

AD-A164 026

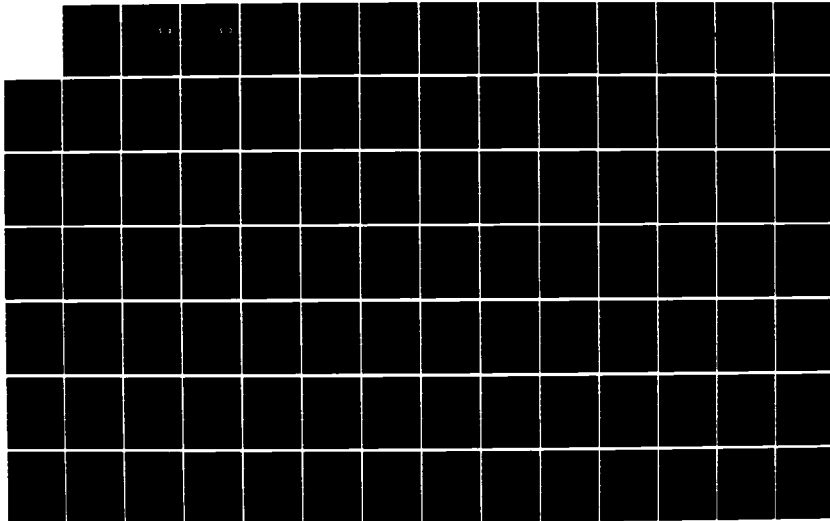
A NATURAL LANGUAGE PROCESSOR AND ITS APPLICATION TO A
DATA DICTIONARY SYSTEM(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. S A WOLFE
DEC 85 AFIT/GCS/ENG/85D-19

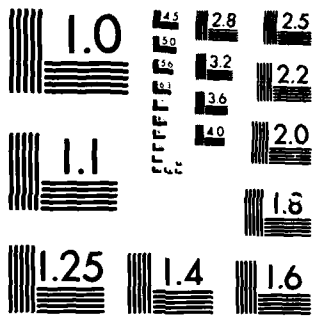
1/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

AD-A164 026



1

DTIC
 ELECTE
 FEB 13 1986
 S D

A NATURAL LANGUAGE PROCESSOR AND ITS
 APPLICATION TO A DATA DICTIONARY SYSTEM

THESIS

Stephen A. Wolfe
 Captain, USAF

DTIC FILE COPY

DISTRIBUTION STATEMENT A

Approved for public release;
 Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
 AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

6 2 1 0 079

1

AFIT/GCS/ENG/85D-19

DTIC
ELECTE
FEB 13 1986
S D D

A NATURAL LANGUAGE PROCESSOR AND ITS
APPLICATION TO A DATA DICTIONARY SYSTEM

THESIS

Stephen A. Wolfe
Captain, USAF

Approved for public release; distribution unlimited

AFIT/GCS/ENG/85D-19

A NATURAL LANGUAGE PROCESSOR AND ITS
APPLICATION TO A DATA DICTIONARY SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Stephen A. Wolfe, A.B.
Captain, USAF

December 1985

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification:	
By _____	
Distribution / _____	
Availability Codes	
Dist	Availability or Special
A-1	

Approved for public release; distribution unlimited

Acknowledgments

I would like to thank my thesis advisor, Dr. Gary B. Lamont, for his guidance throughout this effort, and my readers, Dr. (Captain) Stephen E. Cross and Captain Duard S. Woffinden, for their time and efforts. I am also indebted to the people of Rome Air Development Center/COE, especially Major Mark Stasiak and Mr. Douglas White, for their sponsorship. Finally, I wish to thank my wife, Andrea, for her helpful comments concerning this thesis and for putting up with me for the past year and a half.

Stephen A. Wolfe

Table of Contents

	Page
Acknowledgments	ii
List of Figures	vi
List of Tables	vii
List of Acronyms	viii
Abstract	ix
I. Introduction	1- 1
Thesis Objectives	1- 1
Background	1- 1
Natural Language	1- 2
The Software Development Life Cycle	1- 4
The Software Development Workbench	1- 5
Scope of Effort	1- 6
Standards	1- 7
Approach Taken	1- 8
II. Requirements Definition	2- 1
Introduction	2- 1
Automatic Programming	2- 2
System Requirements	2- 5
Grammar Constructor Requirements	2- 8
Sentence Interpreter Requirements	2-10
Data Dictionary Access Requirements	2-12

	Grammar Requirements	2-15
	Data Dictionary Content Requirements .	2-15
	SDW Interface Requirements	2-16
	Conclusion	2-17
III.	Design	3- 1
	Introduction	3- 1
	System Design	3- 4
	Natural Language Processor Design . . .	3- 6
	Data Dictionary Tool Design	3-11
	Grammar Design	3-17
	Interface Design	3-18
	Conclusion	3-18
IV.	Implementation	4- 1
	Introduction	4- 1
	System Implementation	4- 1
	CoIn Implementation	4- 5
	DDT Implementation	4- 8
	Grammar Implementation	4- 9
	Interface Implementation	4-11
	Integration into the SDW	4-12
	Conclusion	4-13
V.	Conclusion and Recommendations	5- 1
	Introduction	5- 1
	Development Summary	5- 1
	Analysis of the Current System	5- 2

Recommendations for Future Work	5- 3
CoIn	5- 4
DDT	5- 5
The Grammar	5- 6
The Interface	5- 7
Appendix A: System Data Dictionary	A- 1
Appendix B: Structure Charts	B- 1
Appendix C: Grammar Definition	C- 1
Appendix D: Implemented Grammar	D- 1
Appendix E: DDT Object Definition	E- 1
Appendix F: Interface Object Definition	F- 1
Appendix G: Test Plan	G- 1
Appendix H: CoIn User Manual	H- 1
Appendix I: DDS User Manual	I- 1
BibliographyBIB- 1
VitaVIT- 1
Source Code	Vol 2

List of Figures

Figure	Page
2.1 Automatic Programmer Data Flow Diagram	2- 3
2.2 Top-Level Requirements Data Flow Diagram . . .	2- 6
2.3 Data Dictionary System Data Flow Diagram . . .	2- 7
2.4 Construct Grammar Data Flow Diagram	2- 9
2.5 Interpret Sentence Data Flow Diagram	2-11
2.6 Access Data Dictionary Data Flow Diagram . . .	2-13
3.1 Sample Structure Chart	3- 2
3.2 System Design Data Flow Diagram	3- 5
3.3 Natural Language Processor Data Structures . .	3- 7
3.4 Sample Object Hierarchy Chart	3-13
4.1 AFIT ISL VAX-11/780 Hardware Configuration . .	4- 2
4.2 Sample Grammar Production	4-10
5.1 Example of a Valid Sentence Entered into DDS .	5- 4
5.2 Example of an Invalid Sentence Entered into DDS	5- 4

List of Tables

Table	Page
III-1 Sample Module Interface Table	3- 3

List of Acronyms

AFIT	Air Force Institute of Technology
AFIT/ENG	AFIT School of Engineering, Department of Electrical and Computer Engineering
BNF	Backus-Naur Form
CD	Conceptual Dependency
CoIn	Constructor/Interpreter (the natural language processor)
DBMS	Data Base Management System
DFD	Data Flow Diagram
DDS	Data Dictionary System
DDT	Data Dictionary Tool (the data dictionary access subsystem)
DEC	Digital Equipment Corporation
HIPO	Hierarchy plus Input, Process, Output
ISL	Information Sciences Laboratory
MIT	Massachusetts Institute of Technology
NIL	New Implementation of Lisp
SADT	Structured Analysis and Design Technique
SDW	Software Development Workbench
SDWE	SDW Executive

Abstract

The development of a human-computer interface construction and interpretation tool capable of processing English-like or natural language user input is discussed. The utility of the tool is demonstrated by using it to create the natural language interface for a data dictionary system. The data dictionary's development is also documented and is used as the overall context for the presentation.

I. Introduction

Thesis Objectives

The principal objective of this thesis investigation is the development of a human-computer interface construction and interpretation tool capable of processing English-like or natural language user input. The tool is generic in nature. That is, it has the capability to create grammars and interpret input sentences for a wide variety of applications. A secondary objective of this effort is to demonstrate the utility of the tool by using it to create a natural language interface for a useful software engineering environment.

