in an alternative is an otest which fails, execution will pass to the next alternative if all have not been exhausted. If a failing otest is not the first element in the alternative, or if all alternatives have been exhausted, a compiler error will result and the error recovery rule specified in the header will be invoked. 1b12a1a1

Genexps may appear in parse rules after a colon enclosed by square brackets. This possibility is often useful for performing unparse rule operations directly from parse rules in simple parts of the compilation in which it would be cumbersome and unnecessary to build a parse tree branch and then immediately unparse it. In this case node references refer to the K-stack rather than to the tree. (See the Node Reference section below.) 1b12a1a2

Value rule expressions 1b13

valexp = 1b13a

#(otest / value) $('/ #(otest / value)) ; 1b13a1

Value expressions are used in value rules; they operate in a manner almost identical to output expressions. Controls and output generation are not permitted, however. Alternations and failure are the same as for genexps. The last value created in a successful alternative is the value returned. 1b13a1a

Output tests 1b14

otest = # <"?AND">otest1 ; 1b14a

Output tests (otests) permit further testing of the tree structure and flag and variable values in order

to facilitate alternation in the genexp. An otest may be a conjunction of any number of output test elements (otest1) joined by the "?AND" construction; if one otest1 in the series fails, the entire otest fails. 1b14a1

otest1 = 1b14b

These make up the otests; they permit the building and testing of tree structure, the testing of flags and variables, and permit some control over the flow of the program. 1b14b1

nodout / 1b14b2

The node construction and execution in an unparse rule element (nodout) builds a node and invokes the named unparse rule. The called unparse rule works with the newly created tree node. The invoked rule may build more nodes to call more unparse rules. If it fails, the otest fails. Upon return, the node created for it is lost. 1b14b2a

nodref / 1b14b3

A nodref here usually denotes a non-terminal sub-node that is to be invoked. This type of unparse rule call fails if the called rule fails and returns false. 1b14b3a

"?F" .ID /
"?@" .ID nodref / 1b14b4

Flags and attributes may be tested just as in parse rules. Attributes require an additional sub-node reference with respect to the named branch or to a branch specified by an earlier nodref to indicate which symbol is to be tested. An error will result if "CHECK" was specified in the header and the nodref is not a terminal string-storage item. 1b14b4a

'(genexp ') / 1b14b5

Parentheses are used in the unparse rules just as in the parse rules to group and nest genexps. 1b14b5a

'? ["IF"] '& .ID / 1b14b6

This construction permits a normal nodtst inside a genexp or valexp, i.e., on the right of the "=>" or ":=" signs. 1b14b6a

'? ["IF"] "LAST" / 1b14b7

This construction succeeds if the node being considered (having been specified by iteration through a series of nodes by the dollar expression construction) was the last node recognized by one of the arbitrary number operators. See dolexp below. 1b14b7a

'? ["IF"] nodtst / 1b14b8

This construction succeeds if the node being considered has the structure sought by the nodtst as explained above. 1b14b8a

'? ["IF"] value ("">="/"<="/'>/'</'=/'#) value / 1b14b9

This construction is used to perform numerical tests on values which are described below. The usual binary relations are permitted in the tests. (The number sign means "not equal".) 1b14b9a

"?NOT" otest1 ; 1b14b10

This test succeeds if the specified otest1 fails. 1b14b10a

Node construction and execution in unparse rules 1b15

nodout = 1b15a

An unparse rule node construction and execution element (nodout) effectively builds a tree branch and immediately calls it into execution. Upon completion of execution it is discarded from the tree. 1b15a1

.ID '[$ <',>nodel2 '] ; 1b15a2

The given ID will be the name of the generated branch; the optional unparse node elements (nodel2) separated by commas inside square brackets will be subnodes emanating from it. 1b15a2a

The nodel2 elements correspond effectively to most of the nodel1 elements in parse rule node construction elements. The NUM and "$" modes are not present because the K-stack is never accessed in unparse rules-- it is used for pointer manipulation and is sacred. 1b15a2a1

