ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM

Augmentation Resources Center

Bruce L. Parsley
Harvey G. Lehtman
Susan Kahn
This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-205 has been reviewed and is approved for publication.

APPROVED:  
RAYMOND A. LIUZZI  
Project Engineer

APPROVED:  
WENDALL C. BAUMAN, Colonel, USAF  
Chief, Information Sciences Division

FOR THE COMMANDER:  
JOHN P. HUSS  
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.
ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM

Bruce L. Parsley
Harvey G. Lehtman
Susan Kahn

Augmentation Resources Center
20705 Valley Green Drive
Cupertino CA 95014

Rome Air Development Center (ISIE)
Griffiss AFB NY 13441

UNCLASSIFIED

This report is composed of studies that have been conducted to develop the NLS system as an on-line programming environment and to provide an on-line JOVIAL interactive debugger with the capabilities to debug JOVIAL language programs. The final report contains several design additions to the NLS system to create an on-line programming environment. A JOVIAL User's Guide prepared in Addendum Technical Report I provides an extensive set of commands for using the JDAD Debugger. Addendum Technical Report II provides a generalized approach to debugging and describes the NLS/NSW Do-All Debugger (DAD).
EVALUATION

The work described in the final technical report and the addendum reports, one and two, represent a significant accomplishment in establishing parts of the framework of an on-line software programming environment.

The NLS system represents a significant programming tool that can be utilized to develop software programs. The high cost of developing software has been established in numerous studies. One way of reducing this cost is to develop standard programming environments. Parts of these programming environments will require sophisticated on-line debuggers. This effort establishes the feasibility of one such debugger for JOVIAL language programs.

The additions to NLS described in this effort also provide the types of capabilities that must be present in a standard software programming environment.

The result of this effort has been to extend the capability of tools needed in a software programming environment and establish procedures and methods for their implementation.

RAYMOND A. LIUZZI
Project Engineer
Augmentation Resources Center, Tymshare, Incorporated

Introduction

This report is submitted in fulfillment of Task/Technical Requirement 4.1.1 of the On-Line Programmers Management System project (KALC - Skl Contract Number F30602-77-C-0165; Skl - Tymshare Subcontract Number 14354). The Statement of Work said in part:

TASKS AND TECHNICAL REQUIREMENTS

4.1.1 Develop a list of additions and modifications to the NLS family of tools to create an on-line programming environment based on the IBM evaluation of NLS's programming environment, the IBM Structured Programming Series, and the contractor's own suggestions.

4.1.2 Upon receipt of written approval from the contracting officer, or of his authorized representative, implement the approved suggestions from the list.

4.1.3 Demonstrate a Jovial interactive debugger utilizing:

4.1.3.1 Language module (LM) and operating system module (CS) for PDP-10

4.1.3.2 Jovial compiler on PDP-10

4.1.3.3 ARPA Network

4.1.3.4 NLS/NSW Do-All Debugger (DAD)

REPORTING REQUIREMENTS

-- An interim technical report describing the results of the software engineering tools study, including an ordered list of recommended additions to NLS.

-- A final technical report.

This is the report mentioned immediately above.
Summary

As required by the Statement of Work, we have created a version of the ARC Do-All Debugger (DAD) for use with programs written in Jovial and which are compiled on the DEC PDP-10 running under the TENEX operating system. Attached is a user guide to this new Jovial DAD (JDAD).

In the Interim Technical Report for this project submitted on 17 January 1979 as ARC Journal document (46236,), we presented an ordered list of additions and modifications to NLS programming tools. Since then we have worked on and completed the following of those recommendations:

1. Enable NLS to start DAD and JDAD.
2. Encapsulate the Jovial compiler.
3. Document the encapsulation facility.
4. Generalize the PROGRAMS subsystem and its templates.

Additionally, we have created a detailed design for the following task:

5. An interactive, conditional, iterative Process system.

The first four of the tasks have resulted in systems and tools which were demonstrated, along with the JDAD debugger, to the technical monitor of the project at ARC in Cupertino during the week of 25 April 1979. These tasks are discussed in the following sections of this report. The last section of this report is a glossary of terms that may be unfamiliar to readers. It is recommended that any reader unfamiliar with NLS or TENEX terminology read the glossary before reading the following sections.
Detailed Discussion of Tasks Completed

Task 1: Enable NLS to start DAD and JDAD: A "Spliceable" DAD.

Introduction

A facility was added permitting the DAD and JDAD debuggers to "splice" themselves into the fork structure for an already executing program. Before this feature was available, DAD or JDAD (which, for simplicity, will be called [J]DAD in the following) had to be started at the Exec level; from the debugger, it was then necessary to start the programs to be debugged.

Under the new facility, a user would cause an executing program to be interrupted, e.g., via control-C, and issue a special command to the Exec. The command would result in [J]DAD being spliced into the proper place in the interrupted program's fork structure.

As an example of the benefits of such a facility, consider a non-programmer doing normal work in some program. An unexpected bug may be encountered. That user may then call in a programmer who could access [J]DAD to investigate the bug with most of the bug's context intact.

Tasks Performed to Create a Spliceable DAD or JDAD

The TENEX Exec was modified to create a new [J]DAD command which would splice in an instance of the [J]DAD program between the EXEC fork and its subsidiary forks (which presumably are running a program which one or more processes/forks.) After splicing in [J]DAD, execution would begin at a new entry point which would cause the new [J]DAD code described below to be executed.

The [J]DAD Dispatch Module (which performs bookkeeping operations on the internal [J]DAD data structures concerned with the processes being debugged) and the [J]DAD Operating System Module (which interfaces [J]DAD to specific operating systems, in the current case TENEX and TOPS-20) were modified so [J]DAD could begin execution at an alternate entry point. If started at this entry point, [J]DAD would query the operating system for information concerning processes in the fork structure underneath it and the particular states of those forks.

[J]DAD then establishes states of the forks and their
programs in its internal data structures comparable to the states which would exist if the program were initially executed under [J]DAD and the programmer had typed the control-L interrupt character to enter the debugger.

After its initial polling of fork status information and state determination and establishment, [J]DAD may be continued in its usual fashion to set breakpoints, examine and change code, and in general perform its usual debugging tasks.
Task 2: Encapsulate the JOVIAL compiler.

Once the JOVIAL compiler is encapsulated, JOVIAL source code programs that are in NLS files may be directly compiled with a variation of the PROGRAMS subsystem's Compile File command, without the user directly having to go through intermediate steps involving a sequential file (as is presently the case). Also, the encapsulation of the JOVIAL compiler will allow the use of the LIBRARY subsystem for semi-automatic JOVIAL compilations.

Under the strategy implemented, source code for JOVIAL programs is composed and edited in AUGMENT. The "Compile JOVIAL" command is then invoked creating a temporary sequential file (invisible to the user). The encapsulated JOVIAL compiler is executed, using this sequential file as input, and also using switches which may be set as options to the command. Object code is compiled to the specified location; error messages and diagnostics are entered into the desired locations.

The following is the syntax for the command (extracted from the Command Meta Language grammar of the demonstration system):

```
"COMPILE"
<"jovial program in file">
LSEL("GOLDFILE NAME") % the specification of the AUGMEN T JOVIAL source code file %
<"to">
LSEL("TEXT") % specification of the name of the object code file %
[CP110N <"enter switches">
  getswitches % any number of switches as outlined below may be specified %]
[CP110N <"compool filename"> LSEL("TEXT")]
CONFIRM
```

The following are valid switches:

```
("CROSSREF"/ "ACROSS"/ "SYNTAX"/ "INDENT"/ "RBC"/
"STATISTICS"/ "MACROCODE"/ "LCWSEG"/ "HISEG"/
"HOWESI"/ "LSEC"/ "HCPFI"/ "LSTICOPY"/ "NCSOURCE"/
"DEFINE"/ "NOTRACE"/ "NOINFORM"/ "NOWARNING"/ "HISSIGN"/ "NGEACK"/ "HIGHSTA1"/ "KA10"/ "K110"/ "MAGIC"/
"ASSEM")
```

On-Line Programmers Management System:
Final Technical Report
Encapsulation is a technique used to enable a process to control the execution environment of other processes. The controlling process does this by declaring that it wishes to trap selected system calls (JSYS's) when executed by other processes. When a monitored process attempts to execute a system call that will be trapped it is suspended and the monitoring process is notified. After gaining control the monitoring process may take any action it deems necessary. It may handle the call itself or it may allow the monitored process to perform the call or it may modify the arguments to the system call before it is handled.

This discussion here concerns taking a process which currently runs under TENEX or TOPS-20 and encapsulating it under AUGMENT. The purpose of encapsulation in this setting is to allow a user to run a process not currently provided directly in the AUGMENT environment, perhaps using AUGMENT files as input to the process and providing entry via AUGMENT-style commands.

There are three things that need to be done in AUGMENT in order to encapsulate a process designed to run under TENEX or TOPS-20:

1. A correspondence table specifying each JSYS to be trapped and the address of the procedure which will handle the trapped JSYS must be included as a global declaration. The table below specifies that the three JSYS's BOUT, GTIFN, and PEGUT will be trapped by MLs and they will be handled by procedures named EOUTHANDLER, GTJFHANDLER, and PEGOUTHANDLER respectively. The table must end with -1, -1).

```
$ jsys handling correspondence table $
DECLARE
jtraps = (4c2a1a
   $pbout, $pbouthandler,
   $bout, $bouthandler,
   $gtjfn, $gtjfnhndler,
```

On-Line Programmers Management System:
Final Technical Report
2. Write code to start up the specified process in an inferior fork.  

The steps to do this are:  

a.) First create the fork and load the process that is to run in the fork. This is done by first calling the backend procedure CRFORK with no arguments. CRFORK will return a fork handle number which must be saved for later procedure calls. The process is then loaded into the new fork's address space by a call on GTFKFILE. The arguments for GTFKFILE will be the fork handle number and the JFN of the save file to be loaded.  

b.) Define which JSYS's are to be trapped and what procedure is to handle the jsys when the interrupt occurs. This is done by calling the backend procedure DEFINETRAPS with the address of the jsys correspondence table as its argument. Which modifies two NLS tables: BITABLE indicates which JSYS's are to be trapped by setting the appropriate bit and DTABLE indicates the address of the procedure to be called when a particular jsys is trapped.  

c.) Call the backend procedure SETTHAPS with the fork handle of the process being encapsulated, a channel number over which interrupts will be transmitted to the controlling fork, and the desired interrupt level as arguments.  

d.) Save the terminal characteristics in case the encapsulated process changes them by calling the backend procedure SAVTERM. This procedure will return three results which must be saved for a later procedure call. The results are: the terminal mode word, terminal output control word one and terminal output control word two.  

e.) Wait for the inferior fork to finish by calling the backend procedure WAITFORK with the fork handle as its argument.
Tasks Completed

Task 3: Encapsulation Facility Documentation

f.) Call the backend procedure CLEANUP. This procedure will deactivate the channel connection, zero out EITABLE and DTABLE, kill the inferior fork, and release the JFN of the say file.

g.) Restore the terminal characteristics by calling the backend procedure RSTRTRM whose arguments will be the values returned by SAVTRM.