Background

This section summarizes the background material on which the remainder of this thesis effort is based. The first part of this section provides an introduction to natural language and systems designed to interpret natural language. Next, since this thesis describes the development of a software system, a general introduction to the software development process is provided. The section concludes with a description of AFIT's Software Development Workbench (SDW). As part of this thesis effort, the developed system

was interfaced with the SDW. Numerous references are provided for the interested reader.

Natural Language. Natural languages are the languages that we speak, read, and write. Throughout our lives we are inundated with one or more natural languages. Therefore we are quite comfortable with natural languages. They seem "natural" to us. This feature is exactly what is needed in a computer environment to make it seem natural and consequently easy to use.

Because natural languages develop in a complex, constantly-changing, uncontrolled environment, they are themselves complex and ill-defined. They are "informal" systems. These are traits which make natural languages unnatural to a computer, traits which make it difficult for a computer to interpret natural language. Computers demand completely defined, "formal" systems. Because of this difficulty (and many others), the area of natural language interpretation has been widely researched (Bobrow, et al, 1977) (Harris, 1977) (Hendrix, 1977) (Hendrix, 1978) (Rich, 1984) (Schank and Riesbeck, 1981) (Waltz, 1978) (Woods, 1970).

The meaning of the term natural language when used to describe a human-computer interface is different than when used to describe a human-human interface. Human-computer natural language interfaces do not contain the richness (all

of the grammar rules or vocabulary) of, say, English. The guiding principle in building a natural language human-computer interface is to include enough of a natural language so that one can command the computer, in a "natural" way, to do all of the required functions. The interface must be complete enough so as to be a help rather than a hindrance to the user; it must be "simple" enough so that it can be interpreted by a computer within any time and space constraints imposed by an application.

A general introduction to natural language and its interpretation is provided The Handbook of Artificial Intelligence, Volume I (Barr and Feigenbaum, 1981:225-232), Artificial Intelligence (Rich, 1983:295-344), and Artificial Intelligence (Winston, 1984:291-334). In addition Barr and Feigenbaum (Barr and Feigenbaum, 1981:239-321) include an introduction to grammars and their representations and a description of several parsing techniques. They also present an overview of several natural language interpreting systems including Woods' LUNAR, Winograd's SHRDLU, Schank's MARGIE, Schank and Abelson's SAM and PAM, and SRI International's LIFER. More detailed descriptions of these and other specific systems can be found in "GUS, A Frame-Driven Dialog System" (Bobrow, 1977), "User Oriented Data Base Query with the ROBOT Natural Language Query System" (Harris, 1977), "Developing a Natural Language Interface to Complex Data"

(Hendrix, et al, 1978), "Human Engineering for Applied Natural Language Processing" (Hendrix, 1977), Inside Computer Understanding (Schank and Riesbeck, 1981:318-372), "An English Language Question Answering System for a Large Relational Database" (Waltz, 1978), and "Transition Network Grammars for Natural Language Analysis" (Woods, 1970). The LIFER system is of particular interest since it provided many of the ideas for the natural language processor of this investigation.

The Software Development Life Cycle. The software development life cycle has been characterized many different ways (Peters, 1981:8). In this thesis effort, the cycle is broken up into five phases. They are the functional requirements analysis phase, the design phase, the implementation phase, the integration phase, and the maintenance phase. As requirements change or errors are found, this cycle is executed iteratively.

During the functional requirements analysis phase, the emphasis is on "what" the system should do. These requirements are assigned to various hardware and software components during the design phase. Also during the design phase, the defined software components are refined into interacting modules. During implementation, the modules are encoded into a computer language and are tested individually and as groups. The hardware and software components are

assembled into a system and are subjected to testing as a whole in the integration phase. Finally, in the maintenance phase, the system is used and modified as necessary.

For a detailed discussion of the requirements phase, see Structured Analysis and System Specification (DeMarco, 1979). The design phase is covered in Reliable Software Through Composite Structured Design (Myers, 1975) and Software Design: Methods and Techniques (Peters, 1981). Various aspects of the the implementation phase are covered in The Design and Analysis of Computer Algorithms (Aho, et al, 1974), Fundamentals of Data Structures in Pascal (Horowitz and Sahni, 1984), and Algorithms + Data Structures = Programs (Wirth, 1976).

The Software Development Workbench. The Software Development Workbench (SDW), which resides on the AFIT Information Sciences Laboratory (ISL) Digital Equipment Corporation (DEC) VAX-11/780 computer, was conceived and designed to help the software engineer manage the inherent complexity of developing computer software. The SDW consists of "an integrated set of automated tools to assist the software engineer in the development of quality and maintainable software" (Hadfield and Lamont, 1983:171).

The original work on the SDW was done by Steven M. Hadfield for his master's thesis (Hadfield, 1982). In his thesis, Hadfield provided motivation for the development of

an interactive and automated software development environment. He maintained that such an environment should be integrated, traceable, flexible, and user-friendly. Eventually the SDW will consist of a comprehensive set of software development tools which will help the engineer throughout the entire software development cycle. While the current SDW is usable, it does not contain a complete set of integrated tools.

Currently, the tools contained in the SDW are integrated by a menu system, the SDW Executive or SDWE, which allows one to execute any of the tools. The menu system is hierarchical. First one chooses a category of tools such as DESIGN TOOLS from the top-level menu. A menu of the tools in the chosen category is then displayed from which one designates the particular tool to be executed.

Scope of Effort

The scope of this thesis effort includes the design and implementation of the natural language processor. The processor allows a software developer to define and implement a natural language human-computer interface. It allows the developer to construct a grammar by entering and modifying the productions of the grammar. It includes an interpreter which compares user input sentences against the

defined grammar and executes any code included with the grammar productions. The processor is generic in nature and can be used to create and use a grammar for any application domain. As part of this effort the natural language processor is integrated into the SDW.

The second area that is included within the scope of this development is the design and implementation of a data dictionary system and particularly its natural language front-end. The data dictionary system is used primarily as an extended example to demonstrate the usefulness and usability of the natural language processor, but it is also meant to be a useful tool.

Standards

This section first presents standards associated with the natural language processor and then presents standards associated with the data dictionary system. Since the emphasis is on the natural language processor, the standards directly associated with it are of greatest importance.

The natural language processor should allow a software engineer to easily construct a natural language human-computer interface to a software system. If it does, then this thesis effort should be considered a success. To fulfill this goal the grammar constructor needs to provide all of the functions necessary to create and modify a

grammar, the external interfaces to these functions should be consistent, and the interpreter should be able to correctly parse grammatical sentences for any grammar created with the processor.

The sentence interpreter should be able to parse quickly enough so as to not annoy the application user. By providing positive feedback to the user to assure him/her that the program is running and performing the desired task, this time can be extended from a few seconds to perhaps a minute or more.

The data dictionary system should allow its user to manage all of the data associated with a software development effort. Not only does the data dictionary system need to provide for the storage of this data, but its human-computer interface needs to include the functionality necessary to enter and modify the data.

Approach Taken

The software development cycle as described in the Background section is followed in this development. Throughout the chapters that follow, the natural language processor is treated as a subsystem of the data dictionary system. Doing so simplifies the structure of this thesis. Also, when the natural language processor is used to

implement a human-computer interface, it becomes a subsystem of the application.

The next chapter, Chapter II, describes the functional requirements analysis phase of this effort. The design phase is the subject of Chapter III, and the implementation phase is the subject of Chapter IV. To conclude, Chapter V summarizes this development effort, presents an evaluation of the developed systems based on the standards of the previous section, and enumerates a set of recommendations for follow-on work to this effort.

Complete documentation sets, including a data dictionary, structure charts (described in Chapter III) and various other design documents, a test plan, and user manuals for both the natural language processor and the data dictionary system, are included as appendices to this thesis. Volume 2 contains complete source code listings.

II. Requirements Definition

Introduction

This chapter presents the functional requirements used in this thesis effort. First very high level, "blue sky" requirements are described. Then the overall requirements of the data dictionary system are presented. After defining the high-level requirements, the more detailed requirements of the various subsystems are discussed.

The requirements are presented in the form of data flow diagrams (DFDs). DFDs were chosen for use in this effort because of their simplicity. Structured Analysis and Design Technique (SADT) charts (Peters, 1981:63-64), another possible representation, show more information than do DFDs, but are correspondingly more difficult to create, maintain, and understand.