If we are in a parse rule and the genexp is in square brackets following a colon, the nodref permits access to the K-stack as explained below. 1b15a2a2

nodel2 = 1b15b

These node elements will be subnodes of the node being generated in the unparse rule. 1b15b1

nodref / 1b15b2

A nodref here in an unparse rule means that the node referred to will become a node in the newly generated tree branch in the position indicated by the order of the nodref in the list of node elements. 1b15b2a

In a parse rule the nodref permits direct access of K-stack items. The number in the nodref will indicate the relative order of recognition of the item beginning with the first item recognized at the start of the parse rule as 1, the second as 2, and so on. Access beyond the start of a particular parse rule is undefined. 1b15b2a1

In parse rules one must remember to pop the items from the K-stack by using the "$DISCARD" construction. (See above.) 1b15b2a2

nodcon ; 1b15b3

The nodcon was discussed above. It is used to create substructure on the subnode being constructed. 1b15b3a

Output generation and control elements 1b16

genelt = outelt / contrl ; 1b16a

An output generation and control element (genelt) is either an output element responsible for producing instructions or a contrl for controlling the compilation. Contrls were discussed above. 1b16a1

outelt = 1b16b

Output elements are generally things that produce output code and literal cells. See the program environment section for a detailed explanation of code production. 1b16b1

outunit / 1b16b2

Output units (outunits) are the building blocks for the construction of instructions. All outunits through the first output word terminator (outerm) are used to form the instruction. If any instruction is not closed (terminated) before an unparse rule call is made (by a nodout or nodref), the rule called will continue to form the same word. In other words, the formation and closing of instructions is independent of calls and returns-- any number of unparse rules may contribute to the creation of an instruction. 1b16b2a

'= $outunit / 1b16b3

If an equal sign appears before a series of outunits, the cell being formed will be a literal specification. In the literal specification, all outunits after the equal sign through the first output word terminator (outerm) are used to form the literal cell. If any literal is not closed (terminated) before an unparse rule call is made (by a nodout or nodref), the rule called will continue to form the same literal. 1b16b3a

At the appearance of the output terminator both the literal cell and the instruction are closed. Hence the literal must be the last thing put in. The address becomes the address of the literal. 1b16b3a1

outerm ; 1b16b4

Output terminators (outerms) stop construction on

the current instruction and indicate the type of 2-bit loader control that accompanies the word. After the occurence of the outerm the word is appended to the binary output program and the single cell workspace initialized to zero for the generation of the next output word. 1b16b4a

outunit = 1b16c

Outunits are the building blocks of instructions and literal cells. All outunits through an outerm make up one cell. This process is independent of rule calling and failure: any number of rules may be involved in forming an instruction. For a detailed discussion of the generation of ouput see the Program Environment section. 1b16c1

otest1 / 1b16c2

An outunit may be an otest1. (If its presence is not in a literal specification there is no syntactic difference between this occurence of an otest1 and a single element in an otest.) Semantically, however, an interesting thing happens if the otest1 is a string storage item terminal node. In this case the value of the symbol is written out. If the nodref is another non-terminal node, the corresponding unparse rule is invoked as above. Other types of terminal symbol nodrefs (e.g., glabels) will result in compiler errors. 1b16c2a

'[genexp '] / 1b16c3

Genexps which are optional are enclosed in brackets just as optional exps were in parse rules. 1b16c3a

dolexp / 1b16c4

The dollar expression (dolexp) is explained below. 1b16c4a

glabel / 1b16c5

A glabel as an outunit stores the value of the specified label in the address field of the output word being created. The label is generated if, at

the time of reference, it has not been referenced before. 1b16c5a

.SR / 1b16c6

The specified string is written out on the binary output 5 characters per word with a null character appended to the end of the string to satisfy conventions of the TENEX time-sharing system. If an instruction word has been partially formed it is not affected, and work on it may continue after the string has been put out. 1b16c6a

"*S" nodnm / 1b16c7

This construction assumes a pointer to a string on the node referred to by the node name (nodnm) described in the Node Reference section below. This string is appended to the output file 5 characters to a word with a null character appended to the end of the string. As above, partially formed instruction words are not affected by the output of a string. They will be put out, upon termination, after the string. 1b16c7a