The procedure below is a typical example of how a process is encapsulated. All procedures mentioned can be found in the Encapsulator Module (nlsbesrc, encapsulator,).

EXAMPLE:

(runit) % run an encapsulated process %

PROCEDURE(savjfn);

% Procedure description

FUNCTION

This procedure is responsible for starting up and running the encapsulated process as well as cleaning things up when the process has terminated. Note that three globals are required:

chan - is the channel used in issuing the interrupt usually a number in the high twenties or low thirties.
ilev - interrupt it will usually be 3.
frkhandle - handle used to identify the encapsulated fork.

ARGUMENTS

savjfn - jfn of say file to be run in the new fork

RESULTS

none

NON-STANDARD CONTROL

none

% Declarations %

(trmmode);

(ctlwd1);

(ctlwd2);
Task 3: Encapsulation Facility Documentation

% define channel and interrupt level for
% jays trapping in fork to be created
chan_34;
ilev_3;

% guard against errors
INVOKE(catch);

% create new fork and enable all capabilities
frkhandle_orfork();

% get and load the .sav file to run in the
% new fork
getf(kfile(frkhandle, savjfn);

% define jays's to be trapped and set the
% traps for the new fork
definetraps($jtraps);
settraps(frkhandle, chan, ilev);

% save terminal characteristics
trmmode_savtrm(:ctlwd1, ctlwd2);

% wait until it finishes
waitfrk(frkhandle);

% cleanup
cleanup(frkhandle, savjfn, chan);

% drop the catchphrase
DROP(catch);

% Return
RETURN;

% catchphrase definition
(catch) CATCHPHASE();
BEGIN
CASE SIGNALTYPE OF
= aborttype :
BEGIN
DISABLE(catch);
cleanup(frkhandle, savjfn, chan);
END;
ENDCASE;
CONTINUE;
END;
END.

There are several examples of processes which terminate only with a <^C> which would return control to the operating system rather than AUGMENT. One way to handle this is to have a handler detect when the desired processing has completed. The handler will then set a global flag and leave the encapsulated process frozen until such time that
the monitoring process sees the flag. Instead of
waiting until the inferior fork finishes the moni-
toring procedure will have to look at the flag from
time to time. When the flag is seen the monitoring
routine can then kill the fork without creating any
problems.

3. Write the procedures that will handle the trapped
    jsys's.

The programmer will have to know something (per-
haps, a great deal) about the process he wishes to
capture in order to write the handlers. Input
and output are the primary concerns here. If the
process expects a command from the terminal the
handler may feed that command to the process.
Also, an AUGMENT file may be handed to the process
rather than a sequential file. The handler must
know what the encapsulated process is requesting
and how to satisfy that request. The handlers may
have to keep track of many different states in
order to do this successfully.

JSYS handlers are passed four arguments:

A fork handle
The current JSYS number
The address where the encapsulator saves the
current fork PC
The address of a 16 word block containing the
content of the fork's registers

JSYS handlers return two boolean values:

handled
TRUE if JSYS has been handled
FALSE let TENEX handle it
unfreeze
TRUE to unfreeze the fork
FALSE to leave it frozen

On-Line Programmers Management System:
Final Technical Report
Encapsulator Module

The Encapsulator Module is a collection of AUGMENT routines located in <nlslsrc,encapsulator> and are used to handle the low level coding necessary to perform an encapsulation.

It has three types of routines. One group of routines deals with starting up the encapsulation and cleaning up when it is finished:

a.) Create the new fork.

b.) Define and set the traps.

c.) Reset global tables when encapsulation is finished.

d.) Save and reset terminal characteristics.

Another group of routines are invoked by the interrupt mechanism. When an interrupt occurs LUMMYJSYSTRAFFPS1 is invoked and calls other routines in the Encapsulator Module to accomplish the following:

a.) Save the state of the superior fork.

b.) Dispatch the interrupt to the appropriate handler.

c.) Restore the state of the superior fork.

d.) Let the inferior fork continue.

e.) Continue the superior fork in the state existing when the interrupt occurred.

The remaining routines are designed to support writing in the inferior fork's accumulators and address space.

Encapsulator Subsystem

The encapsulator subsystem is designed to assist the programmer faced with an encapsulation project as well as serving as a tool in discovering the nature of any process. It allows the user to run the process as if it were running under the operating system while tracing all JSYS calls.
The subsystem has a single command which asks the user to specify an executable program file (e.g., a TENEX SAV file) that is to be encapsulated, what information is desired (a trace of all JSYS calls, a frequency count, or both) and how that information is to be presented (displayed at the terminal or recorded in a file). The information obtained consists of the identification of the JSYS trapped and its arguments. Actually, the contents of the first four registers are obtained regardless of the number of arguments expected for the jsys.

After the command is entered the specified process is encapsulated trapping each JSYS, obtaining the desired information about the JSYS and letting the operating system handle the call. Thus, the process will run as if it would if it was running directly under the operating system.

Currently, only JSYS's whose numbers are in the ranges 1 to 128, 14B to 315B, and 317B to 337B are trapped. The TIME, GJINF, and JOBIM JSYS's are not included because TENEX does not execute them properly after they have been trapped. These JSYS's should never be trapped in any encapsulation. There may be others that cause problems and it would be useful if they are reported when encountered.

If one is working in display mode and requests a trace, the information about each JSYS trapped will be displayed in the command feedback window and requires an OK before the process is continued. This allows the user to contemplate the flow of the process but can be very annoying if a single JSYS is executing inside a loop a large number of times. In typewriter mode no OK is required and the information will come out as fast as it is encountered.

The frequency distribution table will not be displayed or recorded until the process terminates. Thus, one will not be able to get a frequency distribution of a process that is normally terminated by a "C" since you will be returned to the operating system rather than the encapsulator subsystem.
Task 4: Generalize the PROGRAMS subsystem and its templates.

The AUGMENT PROGRAMS subsystem provides access to tools which aid in the program development process by permitting compilation and testing of programs and by permitting insertion of language dependent program entity templates.

The PROGRAMS subsystem is currently tailored for the L10 and CML languages. We would like to make its features available for other computer languages, e.g., JOVIAL. The main idea was to develop a framework and/or methodology for making the facilities of the PROGRAMS subsystem easily extensible to new computer languages. In the demonstration project, the commands and templates associated with the PROGRAMS system were expanded to deal with the JOVIAL, META (the ARC meta-compiler), and CHL languages as well as L10. Structures relevant to these different languages were created as templates and entered as options into the PROGRAMS commands.

Appendix 1 outlines the command syntax for the new programs subsystem. Appendix 2 presents the current language dependent programming templates inserted via commands in the new PROGRAMS system as extracted from <SSSRC, PROGRAMS-TEMPLATES, >.
Task 5: An interactive, conditional, iterative Process system.

NLS currently has a "Process commands" facility, but it is rather limited. Process commands are currently limited to be the equivalent of NLS commands -- it's roughly equivalent to what a user could type as input, e.g., there are currently no facilities for iteration, or for conditionally performing one set of commands rather than another set, or for interacting with the user in the midst of the execution of the Process commands.

The general idea would be to provide a "Process Language" that had all, or almost all, of the features of a computer language such as ALGOL, e.g., conditional statements, iterative statements, block structure, subroutines, constant and variable data declarations. In addition the language would have provisions for interactions with the user. Calls on L10 procedures probably would also be possible.

Process Language "programs" probably would be interpreted rather than compiled, probably in a manner similar to the present Process commands facility.

Some initial design thoughts on such a facility may be found in <29046>. It was produced for the NSW project and appears as Appendix 3 to this report. It is applicable to the AUGMENT command language. Using this as a guide, we have created a design for the syntax for a complete Process Language. This syntax appears as Appendix 4.
Appendix 1: Command Syntax for the New AUGMENT PROGRAMS Subsystem

This appendix outlines the command syntax for the new PROGRAMS subsystem. The syntax is presented in a modified Command Meta Language description.

Notes:

"12" stands for "12!"

A final CONFIRM is left implicit for all commands.

Curly-brackets are used to indicate a CMU LOOP sort of thing; exit from the loop is made via an OK.

Rules used elsewhere:

\begin{itemize}
  \item anyinput = anytext LSCL / fstructure "at" DSEL
  \item anytext = Character / Invisible / Text / Visible / Word / Statement
  \item fstructure =
    \begin{itemize}
      \item \text{[OPTION "Filtered:" VWSP ECS] (Branch / Group / Plex / Rest / File)}
    \end{itemize}
  \item userprog = Content-Analyzer / Sort-Key / Sequence-Generator
  \item compileunit =
    \begin{itemize}
      \item userprog / Run-Program /
      \item Subsystem /
        \begin{itemize}
          \item Grammar / Parse-Fe (Code/Data) / backend / Support-Module
        \end{itemize}
      \item L10-Program / L101-Program / Cml-Program / Meta-Program / Jovial-Program /
        \begin{itemize}
          \item Procedure / Coroutine /
        \end{itemize}
    \end{itemize}
  \item Insert $\%$ for L10 mode $\%$
    \begin{itemize}
      \item Program
      \item / compileunit
      \item / If-Then-Else / Case
      \item / For / Loop / Do-Until / Do-While / Until-Do / While-Do
    \end{itemize}
\end{itemize}
Appendix 1: PROGRAMS Subsystem Syntax

Insert % for CML mode %
Program <"to follow"> DSEL [LEVADJ]

Insert % for beta mode %
Program <"to follow"> DSEL [LEVADJ]

Insert % for Jovial mode %

( Program
/ Compool!2 Directive / Compool!2 Source / Procedure
/ If-Else / Switch
/ For By-While / For Then-While / While
/ Call / Procedure!2 Call Declaration
)

<"to follow"> DSEL [LEVADJ]