DFDs consist of four basic elements: processes, data flows, data stores, and sources/sinks. Processes, which are represented by circles, transform data. That is, they modify their inputs in some way to produce their outputs. Data flows, as their name implies, are paths along which information moves between the other three element types. Data flows are represented by arrows. Data stores are files or data bases. They are represented by a line segment or by two parallel line segments. Finally, sources and sinks,

which are represented by rectangles, are entities outside of the system which originate and receive data respectively. Each element of a DFD has associated with it a label which describes that element. For an excellent description of the mechanics and use of data flow diagrams, see Structured Analysis and System Specification (DeMarco, 1979).

In addition to the DFDs of this chapter, Appendix A contains a system data dictionary which provides more information about each of the elements contained in the DFDs.

Automatic Programming

It would be nice to have an automatic design and programming system that, given a set of inconsistent, incomplete, and ambiguous requirements, could query the user to resolve these problems and then generate, modify as requested, optimize, and fully document a program which meets the requirements (Figure 2.1). Such a system should allow input in whatever mode is most comfortable to the user, including written and spoken natural language, graphics, menu selections, examples, and mathematical formulae. Its set of output modes, for responses and queries to the user, should be similarly varied and should be user selectable. That is, if the output mode chosen by the system does not include the mode desired by the user,

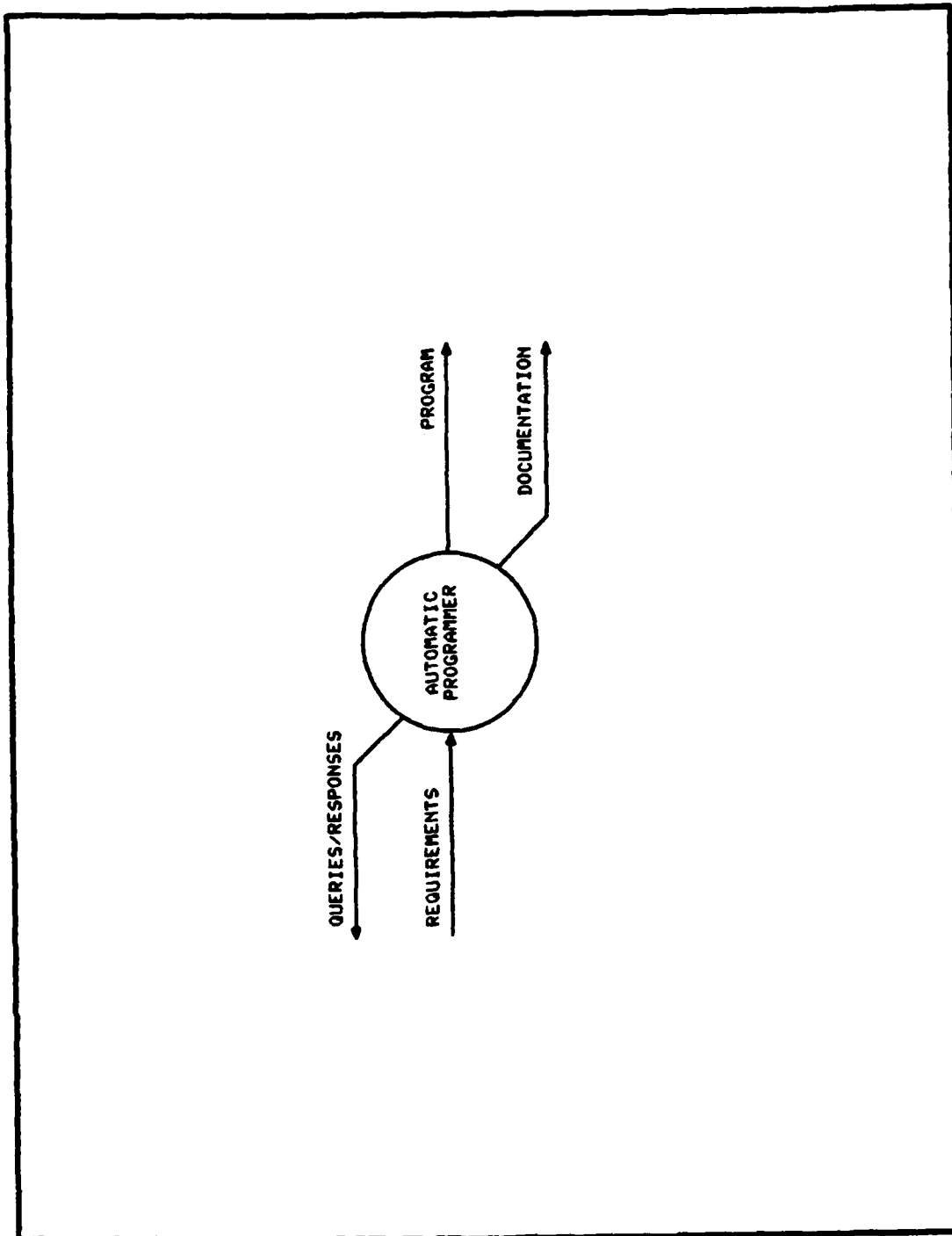


Figure 2.1. Automatic Programmer Data Flow Diagram

he/she should be able to specify the use of additional modes.

The documentation generated by such a system should include all documentation normally generated during the software development life cycle including, but not limited to, requirements specifications, high- and low-level design specifications (including the algorithms and data structures used), source code listings, a test plan, test specifications, and test procedures, including an extensive set of test cases. In addition, when desired by the user, the system should be able to provide correctness verification of any portion of the generated code.

All code generated by an automatic programmer should be traceable through the design to the requirements level. The same is true of the test procedures; all tests should be traceable to the requirements that they validate.

In addition, the generated system should be user-modifiable at any level--requirements, design, or implementation. The user should have ultimate control, but the system should recognize potential problems and advise the user as appropriate.

The Handbook of Artificial Intelligence, Volume II (Barr and Feigenbaum, 1982:295-379) includes an excellent introduction to automatic programming as well as an overview of the recent research in the field.

System Requirements

The previous section described several ideas about what a computer, via an automatic programming system, should be able to do. A system which can generate quality software directly from requirements without extensive intervention by a human programmer is probably many years in the future, if it is even possible. A possibly more practical approach, given today's technology, is to concentrate on developing a set of integrated and automated tools which aid the human engineer in developing software. This is the approach taken in the SDW and is among the justifications for this thesis effort.

This section describes one such tool in terms of its functional requirements: the data dictionary system, DDS, developed for this thesis effort. Figure 2.2 shows a top-level model of the system. This model illustrates the scope of DDS: it accepts natural language input sentences from the user, interprets the sentences as commands (using the natural language processor), and retrieves information from and/or modifies the information in the data dictionary.

Figure 2.3 breaks DDS down into its major subsystems, the grammar constructor and sentence interpreter, which together comprise the natural language processor, and the data dictionary access process. The remainder of this section discusses the more detailed requirements of these