This construction appears here rather than under pvalue below with similar constructions because those constructions all refer to ouput units which would fit in one word; a string could, however, be many words in length. 1b16c7a1

' opcd instr / 1b16c8

Used for inserting hand coded assembly language instructions as in parse rules. 1b16c8a

' idseq '(genexp ') / 1b16c9

This construction saves local variables as did the corresponding construction in parse rules. The values of the flags and variables specified by the list of IDs are saved while the parenthesized genexp is executed. The values of the flags and variables are restored upon completion of the execution. Recursive saves are permitted by this construction. 1b16c9a

pvalue ; 1b16c10

A primitive value (pvalue) permits the specification of words or parts of words and allows some simple arithmetic operations on these entities before being placed in the output word being formed. Pvalue is responsible for determining whether a value is relocatable or absolute. This information is used at instruction termination time for the generation of control bits. 1b16c10a

dolexp = '$ [nodref / "SYMS"] ['←] '(genexp ') ; 1b16d

The dollar expression construction (dolexp) is used to iterate over the genexp enclosed in parentheses and is the unparse construction that corresponds to the arbitrary number operator for parse rules. 1b16d1

If only the "$" appears at the beginning of the construction, the parenthesized genexp is executed once for each subnode of the current node. If another node is specified by the nodref part, execution takes place once for each of its subnodes. If "SYMS" follows the dollar sign, the iteration is over all symbols in symbol storage. 1b16d2

Inside the genexp, a "$" in a nodref is interpreted as "the current node of this iteration," that is *$ is *1 the first time through, *2 the second, etc., if no nodref was made after the opening dollar sign of the construction. If a nodref *2 appeared after the opening dollar sign, a dollar sign in a nodref in the genexp would refer to the node *2:1 the first time through, *2:2 the second, etc. 1b16d2a

Note that a "FAIL" may not occur in the genexp in this context. 1b16d2b

The optional backarrow means that the iteration will take place in reverse order. 1b16d3

outerm = 1b16e

Output terminators determine the type of loader control (2 bit code word) that accompanies the

instruction. They terminate construction on a particular output cell, append the cell to the output file, and initialize the working cell to zero in order to construct the next output word. A detailed disscussion appears in the Program Environment section below. 1b16e1

', / 1b16e2

A "," means the cell will be absolute unless the instruction contains a pvalue that is relocatable. Then it means the output word will also be relocatable. 1b16e2a

' .NUM ; 1b16e3

A slash-number gives the programmer the opportunity to specify the 2 bit code word himself. 1b16e3a

The code words for the PDP-10 are the following: 1b16e3b

0-- absolute.
1-- relocatable right half.
2-- relocatable left half.
3-- both halves relocatable. 1b16e3b1

pvalue = 1b16f

['+ / '-] value ['↑ fld] ; 1b16f1

Any value may be added or placed into the output word being formed in a manner dependent on the type of value. 1b16f1a

A value followed by a field specification, that is, an up-arrow ("↑") followed by a fld, will cause the specified field in the output word to be replaced by the associated value. 1b16f1a1

A value that is an opcode will be ORd into the opcode field of the output word. 1b16f1a2

All other values will be added into the output word. A minus sign subtracts the value from the word or places the negative of the value in

the field. A negative opcode has the same effect as a positve opcode. 1b16f1a3

CAUTION: Pvalue determines whether or not the output word is relocatable and sets a flag used to determine the loader control word when the output terminator appears to close an output word. A pvalue may be a value expression which, as we shall see below, may involve a plus sign. In spite of the fact that each pvalue without specified fields through the outerm is added into the output word, the value of the loader control word depends on the form of the expression. 1b16f1b

If, for example, the output word is formed from the series of pvalues and outerm ". 2 ," the word will be the current value of the location counter plus 2 and will be relocatable because "." is relocatable. On the other hand, if the word is formed from ".+2 ," the word will be absolute because value will calculate the result of the arithmetic expression. Take care in the specification of output words. 1b16f1b1

fld = .UID; 1b16g

A field specification (fld) is an UID which was declared in the header to represent a field in an output instruction word. 1b16g1