Insert Comment <"to follow"> DSEL % for any mode %

Compile

( Content-Analyzer <"in"> (anyinput / Program <"AT"> DSEL)
/ Program <"AT"> DSEL <"using"> LSEL <"to"> LSEL
/ compileunit <"at"> DSEL
)

[ Record <"errors"> <"at"> DSEL
/ Load <"after compilation">
/ Filtered WVSPECS
]

<"to follow"> DSEL [LEVADJ]

Load (Content-Analyzer / Program!2 / compileunit)
<"named"> LSEL

Deinstitute userprog

Institute!2 userprog <"named"> LSEL

Run

( Run-Program <"named"> LSEL
/ Process <"named"> LSEL
<"input from">
( anyinput
/ interactive <"with termination character"> LSEL
/ Sequential-File <"named"> LSEL
/ No-Input
)
<"output to">
( Wls-File <"at"> LSEL
/ Terminal
/ Sequential-File <"named"> LSEL
/ registers ...
)
Appendix 1: PROGRAMS Subsystem Syntax

```plaintext
"wait for completion and then Kill?" (Yes/No) "notify at completion?" (Yes/No) "go?"

Kill Process <"named"> LSEL 5a12b3
Wait <"for completion and then Kill"> Process <"named"> LSEL 5a12b4
Show (Buffer / Insert-Mode / Process <"named"> LSEL) 5a12b5
Set (Buffer-Size <"to"> LSEL / Insert-Mode <"to"> (Cml / Jovial / L10 / Meta) 5a12b6
Reset (Buffer-Size / Insert-Mode) 5a12b7
Delete!2 (All / Last) 5a12b8
Invoke!2 (Dad / Ddt!2) 5a12b9
```
The following are the current programming templates inserted via commands in the new PROGRAMS system as extracted from <SSSRC, PROGRAMS-TEMPLATES, >:

```cml
(grammar)
%GR% FILE SubSysName % Grammar %
% COMPIL-INSTRUCTIONS %
INCLUDE <SsSrc, NLS-Grammar, flags !subsystems>

% DECLARATIONS %
INCLUDE <SsSrc, NLS-Grammar, declarations !universal>
DECLARE COMMAND WORD
%subsystem command words (should be 100 to 127)%
"COMMAND1" = 100,
"COMMANDn" = 10n-1;
%
DECLARE FF FUNCTION
fefunc1,
fefuncn;
DECLARE FUNCTION
PROCESS = "PROCESSX", PACKAGE = "PCKX":
xroutine1,
xroutinen;
DECLARE GLOBAL
global1, % short description of global1 %
globaln; % short description of globaln %
DECLARE VARIABLE
variable1,
variablen;
%
% COMMON-RULES %
INCLUDE <SsSrc, NLS-Grammar, rules !universal> 6a1a1
rule1 = <rule-body>;
rule1n = <rule-body>;
%
% COMMANDS % SUBSYSTEM SubSysName KEYWORD
"SUBSYSNAME"
INITIALIZATION % for SubSysName %
```


```plaintext
initsubsys = xininitsubsys();

TERMINATION % for SubSysName 
trmsubsys = xtramsubsys();

command1 COMMAND = "COMMANDWORD1"
  <rule-body>
  xroutine1(variable1, variable2);

command2 COMMAND = "COMMANDWORD2"
  <rule-body>
  xroutine2(variable1, variable2);

INCLUDE <SsSrc, NLS-Grammar, commands
  universal>
END.
FINISH

(program)

%GH% FILE SubSysName % Grammar %
% COMPIL-E-INSTRUCTIONS %
INCLUDE <SsSrc, NLS-Grammar, flags !subsystems>
% DECLARATIONS %
INCLUDE <SsSrc, NLS-Grammar, declarations
  universal>
DECLARE COMMAND WORD
% subsystem command words (should be 100 to
127)%
"COMMAND1" = 100,
"COMMANDn" = 10n-1;
%
DECLARE EFUNCTION
  efunci,
  efunchn;
DECLARE FUNCTION
  PROCESS = "PROCESSX" , PACKAGE = "PCKX" :
    xroutine1,
    xruniten;
DECLARE GLOBAL
  global1, % short description of global1 
  globaln; % short description of globaln

DECLARE VARIABLE
  variable1,
  variablen;
%
% COMMON-RULES %
```

On-Line Programmers Management System:
Final Technical Report
Appendix 2: PROGRAMS Subsystem Templates

INCLUDE <SsSrc, NLS-Grammar, rules !universal> 6a1b1c1
rule1 = <rule-body>; 6a1b1c2
rule2 = <rule-body>; 6a1b1c3

%%
% COMMANDS % SUBSYSTEM SubSysName KEYWORD
"SUBSYSNAME"
INITIALIZATION % for SubSysName %
initsubsys = xinitsubsys(); 6a1b1d1

TERMINATION % for SubSysName %
trmsubsys = xttrmsubsys(); 6a1b1d2

command1 COMMAND = "COMMANDWORD1"
    <rule-body>
    xroutine1(variable1, variable2); 6a1b1d3

command2 COMMAND = "COMMANDWORD2"
    <rule-body>
    xroutine2(variable1, variable2); 6a1b1d4

INCLUDE <SsSrc, NLS-Grammar, commands !universal> 6a1b1d5
END. 6a1b1d6
FINISH 6a1b1e

(110) 6a2

(FILE SubSysName % BackEnd %

% DECLARATIONS %

% Dispatch-Table: address of routine-name-string, address of routine %
(subsysname) EXTERNAL = ( 6a2a1a1
"XROUTINE1", $xroutine1, 6a2a1a1a
"XROUTINE2", $xroutinen, 6a2a1a2
0, 0); % table must end with zero %

% Command Words %
(constant) CONSTANT = 100B; 6a2a1a2a
(constant) CONSTANT = 10n-1E; 6a2a1a2b
%
% X-ROUTINES %
(xroutines1) % one-line description % 6a2a1b1
(xroutinesn) % one-line description % 6a2a1b2
%
% CORE-ROUTINES %
(coutines1) % one-line description % 6a2a1c1
(crouitinen) % one-line description % 6a2a1c2
% SUPPORT-ROUTES % 6a2a1d
(sroutine) % one-line description % 6a2a1d1
(sroutine) % one-line description % 6a2a1d2
FINISH 6a2a1e

(catchphrase) 6a2b
(catchname) CATCHPHRASE(arg1, arg2, arg3, arg4); 6a2c1
BEGIN
CASE SIGNALTYPE OF
= notypel : 6a2c1a
= helpyle : 6a2c1b
= abortype : 6a2c1b1
CASE SIGNAL OF
= xxx : 6a2c1b2
= yyy : 6a2c1b3
ENDCASE;
CONTINUE;
END;

(content-analyzer) 6a2d
(analyzer-name) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ..., argn % => [meta-res] res1
<type>, ..., resn %); 6a2d1
% Procedure description
FUNCTION
none
ARGUMENTS
none
RESULTS
none
NON-STANDARD CONTROL
none
%
% Declarations %
% procedure body%
% Return %
RETURN;
END.
%%

Appendix 2: PROGRAMS Subsystem Templates

(coroutine) % CL: ; one-line-description %
COROUTINE (arg1 <type>, ..., argn);
% Coroutine description
FUNCTION
none
ARGUMENTS
none
RESULT
none
NON-STANDARD CONTROL
none
% Declarations %
% Initial entry point %
PORT ENTRY
% Initialization %
EXIT PCALL;
%coroutine body%
END.
%

(cml-program)
%GR% FILE SubSysName % Grammar %
% COMPILE-INSTRUCTIONS %
INCLUDE <SsSrc, NLS-Grammar, flags !subsystems>
% DECLARATIONS %
INCLUDE <SsSrc, NLS-Grammar, declarations !universal>
DECLARE COMMAND WORD
%subsystem command words (should be 100 to 127)%
"COMMAND1" = 100,
"COMMANDn" = 10n-1;
%
DECLARE FEFUNCTION
fefunc1,
fefuncn;
DECLARE FUNCTION
PROCESS = "PROCESSX" , PACKAGE = "PCKX" :
xroutine1,
xroutinen;
DECLARE GLOBAL
global1, % short description of global1 %
globaln; % short description of globaln %
DECLARE VARIABLE

On-Line Programmers Management System:
Final Technical Report
variable1,
variable2;

%%
% COMMON-RULES %
INCLUDE <SsSrc, NLS-Grammar, rules universal>

rule1 = <rule-body>;
rule2 = <rule-body>;

%%
% COMMANDS % SUBSYSTEM SubSysName KEYWORD
"SUBSYSNAME"

INITIALIZATION % for SubSysName %
initsubsys = xinitsubsys();

TERMINATION % for SubSysName %
trmsubsys = xtrmsubsys();

command1 COMMAND = "COMMANDWORD1"
<rulbody>
xroutine1(variable1, variable2);

command2 COMMAND = "COMMANDWORD2"
<rulbody>
xroutine2(variable1, variable2);

INCLUDE <SsSrc, NLS-Grammar, commands universal>

END.