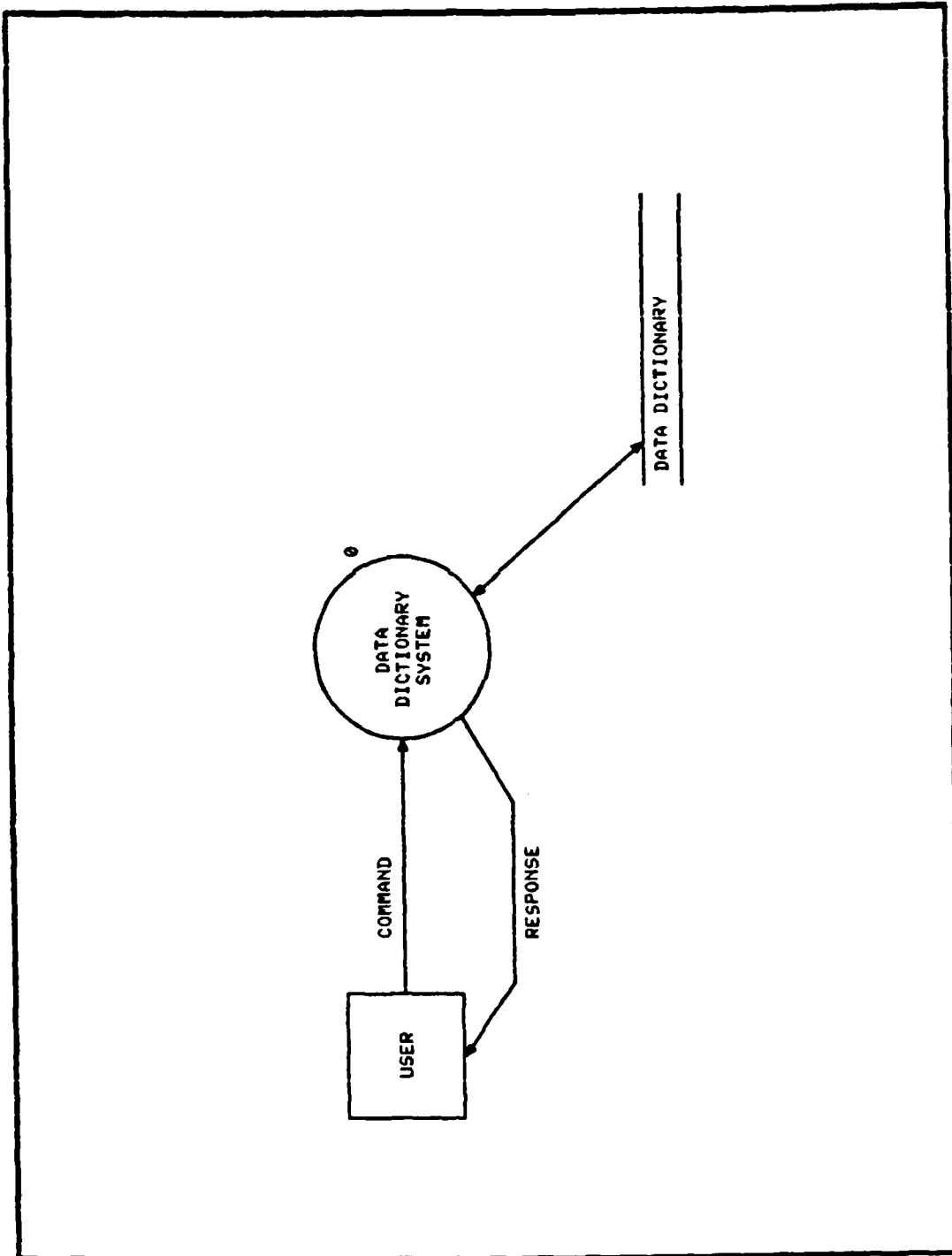


Figure 2.2. Top-Level Requirements Data Flow Diagram

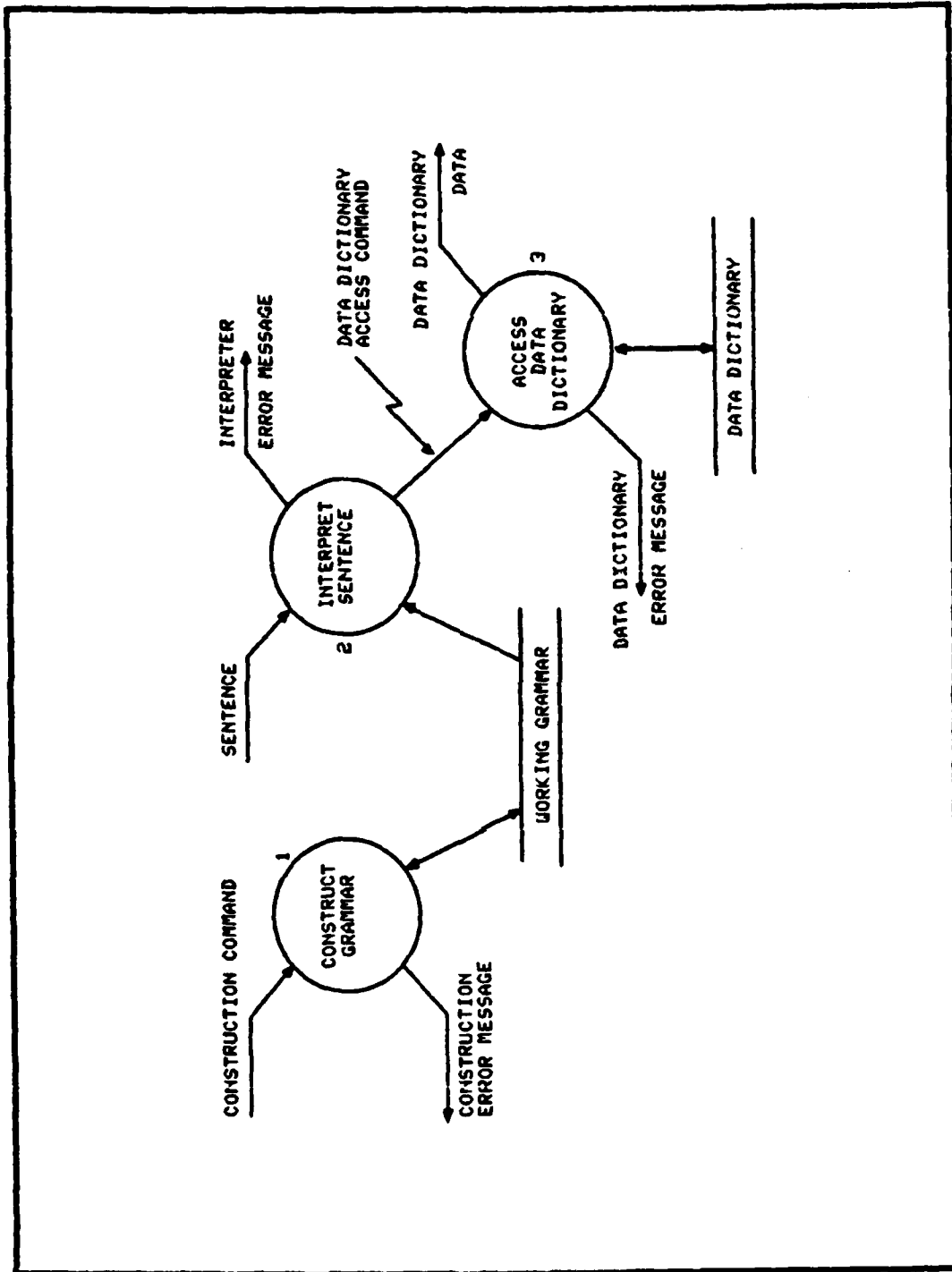


Figure 2.3. Data Dictionary System Data Flow Diagram

subsystems followed by a discussion of DDS's grammar requirements, its data dictionary content requirements, and, finally, its SDW interface requirements.

Grammar Constructor Requirements. The grammar constructor allows the user to define a set of production rules which collectively define a grammar. Figure 2.4 depicts the data flow for the grammar constructor. As is shown in the diagram, the grammar constructor consists of three subfunctions: SAVE GRAMMAR, LOAD GRAMMAR, and MODIFY GRAMMAR. Each of these processes is discussed in turn.

When a save command is entered by the user, the SAVE GRAMMAR process stores the current working grammar to a permanent file. This is necessary so that the grammar does not have to be created each time it is needed. The file must be in a format that allows the grammar to be retrieved into working storage.

Since the grammar is stored in a file, a method for retrieving the contents of the file is needed. This is the function of the LOAD GRAMMAR process. If a grammar is already in working storage, then it is overwritten by an incoming grammar.

The MODIFY GRAMMAR process consists of several functions which allow a user of the constructor to define the rules of a grammar. To insure that the grammar environment is in the proper condition before a grammar is