Node references 1b17

nodref = '* nodnm ; 1b17a

A node reference (nodref) in an unparse rule refers to a tree node relative to the current node. It begins with an asterisk and is followed by a node name (nodnm) specification. 1b17a1

In a parse rule the nodref refers to the K-stack. 1b17a2

nodnm = 1b17b

A nodnm specifies the node or K-stack in a nodref and related constructions (for example, "*S" and "*V"), 1b17b1

(.NUM / '$) $(': .NUM) / 1b17b2

The NUM - colon - NUM chain permits the specification of subnodes of the current node. Each colon - NUM indicates a subnode of the node already specified. (Thus *1:2 is the second node under the first subnode.) 1b17b2a

The dollar sign appears as the first element in such a chain in a genexp over which an iteration is beng done. The dollar sign represents the current node in the iteration. (See the explanation of the dollar iteration mode above.) 1b17b2a1

In a parse rule the nodnm refers to elements put on the K-stack by the current parse rule and the rules it has called. The first level numbers correspond to the order in which the K-stack elements were placed on the stack-- one for the first, two for the second. Since tree branches may have been created in the parse rule, sub-structure references may still be present. 1b17b2b

Such access to the K-stack does not cause elements to be popped off the stack. The "&DISCARD" construction serves for this purpose. 1b17b2b1

.ID / 1b17b3

An ID refers to the contents of the global variable specified by the ID. This construction is usually used after storing the hash number of a string storage item in a global cell. 1b17b3a

'' .ID '' : 1b17b4

The ID is entered in the string storage area and is referred to as if it were a node. 1b17b4a

Values 1b18

value = term $('+ value / '- value) ; 1b18a

A value is an arithmetic expression made up of a series of terms separated by the operators "+" or "-" indicating addition and subtraction respectively. The result is a single 36-bit value. 1b18a1

Tree Meta - FORMAL DESCRIPTION--
Syntax and semantics

```
term     = prim $( ";*" prim / ";/" prim ) ;                    1b18b
```

A term is a series of primitives (prim) separated by the operators ";*" and ";/" indicationg multiplication and division respectively. These operators thus have higher precedence in an arithmetic value expression than the addition and subtraction operators. 1b18b1

```
prim     =                                                      1b18c
```

A prim is the most primitive level of value. 1b18c1

```
.NUM                          /                                 1b18c2
```

A number is just the value of the number. 1b18c2a

```
opcd                          /                                 1b18c3
```

An opcd is the value of an UID declared to be an opcode in the header or predefined in the meta compiler. 1b18c3a

```
setnm                         /                                 1b18c4
```

Setnm is an UID which was set to a particular value in the header. 1b18c4a

```
'* ['V/'L/'N/'C/'Q] nodnm     /                                 1b18c5
```

An asterisk-nodnm (which is only a nodref) must point to a terminal node (symbol table entry) and indicates the value of the symbol. 1b18c5a

The optional characters indicate: 1b18c5b

V the value of a symbol table entry 1b18c5b1

N the 36-bit value in the node 1b18c5b2

L the length of the symbol (assumes a string on the node) 1b18c5b3

C the character code in a single character node 1b18c5b4

Q the number of subnodes on this node 1b18c5b5

```
        '' .ID ''                       /                       1b18c6

           An ID in single quotes is entered in string
           storage if it is not already present.  The
           construction denotes the value of the ID symbol.
           It is similar to a reference of the form "*V1"
           except for the explicit specification of the
           symbol.                                              1b18c6a

        '.                              /                       1b18c7

           Point specifies the value of the location counter.
                                                                1b18c7a
        .ID [ '[ $ <',>node11 '] ]      /                       1b18c8

           An ID denotes a global symbol in the compiler
           program.  The optional node construction part
           assumes the ID is the name of a value rule.  It
           will be called with the indicated node structure
           and, upon return,  the returned value used as a
           prim.                                                1b18c8a

        "RSH("  value ') .NUM           /
        "LSH("  value ') .NUM           /
        "MASK(" value ') .NUM           /                       1b18c9

           Values may be shifted and masked by amounts
           indicated by the number to obtain new values.        1b18c9a

        "HASH(" nodref ')               /                       1b18c10

           The hash construction obtains the 10 bit hash
           number for a symbol.                                 1b18c10a

        '- prim ;                                               1b18c11

           The negative of a prim may be specified by a minus
           sign followed by the prim.                           1b18c11a

    setnm    = .UID;                                            1b18d

       Setnm is an UID which was set to a particular value
       in the header.                                           1b18d1

Assembly code                                                   1b19

   A limited assembly code is permitted.  Since it has been
```