FINISH

(do-until)
DO
BEGIN
END
UNTIL until-clause;

(do-while)
DO
BEGIN
END
WHILE while-clause;

(xefunction)
(FFEfunctionName) % CL: ; one-line description %
PROCEDURE (reason, instruction, accumulator REP,
argcount, arguments REP, saveword % => result %);
% FEFfunction description

On-Line Programmers Management System:
Final Technical Report
23
FUNCTION 6a21a1
none 6a21a1a
ARGUMENTS (show grammar arguments) 6a21a2
none 6a21a2a
RESULT (show grammar results) 6a21a3
none 6a21a3a
NON-STANDARD CONTROL 6a21a4
none 6a21a4a
\%
\% Declarations %
CASE reason OF 6a21b
= parsing: % being invoked for first time during
command % 6a21c1
BEGIN 6a21c1a
% decide whether FF is on the correct path
through the grammar % 6a21c1b
IF % not on right path, may % THEN RETURN
(notme); 6a21c1c
% do processing % 6a21c1d
saveword _ % word of context to be saved or 0
- if 0, will not be called during backup,
etc. %; 6a21c1e
omlresults (N, result1, ... resulth); % re-
turn N results - need not be called if not
returning any results. Results returned will
be freed by the FE automatically% 6a21c1f
RETURN (dosuc, saveword); 6a21c1g
END; 6a21c1h
= terminate: % command is done, cleanup %
= abortcmd: % command was aborted, cleanup and
restore state % 6a21c3
= backup: % command was backed up %
ENDCASE ABORT (???, "%Bug: Illegal reason to a
FEFunction"); 6a21c5
RETURN (notme); 6a21d
END.
%%% 6a21e
(?) 6a21e1

FOR for-clause DO 6a2aj
BEGIN 6a2aj1
END; 6a2ajb

(?) 6a2k
%GR% FILE SubSysName % Grammar % 6a2k1
% COMPILe—INSTRUCTIONS %
INCLUDE <SsSrc, NLS—Grammar, flags !subsystems> 6a2k1a
% DECLARATIONS % 6a2k1b
Appendix 2: PROGRAMS Subsystem Templates

```
INCLUDE <SsSrc, NLS-Grammar, declarations universal>

DECLARE COMMAND WORD 6a2k1b1
    %subsystem command words (should be 100 to 127)
    "COMMAND1" = 100,
    "COMMANDn" = 10n-1;

DECLARE FUNCTION PROCESS = "PROCESSX", PACKAGE 6a2k1b2
    PCKX
      xroutine1, 6a2k1b3
      xroutine0;
    DECLARE FE FUNCTION 6a2k1b4
      fefunc1, 6a2k1b5
      fefuncn;
    DECLARE PARSEFUNCTION 6a2k1b6
      pffunc1, 6a2k1b7
      pffuncn;
    DECLARE GLOBAL 6a2k1b8
      global1, 6a2k1b9
      globaln;
    DECLARE VARIABLE 6a2k1b10
      variable1, 6a2k1b11
      variablen;

% COMMON—RULES

INCLUDE <SsSrc, NLS-Grammar, rules universal>

rule1 = <rule—body>;
rule2 = <rule—body>;

% COMMANDS % SUBSYSTEM SubSysName KEYWORD
"SUBSYSNAME"

INITIALIZATION % for SubSysName %
    initsubsy = xinitsubsyst();

TERMINATION % for SubSysName %
    trmsubsy = xtrmsubsyst();

command1 COMMAND = "COMMANDWORD1"
    <rule—body>
    xroutine1(variable1, variable2);

command2 COMMAND = "COMMANDWORD2"
    <rule—body>
    xroutine2(variable1, variable2);
```
INCLUDE <SSSsrc, NLS-Grammar, commands
tuniversal>
END.
FINISH

(if-then-else)
IF if-clause THEN
BEGIN
END
ELSE
BEGIN
END;

(jovial-program)
!COMPOOL! ('compool-file') name, name;
PROGRAM programe name " Description "
BEGIN
" Program description "
" DECLARATIONS "
" FUNCTIONS DEFINED "
" one-line-description "
DEF PROC proctname (input-parameterl ,..., input-parameter : output parameterl ,..., output-parameter) functiontype ;
" Procedure description
FUNCTION none
ARGUMENTS none
RESULTS none
NON-STANDARD CONTROL none
BEGIN
RETU RN;
END
" SUBROUTINES DEFINED "
" one-line-description "
REF PROC (procname) (input-parameterl ,..., input-parameter : output parameterl ,..., output-parameter) ;
" Procedure description
FUNCTION none
ARGUMENTS none
Appendix 2: PROGRAMS Subsystem Templates

none

RESULTS

none

NON-STANDARD CONTROL

none

" BEGIN
" Declarations "
" procedure body "
" Return "
RETURN;
END

" ITEMS DEFINED "
ITEM itemname itemtype;

"program body"
END

%(110—program)
FILE ProgramName % Description %
% DECLARATIONS %
%%
% PROCEDURE %
(procname) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ... , argn % => [meta- res] res1 <type>, ... , resn %);
% Procedure description
FUNCTION

none
ARGUMENTS

none
RESULTS

none
NON-STANDARD CONTROL

none

% Declarations %
%procedure body%
% Return %
RETURN;
END.
%%

FINISH
%(11011—program)
FILE ProgramName % Description %
% DECLARATIONS %
%%
% PROCEDURE %

On-Line Programmers Management System:
Final Technical Report
27
Appendix 2: PROGRAMS Subsystem Templates

(procname) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ..., argn => [meta-
res] res1 <type>, ..., resn %);
% Procedure description
FUNCTION
  none
ARGUMENTS
  none
RESULTS
  none
NON-STANDARD CONTROL
  none
%
% Declarations %
%procedure body%
% Return %
RETURN;
END.
%%
FINISH
(loop)
LOOP
  BEGIN
END;
(meta-program)
FILE filename Check
META program
% Compiler header. %
ERROR: ;
SIZE: ;
FLAGS: ;
DUMMY: ;
SET: ;
FIELDS: ;
ATTRIBUTES: ;
OPCODES: ;
% Compiler header syntax. %
% Rules. %
END of TREE META
(code)
%PFC% COROUTINE (parm1, parm2, parm3);
END.
(data)
%PFD% FILE SubSysName % Parse/Fe functions Data %
FINISH
(function)
(ParseFunctionName) % CL: ; one-line-description %
COROUTINE (reason, instruction, accumulator REF,
argcount, arguments REF, saveword % => result %);
Appendix 2: PROGRAMS Subsystem Templates

% Parsefunction description
FUNCTION none
ARGUMENTS (show grammar arguments) none
RESULT (show grammar results) none
NON-STANDARD CONTROL none

% Declarations %
% Initial entry point %
PORT ENTRY % initialization %
EXIT PCALL; %coroutine body%
END.

(parsefunction)
(ParseFunctionName) % CL: ; one-line-description %
COROUTINE (reason, instruction, accumulator REF, argcount, arguments REF, saveword % => result %);

% Parsefunction description
FUNCTION none
ARGUMENTS (show grammar arguments) none
RESULT (show grammar results) none
NON-STANDARD CONTROL none

% Declarations %
% Initial entry point %
PORT ENTRY % initialization %
EXIT PCALL; %coroutine body%
END.

(procedure)
(procname) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ..., argn => [meta-res] res1 <type>, ..., resn %);

% Procedure description
FUNCTION none
ARGUMENTS none
Appendix 2: PROGRAMS Subsystem Templates

RESULTS

none

NON-STANDARD CONTROL

none

%

% Declarations

%procedure body

% Return

RETURN;

END.

%%

(program)

%GR% FILE SubSysName % Grammar %

% COMPILE-INSTRUCTIONS %

INCLUDE <SsSrc, NLS-Grammar, flags !subsystems>

% DECLARATIONS %

INCLUDE <SsSrc, NLS-Grammar, declarations !universal>

DECLARE COMMAND WORD

%subsystem command words (should be 100 to 127)%

"COMMAND1" = 100,

"COMMANDn" = 10n-1;

%

DECLARE PEFUNCTION

fefunc1,

fefuncn;

DECLARE FUNCTION

PROCESS = "PROCESSX", PACKAGE = "PCKX" :

xroutinen,

xroutinen;

DECLARE GLOBAL

global1, % short description of global1 %

globaln; % short description of globaln %

DECLARE VARIABLE

variable1,

variablen;

%

% COMMON-RULES %

INCLUDE <SsSrc, NLS-Grammar, rules !universal>
Appendix 2: PROGRAMS Subsystem Templates

%%
% COMMANDS % SUBSYSTEM SubSysName KEYWORD
"SUBSYSNAME"
INITIALIZATION % for SubSysName %
initsubsys = xinitsubsys();
TERMINATION % for SubSysName %
trmsubsys = xtrmsubsys();
command1 COMMAND = "COMMANDWORD1"
<rule-body>
xroutine1(variable1, variable2);
command2 COMMAND = "COMMANDWORD2"
<rule-body>
xroutine2(variable1, variable2);
INCLUDE <ScSrc, NLS-Grammar, commands
!universal>
END.
FINISH

%FPC% FILE SubSysName % Parse/Fe functions Code %
% DECLARATIONS %
% CODE %
(FEFunctionName) % CL: ; one-line description
% PROCEDURE (reason, instruction, accumulator REF, argcount, arguments REF, saveword % => result %);
% FEFunction description
FUNCTION
none
ARGUMENTS
reason - reason function is being invoked
instruction - byte pointer to grammar instruction
accumulator - pointer to global accumulator
argcount - count of number of arguments from grammar call
arguments - pointer to array of argument values from grammar call
saveword - word of context retained by Frontend
RESULT
reason describing function result
NON-STANDARD CONTROL

none

% Declarations %

CASE reason OF

= parsing: % being invoked for first time during command %

BEGIN

% decide whether FF is on the correct path through the grammar %

IF % not on right path % THEN RETURN

(notme);

% do processing %

saveword _ % word of context to be saved or 0 - if 0, will not be called during backup, etc. %;

% commands (N, result1, ... resultN); %

return N results - need not be called if not returning any results. Results returned will be freed by the FE automatically%

RETURN (dosuc, saveword);

END;

= terminate: % command is done, cleanup %

= abortcmd: % command was aborted, cleanup and restore state %

= backup: % command was backed up %

ENDCASE ABORT (???, "$Bug: Illegal reason to a FEFuncti on");

RETURN;

END.

%%

(ParseFunctionName) % CL: ; one-line-
description %

COROUTINE (reason, instruction, accumulator REF, argcount, arguments REF, saveword % => result %);

% Parsefunction description

FUNCTION

none

ARGUMENTS

reason - reason parsefunction is being invoked

instruction - byte pointer to grammar

accumulator - pointer to global accumulator

On-Line Programmers Management System:
Final Technical Report
32
Appendix 2: PROGRAMS Subsystem Templates

argcount - count of number of arguments from grammar call
arguments - pointer to array of argument values from grammar call
saveword - word of context retained by Frontend
RESULT
reason describing parsefunction result

NON-STANDARD CONTROL
none
%
% Declarations %
% Initial entry point %
PORT ENTRY
% Initialization %
EXIT PCALL;
<coroutine body>
END.
%
FINISH

%PFD% FILE SubSysName % Parse/Fe functions Data %
FINISH

%BE% FILE SubSysName % Backend %
% DECLARATIONS %
% Dispatch-Table: address of routine-name-string, address of routine %
(subsysname) EXTERNAL = ( 6a2w2b2a2d
"XROUTINE1", $xroutine1, 6a2w2b2a2e
"XROUTINE2", $xroutine2, 6a2w2b2a2f
0,0); % table must end with zero %
%
% Command Words %
(cw1) CONSTANT = 100B; 6a2w2b2a2g
(cwn) CONSTANT = 10n-15; 6a2w2b2a2h
%
% X-ROUTINES %
(xroutine1) % one-line description %
(xroutinen) % one-line description % 6a2w2b2a2i
% CORE-ROUTINES %
(croutine1) % one-line description %
(croutinen) % one-line description %
% SUPPORT-ROUTINES %
(sroutine1) % one-line description %
(sroutine1) % one-line description %
FINISH
(run-program)

FILE ProgramName % Description %
% DECLARATIONS %

(ProgramName) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ..., argn % => [meta-res] res1
  <type>, ..., resn %);
% Procedure description
  FUNCTION
    none
  ARGUMENTS
    none
  RESULTS
    none
  NON-STANDARD CONTROL
    none
% % Declarations %
% procedure body% % Return %
  RETURN;
END.
FINISH %%

(sequence-generator)

(nameofseqgen) % CL: ; one-line-description %
PROCEDURE (sw REF, entrytype);
% Procedure description
  FUNCTION
    none
  ARGUMENTS
    sw - REF-address of sequence work area. See
    record def, (nine, brecords, seqr).
    entrytype - INTEGER-entry type.
    =sqopn: called at seq open to initialize
    a work area
    =sqgnxt: called for next in seq
    =sqcls: called at seq close to release
    the workarea
% RESULTS
  none
  NON-STANDARD CONTROL
  none
% When called with entry type sqgnxt, a pseudo
  coroutine is used for the sequence generator
  return mechanism. Control is given up by
  calling send or sport, which eventually
  switches the stack. Control is returned when
a call is made to seqgen, which makes a
coroutine port call to the stack associated
with this sequence work area.