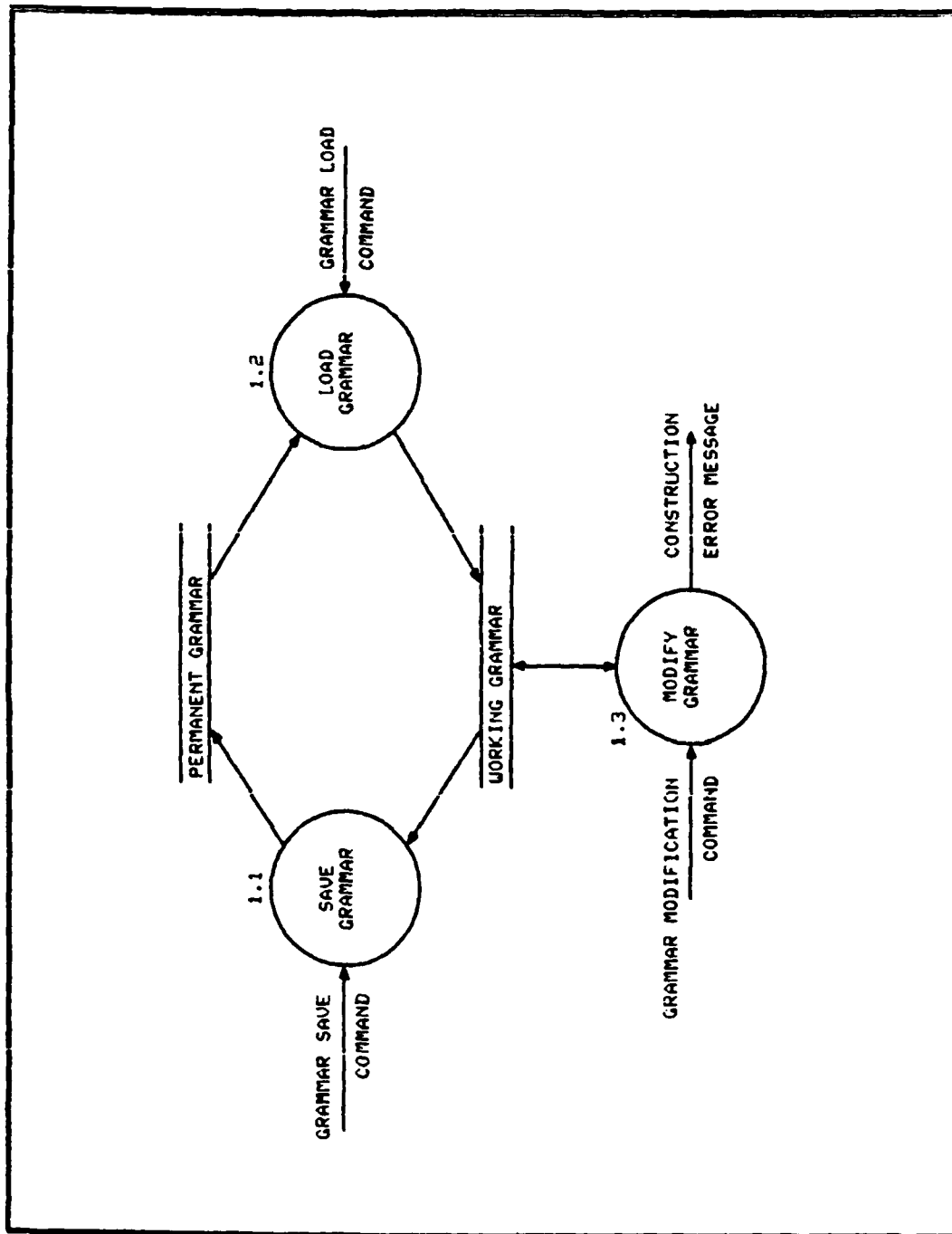


Figure 2.4. Construct Grammar Data Flow Diagram

defined, an initialization function is needed. To allow the user to initially enter the grammar's production rules, a create function is needed. To allow the user to determine what production rules exist, a list function is needed. To allow the user to view the existing production rules, a display function is needed. To allow the user to modify existing production rules, a modification function is needed. Finally, to allow the user to remove production rules which are no longer needed, a delete function is needed.

The MODIFY GRAMMAR process should be able to determine if an invalid request has been entered. If it detects an invalid request, it generates an error message.

Sentence Interpreter Requirements. The sentence interpreter has two functions: a parsing function and a command generation function. The parsing function, PARSE SENTENCE in Figure 2.5, provides the capability to determine if an input sentence is valid within the defined grammar. The parsing function also provides the capability to inform the application user that it cannot parse an input sentence. To make the system more "user-friendly", the parser should, if it is unable to parse a sentence, attempt to show the user where in an input sentence an error occurred. It should also offer suggestions as to how to correct the problem.

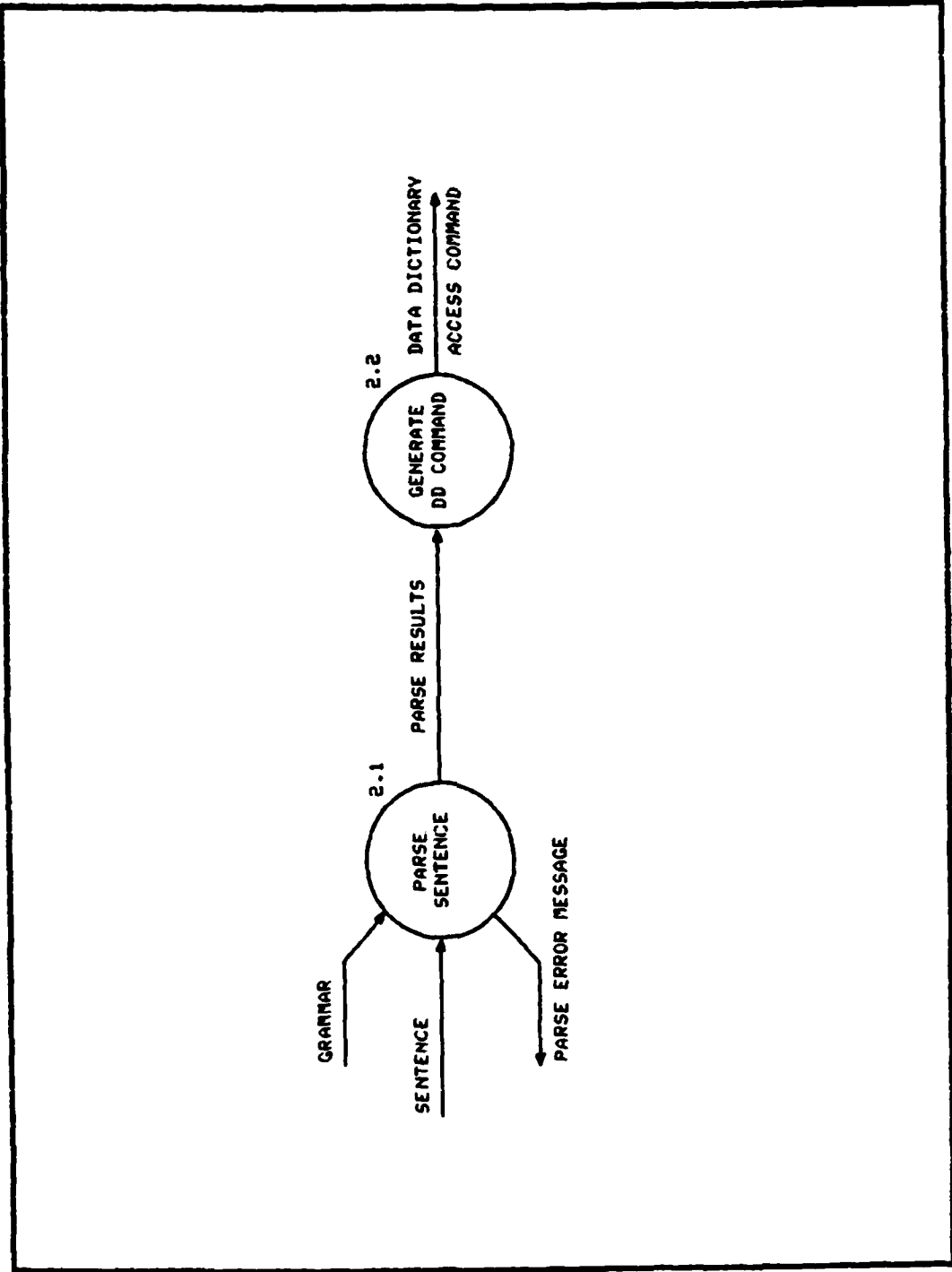


Figure 2.5. Interpret Sentence Data Flow Diagram

The command generation function, GENERATE DD COMMAND in Figure 2.5, provides the capability to syntactically translate the results of a valid sentence parse into the proper execution commands of the application tool (the data dictionary access process in this case).

Data Dictionary Access Requirements. The data dictionary access process executes the commands sent from the sentence interpreter. It provides the means to update and display the data stored in the data dictionary (Figure 2.6). Therefore, it must be "complete" in the sense that it have full access to all the information stored in the data dictionary file. It must also maintain the "consistency" of the data in the data dictionary. For instance, if a module entry is deleted from the dictionary, then ALL references to that module need to be deleted. If this is not done by the access process, the information in the data dictionary can easily become inconsistent.