used so infrequently, its compilation is at present optional, and its constructions will not be discussed here. 1b19a

```
instr    =                                                    1b19b
   [ regstr ', ] address ;                                    1b19b1
address  =                                                    1b19c
   [@] ('= adr / adr) [index] ;                               1b19c1
regstr   = .UID / .NUM ;                                      1b19d
index    = '( regstr ') ;                                     1b19e
adr      =                                                    1b19f
   .ID       /                                                1b19f1
   .SR1      /                                                1b19f2
   '- liter  /                                                1b19f3
   regstr ;                                                   1b19f4
liter    = .NUM ;                                             1b19g
```

1c

```
Compiler header                                                    1c1

   program  =                                                      1c1a
     "FILE" .ID ["CHECK"] "META" .ID  #headelt  #rule
     "END" /                                                       1c1a1
     "LIBRARY" $.ID ;                                              1c1a2
   headelt  =                                                      1c1b
     "FLAGS:"      $.ID ';  /                                      1c1b1
     "ATTRIBUTES:" $.ID ';  /                                      1c1b2
     "DUMMY:"      $.ID ';  /                                      1c1b3
     "OPCODES:"    $( .UID '= .NUM ) ';  /                         1c1b4
     "SET:"        $( .UID '= .NUM ) ';  /                         1c1b5
     "FIELDS:"     $( .UID "= [" .NUM ': .NUM '] ) ';  /           1c1b6
     "SIZE:"       $( ('S /'M /'K /'N /'G /'L) '=  .NUM )
     ';  /                                                         1c1b7
     "ERROR:"      scn exp '; ;

                                                                   1c1b8
Rules                                                              1c2

   rule     =                                                      1c2a
     synrul / unprul;                                              1c2a1
   synrul   =                                                      1c2b
     .ID '=  exp ';  ;                                             1c2b1
   unprul   =                                                      1c2c
     .ID              "=>"  genexp ';   /                          1c2c1
     .ID  #( #nodtst  "=>"  genexp '; ) /                          1c2c2
     .ID  #( #nodtst  ":="  valexp '; ) ;

                                                                   1c2c3
Syntax expressions and subexpressions                              1c3

   exp      =                                                      1c3a
     # <'/>subexp;                                                 1c3a1
   subexp   =                                                      1c3b
     ['←]  #( stest / ntest );

                                                                   1c3b1
Testing syntax elements                                            1c4

   stest    =                                                      1c4a
     '. uid    /                                                   1c4a1
     .ID       /                                                   1c4a2
     .SR       /                                                   1c4a3
     .SR1      /                                                   1c4a4
     '( exp ') /                                                   1c4a5
     "?F" .ID  /                                                   1c4a6
     "?@" .ID  /                                                   1c4a7
```

```
      ( "+" / "-" ) .CHR;                                          1c4a8
    uid      =                                                     1c4b
      "UID" / "ID" / "NUM" / "SR" / "SR1" / "CHR" ;

                                                                   1c4b1
  Non-testing elements
                                                                   1c5
    ntest    =                                                     1c5a
      ': nodcon                  /                                 1c5a1
      '*                         /                                 1c5a2
      ".CHR"                     /                                 1c5a3
      '[ exp ']                  /                                 1c5a4
      contrl                     /                                 1c5a5
      scn                        /                                 1c5a6
      ":[" genexp ']             /                                 1c5a7
      '   opcd instr             /                                 1c5a8
      '   idseq '( exp ')        /                                 1c5a9
      '& ("MARK" / "UNMARK" )    /                                 1c5a10
      "&LABEL"                   /                                 1c5a11
      "&DISCARD"                 /                                 1c5a12
      ( '$ / ".$" / '# / ".# ) ['< stest '>] stest ;                1c5a13
    scn      =                                                     1c5b
      "->" stest;                                                  1c5b1
    opcd     =                                                     1c5c
      .UID;                                                        1c5c1
    idseq    =                                                     1c5d
      # <',>.ID ;

                                                                   1c5d1
  Node construction and execution in parse rules
                                                                   1c6
    nodcon   =                                                     1c6a
      .ID [ '[ $ <',>nodel1 '] ] /                                 1c6a1
      '= value  /                                                  1c6a2
      opcd      /                                                  1c6a3