A signal is generated (err is called) when
the seq generator is called with an illegal
entry type.

Note: for an example look at system sequence
generator.

% Declarations %
% Caution!! Locals are not consistent across en-
tries %
% select entry point %
CASE entrytype OF
  =sqopn: % called at seq open %
    NULL;
  =sqgnxt: % call for next in seq %
    LOOP
    BEGIN
    % perform activities unique to this seq
generator %
    % make "coroutine" call via call to
    send or sport %
    send($sw, - addr of a sequence
    work area
    $str); - an ENDFIL or addr
    of a string %
    sport($sw); - addr of sequence
    work area%
    % returned here by sport call from
    seqgen %
    % get next in sequence %
    END;
  =sqcls: % called at seq close %
    NULL;
ENDCASE err("Bug");
% Return %
RETURN;
END.
%%

On-Line Programmers Management System:
Final Technical Report
Appendix 2: PROGRAMS Subsystem Templates

(defkey) % CL: ; one-line-description %
PROCEDURE (std, % handle of the statement begin
considered %
   buffer, % address of a buffer to hold the
   sorting index %
   bufflength; % the maximum size (in words) of
   the sorting index %
   % Procedure description
   FUNCTION
   none
   ARGUMENTS
   none
   RESULTS
   none
   NON-STANDARD CONTROL
   none
   %
   % Declarations %
   %procedure body %
   % Return %
   RETURN (partial-index-flag, % TRUE - data in
   buffer only high-order bits of the sorting
   index FALSE - buffer value is full sorting
   index %
   word-count); % integer indicating the number
   of words actually used in the buffer %
   END.
   %
   buffer, bufflength);
FINISH

(subsystem)

(FILE SubSysName % Grammar %
% COMPILe-INSTRuctionS %
INCLUDE <SsSrC, NLS-Grammar, flags |subsystems>

% DECLARATIOnS %
INCLUDE <SsSrC, NLS-Grammar, declarations
|universal>
DECLARE COMMAND WORD
% subsystem command words (should be 100 to
127) %
"COMMAND1" = 100,
"COMMANDn" = 10n-1;
%
DECLARE fEFUNCTf
efunct1,  

On-Line Programmers Management System:
Final Technical Report
36
Appendix 2: PROGRAMS Subsystem Templates

FE FUNCTION
PROCESS = "PROCESSX", PACKAGE = "PCKX";
ROUTINE1;
ROUTINE2;
DECLARE GLOBAL
GLOBAL1, % short description of global1%
GLOBALN, % short description of globalN%
DECLARE VARIABLE
VARIABLE1,
VARIABLEN;
%%
% COMMON-RULES%
INCLUDE <SsSrc, NLS-Grammar, rules !universal>

RULE1 = <rule-body>;
RULEN = <rule-body>;
%%
% COMMANDS % SUBSYSTEM SubSysName KEYWORD "SUBSYSNAME"
INITIALIZATION % for SubSysName %
initsubsys = xinisubsys();

TERMINATION % for SubSysName %
trmsubsys = xtrmsubsys();

COMMAND1 COMMAND = "COMMANDWORD1"
<rul-body>
ROUTINE1(VARIABLE1, VARIABLE2);

COMMAND2 COMMAND = "COMMANDWORD2"
<rul-body>
ROUTINE2(VARIABLE1, VARIABLE2);

INCLUDE <SsSrc, NLS-Grammar, commands !universal>
END.
FINISH

%PFC% FILE SubSysName % Parse/Fe functions Code %
% DECLARATIONS %
% CODE %
(FFEFuntionName) % CL: ; one-line description

On-Line Programmers Management System:
Final Technical Report
37
PROCEDURE (reason, instruction, accumulator REF, argcount, arguments REF, saveword % => result %);

% Function description
FUNCTION none

ARGUMENTS
reason - reason function is being invoked
instruction - byte pointer to grammar
accumulator - pointer to global accumulator
argcount - count of number of arguments from grammar call
arguments - pointer to array of argument values from grammar call
saveword - word of context retained by Frontend

RESULT
reason describing function result

NON-STANDARD CONTROL

% Declarations %

CASE reason OF
= parsing: % being invoked for first time during command %
BEGIN
% decide whether FF is on the correct path through the grammar %
IF % not on right path % THEN RETURN (notme);
% do processing %
saveword - % word of context to be saved or 0 - if 0, will not be called during backup, etc. %;
omlresults (N, result1, ... resultN); %
return N results - need not be called if not returning any results. Results returned will be freed by the FE automatically%
RETURN (dosuc, saveword);
END;
= terminate: % command is done, cleanup %
= abortcmd: % command was aborted, cleanup and restore state %
Appendix 2: PROGRAMS Subsystem Templates

= backup: % command was backed up % 6a2aa2b1c4
= ENDCASE AUCHT (???, "Bug: illegal reason
to a FEFunction"); 6a2aa2b1c5
= RETURN;
= END.
= 6a2aa2b1d
= 6a2aa2b1e
= 6a2aa2b1f
% (ParseFunctionName) % CL: ; one line description %
% COROUTINE (reason, instruction, accumulator hEF,
argcount, arguments ARF, saveword % => result %);
% Parsefunction description 6a2aa2b2
FUNCTION 6a2aa2b2a none 6a2aa2b2a1
ARGUMENTS 6a2aa2b2a2
reason - reason parsefunction is being invoked 6a2aa2b2a2a
instruction - byte pointer to grammar instruction 6a2aa2b2a2b
accumulator - pointer to global accumulator 6a2aa2b2a2c
argcount - count of number of arguments from grammar call 6a2aa2b2a2d
arguments - pointer to array of argument values from grammar call 6a2aa2b2a2e
saveword - word of context retained by Frontend 6a2aa2b2a2f
RESULT 6a2aa2b2a3 reason describing parsefunction result 6a2aa2b2a3a
NON-STANDARD CONTROL none 6a2aa2b2a4
% % Declarations %
% Initial entry point %
ENTRY 6a2aa2b2a5
% Initialization %
EXIT PCALL; 6a2aa2b2a6
<coroutine body>
END.
% FINISH 6a2aa2b2a7
% PFD% FILE SubSysName % Parse/Fe functions Data % 6a2aa3
FINISH 6a2aa3a
% BE% FILE SubSysName % BackEnd % 6a2aa4
% DECLARATIONS %
6a2aa4a

On-Line Programmers Management System:
Final Technical Report
39
 Dispatch-Table: address of routine-name-string, address of routine%
(subsysname) EXTERNAL =
"XROUTINE1", $xroutine1,
"XROUTINE2", $xroutine2,
0,0); % table must end with zero%
%
 Command words%
(cw1) CONSTANT = 100B;
(cwn) CONSTANT = 10n-1b;
%
 X-ROUTINES%
(xroutine1) % one-line description%
(xroutinen) % one-line description%
%
 CORE-ROUTINES%
(croutine1) % one-line description%
(croutinen) % one-line description%
%
 SUPPORT-ROUTINES%
(sroutine1) % one-line description%
(srcoutinen) % one-line description%

 On-Line Programmers Management System:
 Final Technical Report

Appendix 2: PROGRAMS Subsystem Templates

(procname) % CL: ; one-line-description %
PROCEDURE (arg1 <type>, ..., argn = > [meta-
res1 <type>, ..., resn %);       6a2ab1d1
% Procedure description
FUNCTION
  none
  6a2ab1d1a1
  6a2ab1d1a1
ARGUMENTS
  none
  6a2ab1d1a2
  6a2ab1d1a2
RESULTS
  none
  6a2ab1d1a3
  6a2ab1d1a3
NON-STANDARD CONTROL
  none
  6a2ab1d1a4
  6a2ab1d1a4
% Declarations %
%procedure body%
% Return %
  RETURN;
END.
%%
FINISH

(until-do)       6a2ac
UNTIL until-clause DO
BEGIN
END;

(while-do)       6a2ad
WHILE while-clause DO
BEGIN
END;

(jovial)         6a3

(directive)      6a3a
ICOMPOOL 'compool-file' name, name ;

(source)         6a3b
COMPOOL compool-name ;
BEGIN
END

(by-while)        6a3c
FOR index: initial BY increment WHILE index relational
  stopvalue;
BEGIN
END

(then-while)      6a3d
FOR item-name THEN formula WHILE conditional-formula ;

On-Line Programmers Management System:
Final Technical Report
41
statement ;

(if-else)
IF conditional-formula ;
statement ;
ELSE else-statement ;

(call)
procedure-name@data-base(input-parameter, input-parameter : output-parameter, output-parameter) ;
"comment"

(declaration)
PROC procedure-name data-allocator(input-parameter, input-parameter : output-parameter, output-parameter) ;
BEGIN
"DECLARATIONS"
"PROCEDURE BODY"
END

(program)
!COMPOOL! ('J73IO.CMP');

PROGRAM programname " Description "
BEGIN
" Program description "
" DECLARATIONS "
" EXTERNAL PROCEDURES "
" one-line-description "
REF PROC prooname (input-parameter1, ..., input-parameter : output-parameter1, ..., output-parameter) ;
" Procedure description
FUNCTION
none
ARGUMENTS
none
RESULTS
none
NON-STANDARD CONTROL
none
" BEGIN
" Declarations "
END
" SUBROUTINES DEFINED "
" one-line-description "