The UPDATE DATA DICTIONARY process shown in Figure 2.6 consists of several functions which allow the user of DDS to define the information to be stored in the data dictionary. To allow the user to define new entries in the data dictionary, an addition function is needed. To allow the user to change existing information, a modification function is needed. A reinitialization function which resets the information in an entry to its initial state can be handy,

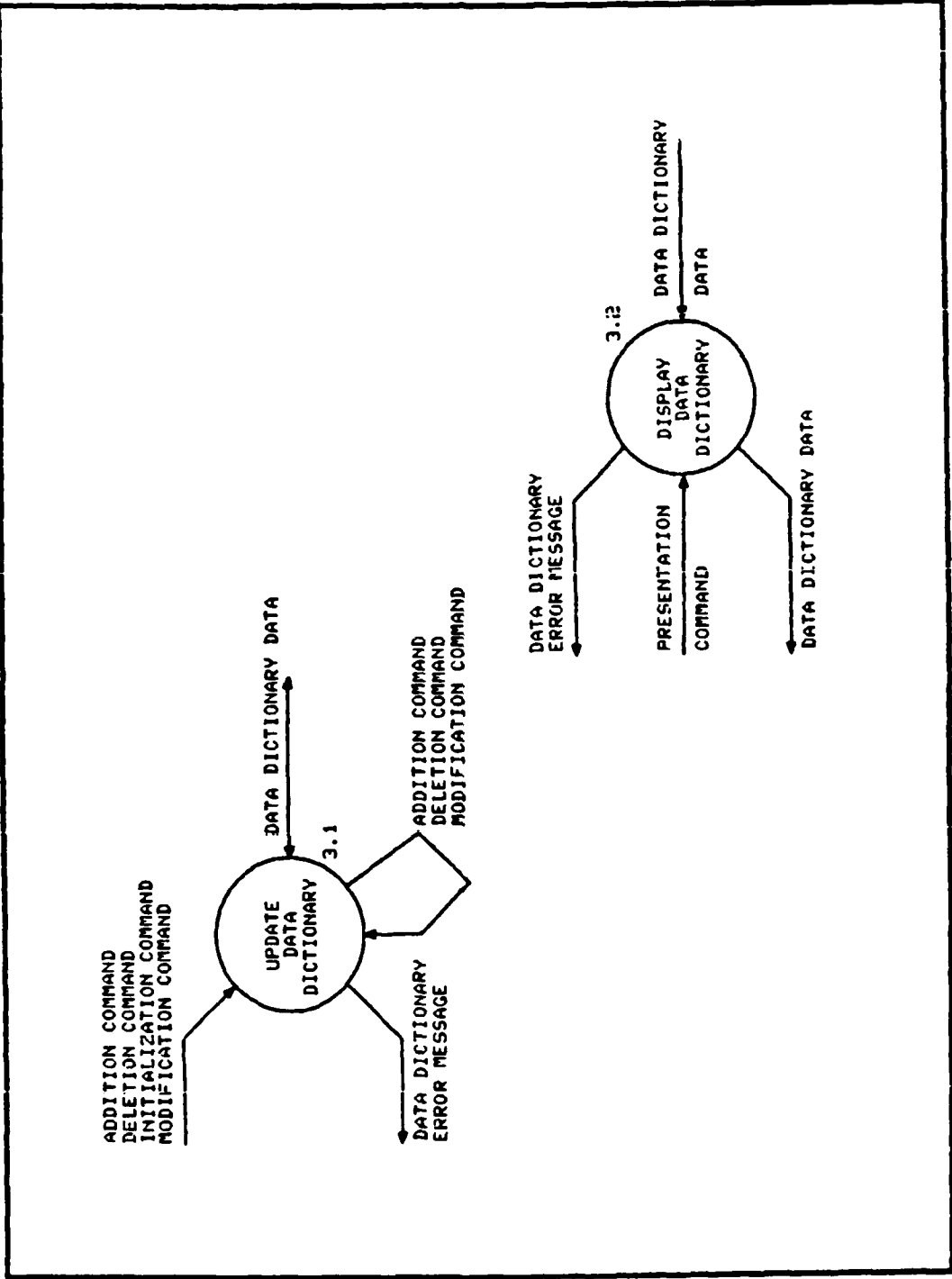


Figure 2.6. Access Data Dictionary Data Flow Diagram

especially when the modifications to be made to the entry are extensive. Finally, a deletion function, which allows the user to remove an entry from the data dictionary, is needed. As is shown in Figure 2.6, the UPDATE DATA DICTIONARY process can generate its own update command. This is necessary to enforce the data consistency discussed in the previous paragraph. If the information requested to be modified is not contained in the dictionary, this process displays an error message.

The second process shown in Figure 2.6, DISPLAY DATA DICTIONARY, retrieves, formats, and presents the information stored in the data dictionary. Again, if the information requested is not contained in the dictionary, an error message is displayed.

The data dictionary access process should be independent of the natural language interpreter. That is, it should be usable without the natural language front-end. Therefore, it needs to implement all the requirements of DDS except for those that deal specifically with the natural-language human-computer interface. By making the actual data dictionary tool independent of the human-computer interface, prototypes of various kinds of interfaces and various designs of natural language interfaces can be developed and easily tested with the

access tool; then one or more can be selected and fully developed.

Grammar Requirements. The main requirements of a grammar in a natural language human-computer interface are that it be functionally "complete" for the application environment and that it be flexible. A functionally complete grammar provides at least one way for the application user to specify each function of the application. A flexible grammar provides multiple ways to specify those functions of the application that are "naturally" specified in multiple ways. Flexibility enhances ease of use.

Since DDS is to have a natural language human-computer interface, its grammar should be functionally complete and flexible. DDS's grammar should allow sentences which specify each of the operations provided by the data dictionary access mechanism. The grammar should allow access to any of the entries contained in the data dictionary. The grammar should provide for needed meta-operations, that is, operations that deal with the environment (such as exiting from the system) vice the information in data dictionary.

Data Dictionary Content Requirements. A data dictionary needs to provide storage for the information needed by an engineer during the entire software development

process. AFIT/ENG's Development Documentation Guidelines and Standards (AFIT/ENG, 1984) was written with the intent of standardizing the composition of this information. The contents of the data dictionary should be consistent with this document since this is an AFIT thesis project.

AFIT/ENG's documentation standard specifies several data dictionary entry types for each of the software life cycle phases. For the functional requirements analysis phase, it specifies three entry types: Activity, Data Element, and Alias. Process, Parameter, and Alias entries are specified for the design phase. The implementation phase requires Module and Variable entry types. No entries are specified for the later life cycle phases as the documentation standard is not yet complete. For each entry type, the standard describes a set of data elements which should be included in the data dictionary entry.

The data dictionary should provide for all of these entry types. As the documentation standard is updated and completed, the data dictionary contents should also be revised to remain consistent with it.

SDW Interface Requirements. Since both the data dictionary system and the natural language processor are tools which may be of help to the software developer, they should both be interfaced into the SDW. Doing so makes them easier to access and, one hopes, easier to use.

Conclusion

This chapter has presented and attempted to justify the results of the functional requirements analysis phase of this thesis effort. As a prelude to the discussion of the requirements, the symbology of data flow diagrams was introduced. The concept and feasibility of an automatic programming system was also briefly discussed. The next chapter describes the design phase of this effort.

III. Design

Introduction

In this chapter, the design of the natural language processing system, CoIn (Constructor/Interpreter), and the data dictionary system, DDS, are described and justified. The description begins with the overall functional system design of DDS. Following this, more detailed functional descriptions of CoIn's design, the data dictionary tool's (DDT) design, the grammar's design, and the grammar-to-DDT interface's design are given.

The system-level design is presented in the form of a data flow diagram (see Chapter II for a description of DFDs). The lower-level designs are in the form of structure charts which are included as Appendix B. Figure 3.1 shows an example of a structure chart. Structure charts depict the modules of a program as rectangular boxes. Each box is labeled with the name of the module it represents. The boxes are interconnected with arrows from the invoking to the invoked modules. Each arrow is labelled with a number which corresponds to a number in an accompanying table. The tables tell what information is passed between the modules. As an example, Table III-1 corresponds to the structure chart of Figure 3.1.