      glabel    /                                                  1c6a4
      .SR       /                                                  1c6a5
      .SR1      ;                                                  1c6a6
    nodel1   =                                                     1c6b
      .NUM    /                                                    1c6b1
      '$      /                                                    1c6b2
      nodref  /                                                    1c6b3
      nodcon ;                                                     1c6b4
    opcd     =                                                     1c6c
      .UID ;                                                       1c6c1
    glabel   =                                                     1c6d
      '# gref / "G#" gref ;                                        1c6d1
    gref     =                                                     1c6e
```

```
        '1 / '2 / .ID ;

                                                                    1c6e1
    Controls
                                                                      1c7
      contrl    =                                                   1c7a
        '$ 'F ('S/'R) idseq          /                              1c7a1
        '$ call                      /                              1c7a2
        '$ asgn                      /                              1c7a3
        '$ "G#" gref                 /                              1c7a4
        '$ "NAME" [nodref]           /                              1c7a5
        '$ "BSS" value ',            /                              1c7a6
        '$ "TABLES"                  /                              1c7a7
        '$ "FAIL"                    /                              1c7a8
        "TRUE"                       /                              1c7a9
        '> def                       /                             1c7a10
        '@ ('S/'R) idseq [nodref]    /                             1c7a11
        '< $mesgelt '> ;                                           1c7a12
      call      =                                                    1c7b
        .ID;                                                       1c7b1
      asgn      =                                                    1c7c
        .ID '← ( value / glabel );                                 1c7c1
      mesgelt   =                                                    1c7d
        .SR / "LOC" / "LINE" / nodref / value ['B];

                                                                   1c7d1
    Definitions
                                                                     1c8
      def       =                                                  1c8a
        ['↑] def1 [ '← value ];  % ↑ not permitted with
        glabel%                                                    1c8a1
      def1      =                                                  1c8b
        .ID / .SR / nodref / glabel;

                                                                   1c8b1
    Output rules
                                                                     1c9
      unprul    =                                                  1c9a
        .ID                "=>"  genexp ';    /                    1c9a1
        .ID  #( #nodtst    "=>"  genexp ';  ) /                    1c9a2
        .ID  #( #nodtst    ":="  valexp ';  ) ;

                                                                   1c9a3
    Node testing
                                                                    1c10
      nodtst    =                                                 1c10a
        "[$]" /  '[ $ <',>item '] ;                               1c10a1
      item      =                                                 1c10b
```

```
        '-                          /                                   1c10b1
        '. udtst                    /                                   1c10b2
        opcd                        /                                   1c10b3
        '# gref                     /                                   1c10b4
        (.ID / '?) $nodtst          /                                   1c10b5
        nodref                      /                                   1c10b6
        .NUM                        /                                   1c10b7
        .SR ;                                                           1c10b8
      udtst     =                                                       1c10c
        "UID" / "ID" / "SR" / "SR1" / "CHR" / "LABL" ;

                                                                        1c10c1
Output expressions
                                                                        1c11
      genexp    =                                                     1c11a
        #(otest / genelt) $( '/ #(otest / genelt) ) ;

                                                                        1c11a1
Value rule expressions
                                                                        1c12
    valexp    =                                                       1c12a
        #(otest / value) $( '/ #(otest / value) ) ;
                                                                        1c12a1
Output tests
                                                                        1c13
      otest     =                                                     1c13a
        # <"?AND">otest1 ;                                            1c13a1
      otest1    =                                                     1c13b
        nodout              /                                       1c13b1
        nodref              /                                       1c13b2
        "?F" .ID            /                                       1c13b3
        "?@" .ID nodref     /                                       1c13b4
        '( genexp ')        /                                       1c13b5
        '? ["IF"] '$ .ID    /                                       1c13b6
        '? ["IF"] "LAST"    /                                       1c13b7