DEF PROC (prooname) (input-parameter1, ..., input-parameter-n : output parameter1, ..., output-parameter-n);  
  " Procedure description
  FUNCTION
  none
  ARGUMENTS
  none
  RESULTS
  none
  NON-STANDARD CONTROL
  none

BEGIN
  " Declarations "
  " procedure body"
  " Return "
  RETURN;
END

" ITEMS DEFINED "
ITEM itemname itemtype;
"program body"
END
%%

(program)
FILE filename % one line comment %
META program
Appendix 2: PROGRAMS Subsystem Templates

% COMPILED HEADER %
DUMMY: 6a4a1a1
ERROR: 6a4a1a1a
FLAGS: 6a4a1a1b
ATTRIBUTES: 6a4a1a1c
OPCODES: 6a4a1a1d
SET: 6a4a1a1e
SIZE: 6a4a1a1f
FIELDS: 6a4a1a1g
% SYNTAX RULES %
identifier = rulebody;
% PRODUCTION RULES %
identifier[test-expression] => 6a4a1a3
production-rule-body;
identifier[test-expression] => 6a4a1a4
production-rule-body;
% VALUE RULES %
identifier[test-expression] := 6a4a1a4a
value-rule-body;
identifier[test-expression] := 6a4a1a4b
value-rule-body;
FINISH

On-Line Programmers Management System:
Final Technical Report
Appendix 3: Command Sequence Processor Design Specifications

Preface

This report is a user-level description of a National Software Works facility for writing and executing Command Sequences for NSW tools and the NSW EXEC. It was prepared for the RADC NSW project, Contract F 30602-75-C-0320.

Introduction

A Command Sequence is a collection of one or more commands with a unique name. The user invokes a Command Sequence by its name; the NSW Frontend then processes the Sequence as if the user were typing in that collection of commands himself. The commands available for Command Sequence use include the NSW EXEC commands, all split tool commands, and unsplit tool input—in short, everything the user is allowed to do in the NSW, including the use of other Command Sequences. This Frontend feature and its associated program modules are called the Command Sequence Processor.

The great advantage of a Command Sequence facility, of course, is that it allows users to "program" in the command language with which they are familiar; that is, they can specify a series of operations and have this "Program" executed at any time. No programming language must be learned. Although this kind of facility is available on many time-sharing systems, it is generally missing the control constructs (e.g., IF, FOR, CASE) so heavily used in algorithmic languages. The NSW Command Sequence Processor includes control features, and hence provides a complete language for "command programming".

The CLI grammar-driven interface system, with its recognition modes, feedback and noise words, and help features, is a significant improvement in making man-machine interfaces coherent and natural. The inclusion of the Command Sequence facility complements a powerful system by bringing programming-like capabilities into the user interface.
Appendix 3: Command Sequence Processor Design Specifications

Capabilities

The Command Sequence Processor (CSP) will have three basic capabilities:

7d1 Running "canned" command strings, in the same manner as TENEX Runfile.

7d1a Collecting selections from the user at Command Sequence execution time. These may be used (perhaps more than once during an execution) as user supplied arguments in the canned commands.

7d1b Testing conditions and doing different things based on the outcome. The conditions may be user input, variables, or the results of commands. Further, control constructs allow sequences to loop over a group of commands until a specified condition is met.

7d1c The CSP operates independently of the Command Language Interpreter, and hence functions across the EXEC and all tools. The sequences may contain commands to run a tool, followed by commands for the tool, followed by more EXEC commands, and so forth.

What the User Needs to Know

7e A minimum of information is required to use the CSP. The user need not learn a new language since a Command Sequence is constructed of user-level commands, written in textual form exactly as he would see them when executing such commands. The command words are written in full; noise words may or may not be present.

7e1 To have the Command Sequence performed, the user executes a CSP command (available at the EXEC and all split tools), specifying a Command Sequence name and arguments, if any. As an alternative, he can have the sequence name available as a top-level command in all grammars (EXEC and all split tools).

7e2 Although Command Sequences cannot be invoked from an unsplit tool without escaping back to the EXEC, they can specify commands for unsplit tools. In that case, the Command Sequence will contain the text that would be typed to the tool from the terminal, which may not be as readable for the user.
Appendix 3: Command Sequence Processor Design Specifications

as Command Sequences for split tools. This is unavoidable since the command recognition mode of unsplit tools in general cannot be controlled.

To take advantage of advanced CSP capabilities, the user must know the syntax and semantics of the control constructs necessary to obtain selections from him, test variables, perform looping, etc. These constructs are as simple and intuitive as possible.

Command Sequence Generation

A Command Sequence may be generated in several ways. The most obvious is to write the text or retrieve and edit an existing Command Sequence with an editor. However, there are more convenient methods.

A Command Sequence can be generated with the aid of the CSP itself. Basically, the user executes a CSP command to begin recording a Command Sequence. From that point on, every bit of input the user gives is incorporated into the Command Sequence. This includes EXEC, split, and unsplit tool commands. This continues until the user terminates the recording with another CSP command. The result is a Command Sequence that can be invoked immediately or stored for future use.

At "start recording" time the user may specify whether or not to actually execute the following (recorded) commands; that is, he may generate a Command Sequence without actually executing any commands. If unsplit tool commands are given in this mode, there is no feedback from the tool, since it is not really executing. For split tool and EXEC commands, the command feedback is exactly as if the command were executed, except for messages that would come from the tool itself or from the Works Manager.

The Command Sequence Generator Tool may also be used to originate a Command Sequence. This tool aids the user while he steps through the commands for his sequence. It simulates the "recording without executing" case above, but makes it possible to specify user input collection, testing, branching, and looping points within the sequence. The result is again a Command Sequence ready for use with the CSP. This tool also has a "debugging" mode whereby Command Sequences can be executed in slow motion and modified if necessary.

On-Line Programmers Management System:
Final Technical Report
47
Appendix 3: Command Sequence Processor Design Specifications

Command Sequence Control Constructs

This section is intended to give the reader an idea of the form and capabilities of the Command Sequence control constructs. It may be incomplete in some respects. The control escape character is printed as an exclamation mark (!). Command words are capitalized and user input is indicated inside angle brackets (< and >). Noise words are in parentheses.

All of the following commands may be executed when the user is in any split tool or at the NS~ EXEC. They may also appear in Command Sequences. Note that the "canned" Command Sequence capability can be used when only the "recording" and "executing" commands are known. The other control commands are for more advanced capabilities.

Executing Command Sequences

!Do (CS name) <name>

This command causes the Frontend to execute commands in the Command Sequence named "name".

Recording Command Sequences

!Start Recording (CS name) <name>

<commands to be in Command Sequence "name">

!Stop Recording

The Start and Stop commands are used to create a Command Sequence named "name".

Getting Selections from the User and Showing Strings

!Text (from user into) <varname>

!Character (from user into) <varname>

!Word (from user into) <varname>

The above commands cause the Command Sequence Processor to collect the specified kind of selection from the user rather than obtaining it from the Command Sequence text. After collecting the selection from the user, the CSP stores the input in the named variable.
Appendix 3: Command Sequence Processor Design Specifications

("varname" here) and directs its attention back to the Command Sequence text. These commands might be used to allow the user to specify a file name or other argument in a command the Command Sequence is performing.

!Selection (from user into) <selvar>

This command collects a specific kind of selection from the user. It is used when the Command Sequence writer wants to get user input appropriate for a command in his Sequence. To do so, he specifies that command, using the "Selection" command at the point in which user input is required. The Selection command then looks at the grammar for the command he is specifying to determine what kind of selection is needed from the user. The selection input is stored in "selvar" and at the same time is provided as input to the selection instruction. In the following example the user will give a selection for the Insert Statement command. The selection will be saved in variable "place" for possible later use:

```
Insert Statement !Selection (from user into)
<place> <CA>
```

! ( <noise words> )

The "noise words" command () puts the given text in the command feedback line, allowing the CS writer to prompt the user with text strings. (Syntax note: In this command the parentheses indicate what the user inserts in his Command Sequence rather than noise words displayed by a command.) The following illustrates the use of this command. It shows how to collect a string from the user and save it:

```
! ( <some text> )

!Text (from user into) <save>
```

Conditionals, Looping

The following commands control the CSP's path over the Command Sequence. Their arguments are single commands.

```
!Begin (command group) <commands> !End
```
Appendix 3: Command Sequence Processor Design Specifications

The Begin and End commands make several commands (those grouped between the commands) into a single command.

!If <command> (then) <command> (else) <command>

The If command permits the execution of either one command or another, based on the result of the first command. Every command has a True or False result which determines whether the (then/True) or (else/False) commands are executed. A command always has the result TRUE unless one of the following happens:

A remote (backend) call returns a failure result.
A global variable is explicitly set by the grammar during command execution (the variable's name is not specified at this time).
The CLI aborts the command for some reason.
The command is "!FALSE".

!Loop <command>

The Loop command causes the specified "command" to be executed repeatedly.

!Exit (loop)

This command causes the CSP to stop performing the innermost loop and continue with the command following the Loop command.

!Repeat Loop

This causes the CSP to start over at the first command of the current loop being performed.

Variables

!Define (variable named) <varname>

!Local (variable named) <locname>

The Define command defines a "global" variable; that is, the variable can be used by any Command Sequence. A "local" variable, defined by the Local command, is
Appendix 3: Command Sequence Processor Design Specifications

used only in one specific Command Sequence and will be
deleted when that sequence is completed. A variable
defined by either command may be an integer, string,
or boolean. The type is determined when a value is
assigned to it.

!Assign (variable) <varname> (%) <expression>

The Assign command assigns a value and type to the
variable (either global or local). The expression is
made up of variables, user selections, and operators.
(Although the operators are not specified at this
time, they will include addition, subtraction, and
string concatenation.)

!Test (variable) <varname>

!Relation <expression> <relation> <expression>

The Test and Relation commands are intended to be used
as commands within the if commands. The Test command
simply sets a condition flag based on the variable.
True/False results will be defined for all variable
types. The Relation command applies the relational op-
erator to two expressions—the Relation command is
TRUE if and only if the specified relation is TRUE.

!name

This command identifies a global or local name. It
may be a variable name or a Command Sequence name.
The Command Sequence will be executed or, if the vari-
able contains a string, treated as a Command Sequence
and executed. If the variable is not a string, a con-
dition flag will be set.