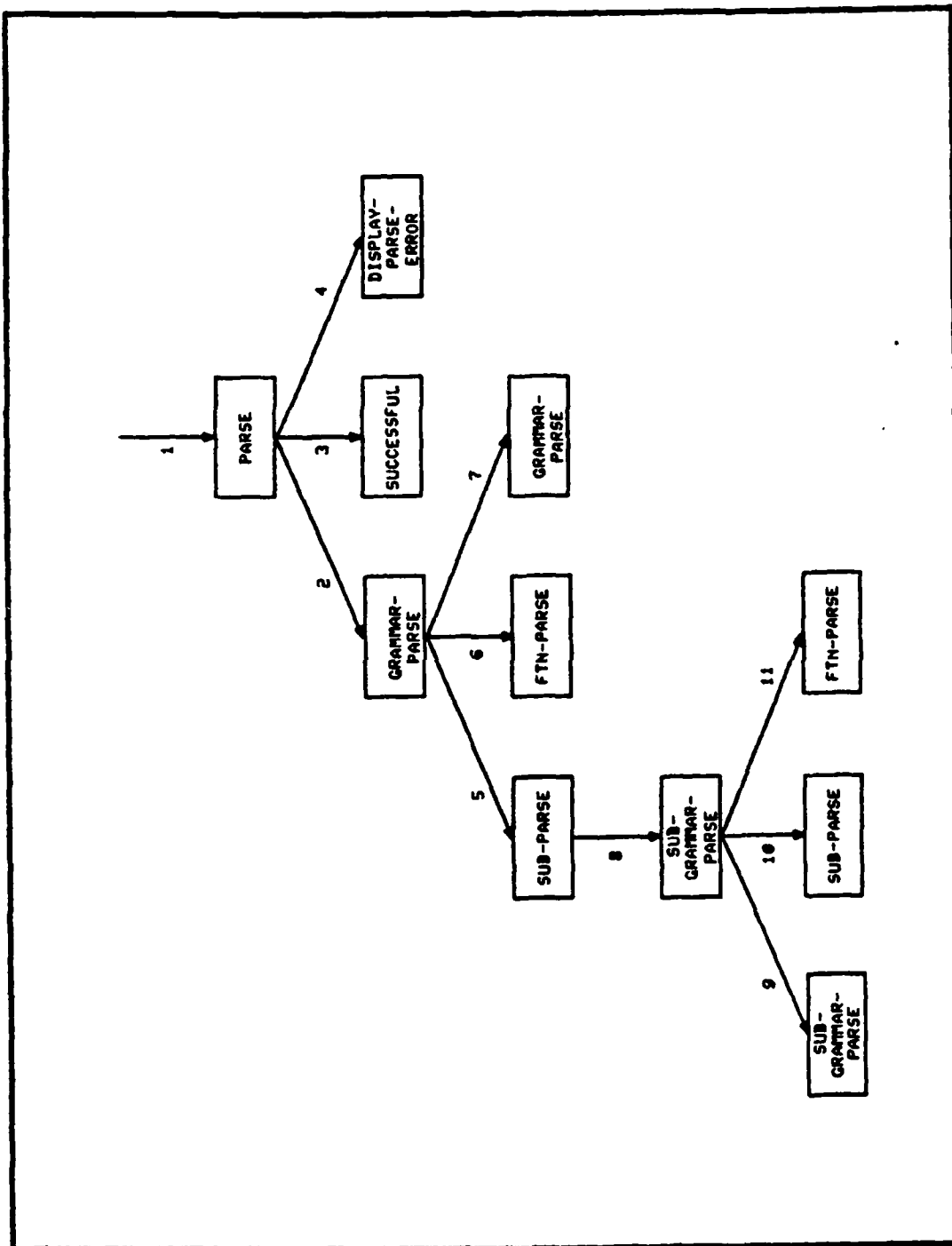


Figure 3.1. Sample Structure Chart

parse Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	sentence grammar	data data		
2	sentence sub-grammar-list	data data	result	data
3	result	data	successful parse?	flag
4	(none)			
5	sentence sub-grammar	data data	result	data
6	sentence function	data data	result	data
7	cdr*(sentence) sub-grammar-list	data data	result	data
8	sentence sub-grammar-list	data data	result	data
9	cdr*(sentence) sub-grammar-list	data data	result	data
10	sentence sub-grammar	data data	result	data
11	sentence function	data data	result	data

* The cdr function returns a list of all but the first element of a list passed to it. For instance, the cdr of the list (a b c) is the list (b c).

Table III-1. Sample Module Interface Table.

Structure charts were used to document the design because their notation is simple and therefore easy to use and understand, and because the SDW provides a tool (SYSFL) which partially automates their generation. Other notations which can be used to document software design include Leighton diagrams, Hierarchy plus Input, Process, Output (HIPO) charts, and structure trees. Again, for the reasons noted above, structure charts were chosen for use in this thesis effort. For more information about these design documentation methods, see Software Design: Methods and Techniques (Peters, 1981:44-62).

System Design

The design of DDS is divided into four functional elements: CoIn, DDT, the grammar, and the interface. As shown in Figure 3.2, CoIn consists of the GRAMMAR CONSTRUCTOR and SENTENCE INTERPRETER processes, DDT consists of the DATA DICTIONARY ACCESS process and its associated data store, the grammar consists solely of the GRAMMAR store, and finally, the interface consists of the INTERFACE process and its data store.

The EXECUTION COMMAND shown in Figure 3.2 is built into the grammar and is sent to the INTERFACE process by the SENTENCE INTERPRETER process at the end of a successful sentence parse. Similarly, the DATA STORAGE COMMANDS are

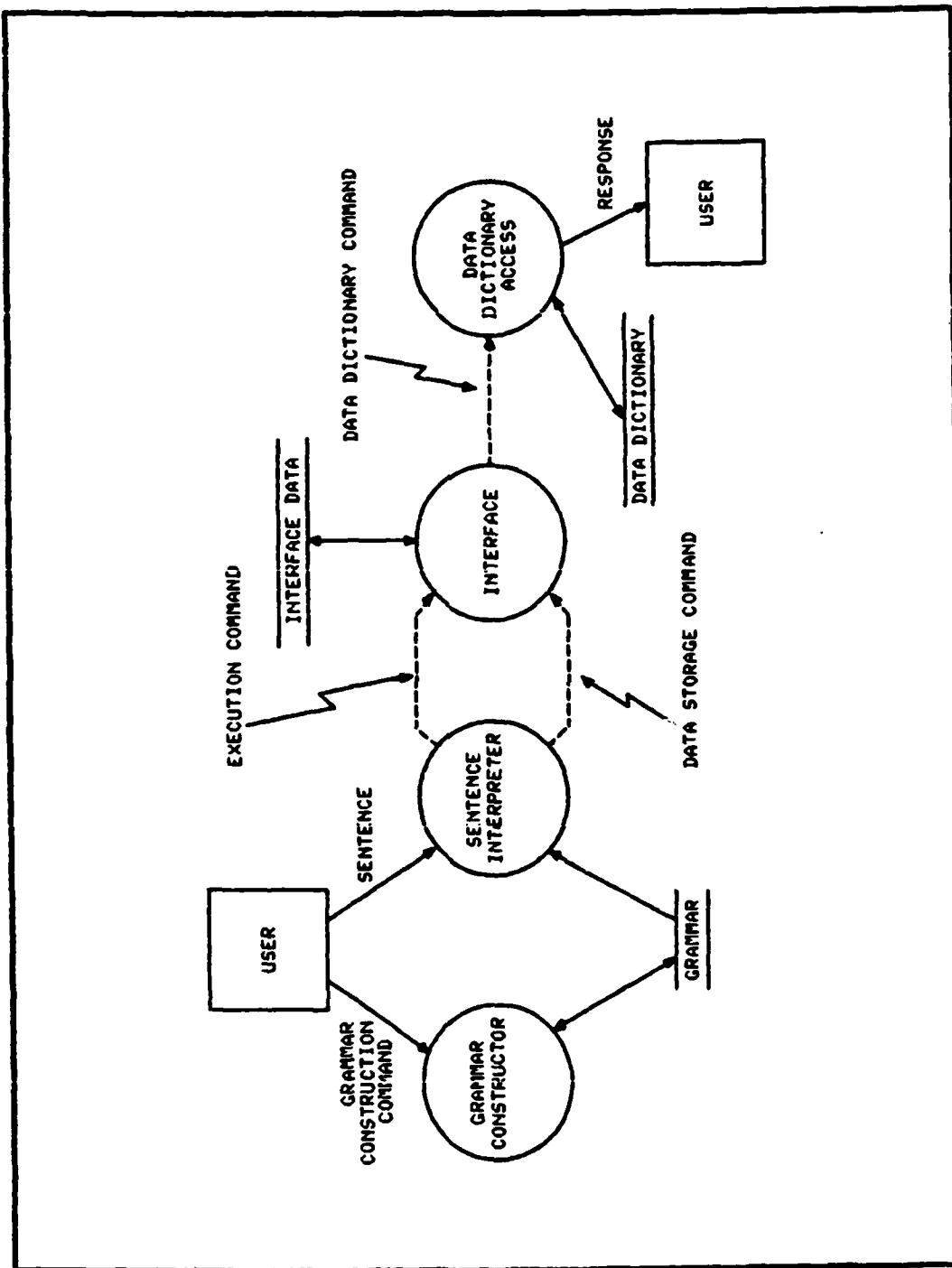


Figure 3.2. System Design Data Flow Diagram

built into the grammar and are executed by the INTERPRETER process as the input sentence is traversed. These commands tell the INTERFACE process what actions are desired and upon which data dictionary items these actions should be performed. The DATA DICTIONARY COMMANDS are generated by the INTERFACE process upon receipt of the EXECUTION COMMAND. They are based on the information in the INTERFACE DATA data store and are sent to the DATA DICTIONARY ACCESS process to be acted upon.