        '? ["IF"] nodtst    /                                       1c13b8
        '? ["IF"] value (">="/"<="/'>/'</'=/'#) value /             1c13b9
        "?NOT" otest1       ;                                       1c13b10
Node construction and execution in unparse rules
                                                                        1c14
      nodout    =                                                     1c14a
        .ID '[ $ <',>nodel2 '] ;
                                                                        1c14a1
      nodel2    =                                                     1c14b
        nodref / nodcon ;

                                                                        1c14b1
```

```
Output generation and control elements                                  1c15

   genelt    =                                                          1c15a
     outelt / contrl ;                                                 1c15a1
   outelt    =                                                          1c15b
     outunit      /                                                    1c15b1
     '= $outunit /                                                     1c15b2
     outerm ;                                                          1c15b3
   outunit   =                                                          1c15c
     otest1                  /                                       1c15c1
     '[ genexp ']            /                                       1c15c2
     .SR                     /                                       1c15c3
     '$ dolexp               /                                       1c15c4
     glabel                  /                                       1c15c5
     '  opcd instr           /                                       1c15c6
     '  idseq '( genexp ') /                                         1c15c7
     "*S" nodnm              /                                       1c15c8
     pvalue ;                                                        1c15c9
   outerm    =                                                          1c15d
     ',   / '  .NUM ;                                                1c15d1
   dolexp    =                                                          1c15e
     [nodref / "SYMS"] ['←] '( genexp ') ;                           1c15e1
   pvalue    =                                                          1c15f
     ['+ / '-] value ['↑ fld] ;                                      1c15f1
   fld       =                                                          1c15g
     .UID;


                                                                       1c15g1
Node references                                                         1c16

   nodref    =                                                          1c16a
     '* nodnm ;                                                      1c16a1
   nodnm     =                                                          1c16b
     (.NUM / '$) $( ': .NUM) /                                       1c16b1
     .ID  /  .SR ;


                                                                       1c16b2
Values                                                                  1c17

   value     =                                                          1c17a
     term $( '+ value / '- value ) ;                                 1c17a1
   term      =                                                          1c17b
     prim $( ";*" prim / ";/" prim ) ;                               1c17b1
   prim      =                                                          1c17c
     .NUM                          /                                 1c17c1
     opcd                          /                                 1c17c2
     setnm                         /                                 1c17c3
     '* ['V/'L/'N/'C/'Q] nodnm     /                                 1c17c4
```

```
        '' .ID ''                          /                    1c17c5
        '.                                 /                    1c17c6
        .ID [ '[ $ <',>node11 '] ]         /                    1c17c7
        "RSH(" value ') .NUM               /                    1c17c8
        "LSH(" value ') .NUM               /                    1c17c9
        "MASK(" value ') .NUM              /                    1c17c10
        "HASH(" nodref ')                  /                    1c17c11
        '- prim ;                                               1c17c12
      setnm     =                                                 1c17d
        .UID;

                                                                  1c17d1
    Assembly code

                                                                    1c18
      instr     =                                                 1c18a
        [regstr ', ] address ;                                    1c18a1
      address   =                                                 1c18b
        [@] ('= adr / adr) [index] ;                              1c18b1
      regstr    =                                                 1c18c
        .UID / .NUM ;                                             1c18c1
      index     =                                                 1c18d
        '( regstr ') ;                                            1c18d1
      adr       =                                                 1c18e
        .ID         /                                             1c18e1
        .SR1        /                                             1c18e2
        '- liter    /                                             1c18e3
        regstr ;                                                  1c18e4
      liter     =                                                 1c18f
        .NUM ;                                                    1c18f1
```

Tree Meta Report -- Preliminary Report, Formal Description

(J14046) 23-JAN-73 14:05; Title: Author(s): Stanford Research Institute/&SRI-ARC ; Distribution: Auerbach, Marilyn F./MFA; Sub-Collections: SRI-ARC; Clerk: MFA;