Miscellaneous

% <some text> %

A comment may be inserted in the Command Sequence text
anywhere a space is allowed, by surrounding the com-
ment with percent signs.

!Null

The Null command does nothing and always has the value
TRUE.
Appendix 3: Command Sequence Processor Design Specifications

!False

The False command does nothing and always has the value FALSE.

!Echo (y/n) <Y or N>

The Echo command determines whether or not the user sees the normal command feedback text at his terminal while the Command Sequence is being executed. Echo Y will result in showing the user the feedback for each command as it is executed. Echo N will result in showing the user only the Command Sequence noise words defined by the noise word command.
Appendix 4: Proposed External Design for a Process System

INTRODUCTION

This document proposes an external design for a Process System (PS). This includes the user interface — primarily what relevant AUGMENT commands there are — and a new artificial language called the Process Language (PL).

The basic construct with which we are concerned here is called a "Process Command Sequence" or "PCS". This is basically a sequence of AUGMENT commands as described in the text of statements in an AUGMENT file. PCSs are basically the same as the old Process Commands, but with several additional features.

Almost nothing is said in this document about how this proposed design might be implemented, only external specifications are proposed here. Note also that initially it would not be necessary to implement this design in full: there are several features that could be added after the initial implementation.

Not much effort has been taken to make this document easy to understand because of its role as a draft language syntax. Further revisions are expected containing examples and explanations where appropriate.

USER INTERFACE

There are two ways to invoke a Process Command Sequence: with the AUGMENT Process command and via a "User Command".

NE: The meta-language used in this section to describe the syntax of AUGMENT commands is something of a mix between CML and the meta-language used in ARC's user documentation, plus some informalities of my own devising. I hope it's understandable.

Process command:

This command, which is a universal command rather than just a BASE command, has the following syntax:

Process FSTRUCTURE (at) SSEL (CONFIRM / PARAMETERS CONFIRM)

FSTRUCTURE = the "filtered structure" that will replace STRUCTURE
PARAMETERS = see next section

Note that this is nearly the same as the old Process command with the added possibility of parameters, which are discussed in the next section.
User Commands:

There is a facility for users to define and invoke their own User Commands.

Defining User Commands:

A new command is added to the BASE subsystem with the following syntax:

Define User-command (at) SSEL(Branch) (with level) (OK/1/2) CONFIRM

NB: The "OK" above will act the same as a "1".

The indicated branch is assumed to be a properly formed PCS. In addition it must start with a "label" (see next section for the definition of a label). The label is examined and used as the associated command word.

Note that such User Commands are only "defined" for that session. In subsequent sessions with AUGMENT the User Commands will be unknown.

There is also one or more new commands in the USEROPTIONS subsystem that will provide for the definition of User Commands. User Commands defined in this manner will be recognized in all subsequent AUGMENT sessions until the user deletes/excludes the definition, e.g., by using another command in USEROPTIONS.

This USEROPTIONS feature is analogous to USEROPTIONS' Include (subsystem/program) feature.

Invoking User Commands:

After a User Command has been "defined" by any of the BASE or USEROPTIONS commands, the FrontEnd CLI will act as if "label" were a universal command at the specified level. Then any time the user inputs the proper character(s) at the base command state of any subsystem, the CLI will act exactly as if the user had input the following:

Process Branch (at) SSEL(Branch) <OK>

PROCESS LANGUAGE

Introduction

Following is a complete, formal description of the syntax of the proposed Process Language (PL).
The meta-language in which the description is written is ARC's usual version of BNF with the following additions:

- "#<foo, bar>" is equivalent to "foo $(bar foo)" and
- "$<foo, bar>" is equivalent to "[foo $(bar foo)]",

i.e., a sequence of n foos separated by n-1 bars;

- "SP" means the character with code 40B (space);
- "CA" means the character with code 04B (control-D, Command-Accept);
- "CD" means the character with code 30B (control-X, Command-Delete);
- "CH" means any character except SP, CA, or '!

"CH -> some character" means any number of characters terminated by, but not including, the specified character. Thus, the modified BNF description of Label below, "Label = CH -> '!'" says that a label is any number (greater than or equal to one) of characters terminated by (but not including) a colon.

Note that PL is a fully typed and type-checked language. PL is interpreted rather than compiled.

Some information about the semantics is included. Only things that are novel or that might be obscure are discussed. Readers are assumed to be familiar with typed, block-structured computer languages.
LevAdj = \$\{'d\'u\} [CA] / '!' [TextLHS '_ ' / TextExp ] '!';

ViewSpecs = CA / CH -> CA / '!' [TextLHS '_ ' / TextExp ] '!';

Result = ': BooleanLHS;

In the syntax for the NLSCommands, the alternatives before the last are meant to represent the old Process Commands stuff. Note that it is not quite accurate or complete, e.g., a user may have changed his/her Command-Accept character, a CD occurring almost anywhere would screw up the "parse".

The semantics of alternatives of the form
'
' [LeftHandSide '_'] [Expression] '

is as follows:

Characters are taken from the user until the "thing" the CLI is looking for is complete.

Characters are taken from the user until the "thing" the CLI is looking for is complete and the value of the user response is stored in the indicated LeftHandSide (a variable in the PL program).

The Expression is evaluated and fed to the CLI as if the user had typed it in.

Note that NLSCommands may have a Result. If an NLSCommand has a Result present, the success or failure of the execution of that command is stored in the indicated BooleanLHS and is thus subsequently available to the PL program.

ProcessCommand = '!' ( Label [ParameterList] / Declaration / Statement ) ;

Label = CH -> ':

ParameterList = '(' \$<.ID ':' TypeIdentifier, ',> ') ;

ProcessCommands that have Labels can be used as objects of Process Branch commands. If they are so used and have a ParameterList, the user will be prompted to provide the values for the parameters.

Declaration = Label [Persistence]
( TypeDec / VariableDec / ProcedureDec / RoutineDec ) ;

On-Line Programmers Management System:
Final Technical Report
56
Label = CH -> ' ';  
Persistence = "TEMPORARY" / "SEQUENCE" / "SESSION" ;

Note that Declarations may occur anywhere in a PL program. Note also that because PL programs are interpreted, the scope of a Declaration name is determined by the execution path, not lexically. A Declaration name is known and may be referenced any time between the time the interpreter has seen the declaration and the time the variable is destroyed. The Persistence determines when the variable will be destroyed:

TEMPORARY variables are destroyed when a RETURN statement is executed.

SEQUENCE variables are destroyed when a FINISH statement is executed.

SESSION variables are destroyed when an AUGMENT "session" ends.

The default Persistence is TEMPORARY.

TypeDec = "TYPE" '=' TypeSpecification ;

VariableDec = TypeSpecification [('(' '/' '=' Expression) ;

ProcedureDec = ProcedureTS '=' $ { Declaration / Statement } 

"END." ;

RoutineDec = RoutineTS ;

TypeSpecification =

ArrayTS / BooleanTS / CharacterTS / CommandWordTS / EnumerationTS / 
IntegerTS / IntervalTS / ProcedureTS / RoutineTS / SelectionTS / 
TextTS ;

OrderedTS = SelectionTS / CountTS ;

CountTS = CharacterTS / EnumerationTS / IntegerTS ;

IntgrTs = IntervalTS / IntegerTS ;

TextTS = CommandWordTS / SelectionTS / TextTS ;

RangeTS = IntervalTS / EnumerationTid ;

ArrayTS = ArrayTid / ArrayTypeConstructor;

BooleanTS = BooleanTid;

CharacterTS = CharacterTid;

CommandWordTS = CommandWordTid;

EnumerationTS = EnumerationTid / EnumerationTypeConstructor;

IntegerTS = IntegerTid;

IntervalTS = IntervalTid / IntervalTypeConstructor;

Appendix 4: Proposed External Design for a Process System

ProcedureTS = ProcedureTld / ProcedureTypeConstructor;
RoutineTS = RoutineTld / RoutineTypeConstructor;
SelectionTS = SelectionTld;
TextTS = TextTld;

ArrayTypeConstructor = "ARRAY" RangeTS "OF" TypeSpecification;
EnumerationTypeConstructor = '{ #<.ID, ',> '};
IntervalTypeConstructor = ($([]), OrderedExp ', OrderedExp (')/'));

ProcedureTypeConstructor = "PROCEDURE" [ParameterList] [ReturnsClause];

ParameterList = '(' $<.ID ': TypeIdentifier, ',> ')';

ReturnsClause = "RETURNS" '( $<TypeIdentifier, ',> ')';

RoutineTypeConstructor = "Routine" [ParameterList] [ReturnsClause];

Before a PL program can call an LIO routine in the FrontEnd or Backend, the name and calling sequence of that routine must be specified. Note that this is analogous to CML programs.

TypeIdentifier =
  ArrayTld / BooleanTld / CharacterTld / CommandWordTld / EnumerationTld /
  IntegerTld / IntervalTld / ProcedureTld / RoutineTld / SelectionTld /
  TextTld;

OrderedTld = SelectionTld / CountTld;
CountTld = CharacterTld / EnumerationTld / IntegerTld;

StringTld = CommandWordTld / SelectionTld / TextTld;

RangeTld = EnumerationTld / IntervalTld;

On-Line Programmers Management System:
Final Technical Report
RoutineTid = .ID;
SelectionTid = .ID / "SELECTION";
TextTid = .ID / "TEXT";

Selections are meant to hold AUGMENT addresses in text form, i.e., they should be convertible by 'caddexp', AUGMENT address evaluation routine, into L10 TEXT POINTERS.

Note also that when Selections are taken as pointers to nodes in an AUGMENT file tree structure, they are strictly ordered if they point to the same file. Thus there can be intervals/sequences of Selections that can be iterated over.

Statement = [Label] (AssignmentStmt / BlockStmt / BumpStmt / CallStmt / CaseStmt / EchoStmt / ExitStmt / FinishStmt / GotoStmt / IfStmt / IterativeStmt / NullStmt / RepeatStmt / ReturnStmt)

Label = CH -> ';

AssignmentStmt = #<LeftHandSide, '>', _ #<Expression, '>', ;

blockStmt = "BEGIN" $Statement "END";

BumpStmt = "BUMP" ["UP" / "DOWN"] #<CountLHS, '>', / ["NEXT" / "BACK" / "SUCCESSOR" / "PREDECESSOR"] #<SelectionLHS, '>', ;

The defaults are UP and SUCCESSOR.

Note that since Selections have order, they may be BUMPed.

CallStmt = "CALL" Call;

Call = (RoutineExp / SelectionExp) [ ( #<Expression, '>', '>', ) ];

Any of FEROUTINES, BEROUTINES, or PL PROCEDURES may be Called.

CaseStmt = "CASE" Expression "OF"
 StmtChoice
 "ENDCASE" ' ; Statement;

StmtChoice = ' ; #<RelationTail, '>', ' ; Statement;

RelationTail =
  ( #<RelationTail, '>', ' ; Statement;
  ( '=' / '<' / '>' / '<=' / '>=' / 'NOT' ) "IN" RangeTS ;

On-Line Programmers Management System: 
Final Technical Report
Note that since Selections have order, they may be compared/related using any of the comparison/relational operators.

EchoStmt = "ECHO" BooleanExp;

The value of the BooleanExp determines whether the user will see things happen in the Command Feedback Window.

ExitStmt = ( "EXITBLOCK" / "EXITLOOP" / "EXITCASE" ) [IntgrExp];

FinishStmt = "FINISH";

There is an implicit FinishStmt after the end of any PCS, whether the limits of the sequence was determined by the selections in a Process Group command or any other way.

GotoStmt = "GOTO" SelectionExp;

IfStmt = "IF" BooleanExp "THEN" Statement ["ELSE" Statement];

IterativeStmt = [ Iteration / Assignment ] [ConditionTest] "DO" Statement [ConditionTest];

Iteration = [ "FOR" OrderedLhs ] "IN" RangeIs [Increment];

Increment = ("UP"/"DOWN") [IntgrExp] / "USING" ("NEXT"/"BACK"/"SUCCESSOR"/"PREDECESSOR");

The USING can be used only if the OrderedExp in the RangeIs is of the Selection TYPE.

Assignation = "FOH" LeftHandSide `- Expression `, Expression;

ConditionTest = ("WHILE"/"UNTIL") BooleanExp;

NullStmt = "NULL";

RepeatStmt = ( "REPEATBLOCK" / "REPEATLOOP" ) [IntgrExp] / "REPEATCASE" [IntgrExp] "WITH" Expression;

ReturnStmt = "RETURN" [ ( "<Expression," ,">" ) ];

A ReturnStmt executed at that "highest" level acts as a FinishStmt.
Appendix 4: Proposed External Design for a Process System

LeftHandSide = LHSList / ArrayLHS / BooleanLHS / CharacterLHS / CommandWordLHS / EnumerationLHS / IntegerLHS / IntervalLHS / ProcedureLHS / RoutineLHS / SelectionLHS / TextLHS ;

LHSList = '(', 2#<LeftHandSide, ', ', ')';

This form may be used to store multiple return values from a Call.

OrderedLHS = SelectionLHS / CountLHS ;
CountLHS = CharacterLHS / EnumerationLHS / IntegerLHS ;
IntegerLHS = IntervalLHS / IntegerLHS ;

StringLHS = CommandWordLHS / SelectionLHS / TextLHS ;

RangeLHS = EnumerationLHS / IntervalLHS ;

ArrayLHS = .ID / IndexedReference ;
BooleanLHS = .ID / IndexedReference ;
CharacterLHS = .ID / IndexedReference ;
CommandWordLHS = .ID / IndexedReference ;
EnumerationLHS = .ID / IndexedReference ;
IntegerLHS = .ID / IndexedReference ;
IntervalLHS = .ID / IndexedReference ;
ProcedureLHS = .ID / IndexedReference ;
RoutineLHS = .ID / IndexedReference ;
SelectionLHS = .ID / IndexedReference ;
TextLHS = .ID / IndexedReference ;

IndexedReference = ArrayLHS '(', OrderedExp , ')';


OrderedExp = SelectionExp / CountExp ;
CountExp = CharacterExp / EnumerationExp / IntegerExp ;
IntegerExp = IntervalExp / IntegerExp ;

StringExp = CommandWordExp / SelectionExp / TextExp ;

RangeExp = EnumerationExp / IntervalExp ;

ArrayExp = AnyTypeExp / '(', Expression, ', ', ')';

BooleanExp = AnyTypeExp / ;
Appendix 4: Proposed External Design for a Process System

This expression will test whether or not the specified AUGMENT address actually exists.

"TRUE" / "FALSE" ;

CharacterExp = AnyTypeExp / ("FIRST"/"LAST") intervalTld / .SR1 ;

The OrderedExp associated with the IntervalTld must be of CHARACTER TYPE.

CommandWordExp = AnyTypeExp / .SR ;

EnumerationExp = AnyTypeExp / ("FIRST"/"LAST") RangeTld ;

The OrderedExp associated with the RangeTld must be of Enumeration TYPE.

IntegerExp = AnyTypeExp /

intExp ('+/-') intExp /
intExp ('#/"MOD") intExp /
'- intExp

"LEVEL" ' ( SelectionExp ')' /

The AUGMENT "level" of a statement.

(ArrayLHS/StringLHS/.SR/RangeTld) ".L" /
("FIRST"/"LAST") RangeTld /

("MIN"/"MAX") ' ( 2< intExp, ',> ')' /
"ABS" ' ( intExp ')' /

.NUM ;

IntervalExp = AnyTypeExp /
Appendix 4: Proposed External Design for a Process System

intervalExp ("+/-") intervalExp / intervalExp ("#"/"MOD") intervalExp / 
'- intervalExp 8c10j1

"LEVEL" '( SelectionExp ') / 8c10j2

The AUGMENT "level" of a statement. 8c10j2a

(ArrayLHS/StringLHS/.SR/RangeTld) ".L" / 
("FIRST"/"LAST") RangeTld / 8c10j3

("MIN"/"MAX") '( 2<IntervalExp, ',> ') / "ABS" '( IntervalExp ') / 8c10j4

.NUM ; 8c10j5

ProcedureExp = AnyTypeExp / '§ ProcedureLHS ; 8c10k

RoutineExp = AnyTypeExp / '§ RoutineLHS ; 8c101

SelectionExp = AnyTypeExp / 8c10m

("BACK"/"DOWN"/"END"/"HEAD"/"NEXT"/"ORIGIN"/
"PREDECESSOR"/"SUCCESSOR"/"TAIL"/"UP")
' ( SelectionExp ') / 8c10m1

These Expressions enable moving around AUGMENT tree-structured files. 8c10m1a

.SR ; 8c10m2

TextExp = AnyTypeExp / .SR ; 8c10n

AnyTypeExp = '( Expression ') / .ID / 
AssignmentExp / Call / CaseExp / IfExp / IndexedReference / UserExp ; 8c10o

AssignmentExp = LeftHandSide ("_":"=") Expression ; 8c10o1

Call = ( RoutineExp / SelectionExp ) ' ( $<Expression, ',> ') ; 8c10o2

Any of FEROUTINEs, BRROUTINEs, or PL PROCEDURES may be Called. 8c10o2a

CaseExp = "CASE" Expression "OF"
#ExpChoice
"ENDCASE" ': Expression; 8c10o3

ExpChoice = ' | $<RelationTail, ',> ': Expression; 8c10o3a

RelationTail =

On-Line Programmers Management System:
Final Technical Report
63
Appendix 4: Proposed External Design for a Process System

('='/'#) Expression /
('>'/'<'/'=/'<='/') OrderedExp /
['"NOT"] "IN" RangeTS ;

IfExp = "IF" BooleanExp "THEN" Expression "ELSE" Expression;

IndexedReference = ArrayLHS '[ OrderedExp '] ;

UserExp = '; (LeftHandSide '_ / TypeIdentifier) ';

The user will be interactively prompted to provide a value. The user-provided value will be evaluated according to the TYPE indicated by the LeftHandSide or TypeIdentifier.

The user will be inputting a LSEL. Thus he or she may BUG rather than type.

CNWn = Comment / NoiseWords / Message ;

Comment = '% -> '%;

NoiseWords = '{ -> ');

Message = '; -> ';

A CNWn may appear any place a space may appear. Comments are ignored. The value of a NoiseWord will appear in the Command Feedback Window enclosed in parentheses. The value of a Message will appear in the ITY Window enclosed in semi-colons.

Both NoiseWords and Messages may be variable. They both may contain: '; Expression ';. If so, the Expression is evaluated, converted to text if necessary, and shown in the appropriate window.
Glossary

CML: The Command Meta Language. The user interface of all NLS subsystems is specified in CML.

DAD: Do-All Debugger

encapsulation: There is a facility for encapsulating programs for NLS. This facility allows programs (.sav or .exe files) that were written without NLS in mind to be executed as sub-forks of NLS. Encapsulated programs may get their input from NLS files or, to a degree, interactively from the user. Similarly for the output from encapsulated programs. Currently the Meta, L10, and CML compilers are encapsulated.

fork: TENEX's term for what is most often called a "process" in computer terminology

INCLUDE statement: We have facility for "including" a group of statements from any NLS file as if that group of statements were actually present in place of the INCLUDE statement.

index, a LIBRARY: An index file is produced by the LIBRARY subsystem from a source code file. It contains a sorted list of all the global variable and procedure names in that module with pointers to their locations in the source code file. See "SysGuide".

JDAD: JOVIAL DAD

L10: An ALGOL-like language with additional string manipulation facilities. L10 is the primary implementation language of NLS.

LIBRARY subsystem: An NLS subsystem that will conditionally perform various clerical and bookkeeping chores on a collection of modules, e.g., compiling, loading, printing, indexing, constructing SysGuides. The reference manual may be found in <29151>.

Meta: A "meta-compiler" system used to produce compilers. L10, CML, and of course Meta are written in Meta.

PROGRAMS subsystem: An NLS subsystem having commands of use to programmers, e.g., Compile, Insert Procedure. Users' documentation may be found in <ArcDocumentation,Programs>.

subsystem, an NLS: NLS may be viewed as a collection of subsystems. Each subsystem has a collection of commands that are functionally related, e.g., the BASE subsystem has editing commands (and some others).
SysGuide: A sorted collection of indices (see "index" above). Typically a SysGuide will contain the indices from all the modules in the entire scope of an address space (in a fork or .sav file). A SysGuide may be used in the NLS Jump (to) Name External command.

template: See <SsSrc,Programs-Templates,>. A group of NLS statements used by the PROGRAMS subsystem Insert command, e.g.,

\begin{verbatim}
UNTIL until-clause DO
BEGIN

\end{verbatim}

templates file: An NLS file containing templates, see <SsSrc,Programs-Templates,>.
MISSION
of
Rome Air Development Center

RAOC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.