Natural Language Processor Design. As suggested by the requirements in Chapter II, the natural language system consists of two distinct subsystems: a grammar constructor and a sentence interpreter. The design of both of these is described in this section, but first the data structures used to represent the grammar are defined.

Data Structures. The first decision in the design of CoIn was the form of the data structures in which the grammar is maintained. Three structures were defined: the production record, the function record, and the meta-symbol list (Figure 3.3). These structures and their purposes are described in the following paragraphs.

Production Records. Production records are used to store the productions of a grammar and consist of three fields. The first field is the element which the current word of the input sentence must match for a parse to

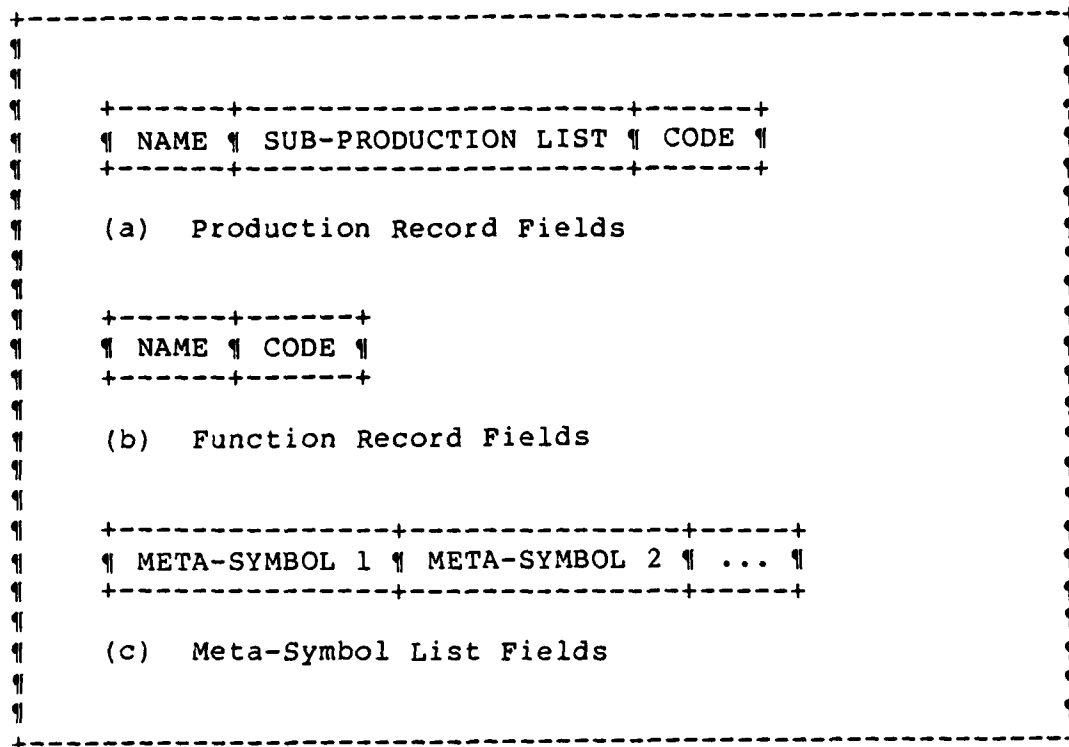


Figure 3.3. Natural Language Processor Data Structures

proceed. This could be a literal, which the word must match exactly, the name of a sub-production record to be used as the grammar in a recursive call to the parse routine, the name of a function record to apply to the current word, an end-of-production marker which always matches but does not consume any of the input sentence, or an end-of-sentence marker which matches the end of a sentence.

The next member of a production record is a list of possible sub-productions which can be used to continue the parse should the current word parse successfully. Each of

these sub-productions are identical in structure to the parent production record. This eases the parsing task by allowing recursion.

The third and final production record element is a piece of executable source code to be invoked if the production is successfully traversed.

Function Records. The second structure needed by CoIn is called a function record. Function records consist of a name and a piece of executable source code which is used to determine if the word passed to a function is a member of the function's domain. If it is, then the parse of the word is considered successful.

The Meta-symbol List. The meta-symbol list structure is simply a list of the names of a grammar's production and function records. These names are called meta-symbols because they are names and not terminal symbols of the grammar. The meta-symbol list is used to keep track of the meta-symbols that have been defined so they are not accidentally redefined and so they can be easily stored by the grammar save routine.

The Constructor. The grammar constructor consists of eight subprograms which are used to build the structures described above. These include routines to initialize a grammar, create production records, create function records, add productions to an existing production record, modify an

existing function record's executable code, destroy production records, destroy function records, and save a grammar. Each of these subprograms is described below.

The grammar initialization subprogram (initialize-grammar) destroys all of the production records and function records of the current working grammar. It also empties the list of meta-symbols. This subprogram requires no input parameters.

The grammar constructor subprogram that creates production records (create-production) takes as input the name of the production record to be created and, optionally, a piece of executable code to be executed whenever the production record is successfully traversed by the interpreter. Before attempting to create a production record, this subprogram confirms that the name passed to it has not already been used.

The grammar constructor subprogram that creates function records (create-function) is similar to the one that creates production records. It too requires a name which it assigns to the function and an optional piece of executable code. It too confirms that the name passed to it has not already been used.

The subprogram that adds productions to a production record (add-production) requires three input parameters: the grammar production, a piece of executable code to be

executed whenever the production is successfully traversed by the interpreter, and the name of the production record that the production is to be added to. This subprogram first determines whether the name passed to it is a defined production record. If it is, then the subprogram adds the production and the code to the record. Otherwise, it displays an error message.

The subprogram that modifies the executable code of a function record (modify-function) requires the new code and the name of function to modify as input parameters. This subprogram determines whether the name is a defined function record and, if it is, replaces the old code with the new. Otherwise, if the name is not a defined function, it displays an error message.

The production record and function record destruction subprograms (destroy-production and destroy-function, respectively) purge the production or function record whose name matches their input parameter. If the production or function record does not exist, then an error message is displayed.

The grammar save subprogram (save-grammar) stores all of the production records and function records of the current grammar to a disk file. It also stores the list of meta-symbols. This subprogram requires no input parameters.

The Interpreter. The sentence interpreter consists of a single user-callable routine. The user provides an input sentence and a grammar to the interpreter. It attempts to match the sentence to the grammar. As a successful parse continues, the interpreter executes the associated code built into the grammar. If the interpreter is unable to successfully parse a sentence, it displays an error message on the user's terminal screen.

As was discussed in the section on data structures, the interpreter uses recursion to traverse the grammar. The objective was to model the interpreter after the data structures in order to make the code as simple and easy to understand as possible.

Data Dictionary Tool Design. DDT was designed from an object-oriented viewpoint. Goldberg defines an object as a "uniform representation of information that is an abstraction of the capabilities of a computer" (Goldberg, 1984:76). An instance of an object has associated with it a set of memory locations, called instance variables, and a set of operations, called methods, which can access the instance variables. An object's instance variables can be accessed only through its methods. This trait forces a well-defined interface to the information stored within the instance. Each "type" of object has associated with it a schema which defines the information that can be stored.

The set of objects which are instances of a particular schema is called a class. For more information about object-oriented programming, see Smalltalk-80, The Language and Its Implementation (Goldberg, 1984:76-80) and Smalltalk-80, The Interactive Programming Environment (Goldberg and Robson, 1983:6-9).

Object Classes. This subsection defines the object classes which make up the data dictionary. Appendix E provides a set of diagrams which show the hierarchical structure of the high-level data dictionary object classes. In these diagrams, solid lines indicate that the lower level class is explicitly part of the higher level class. Dashed lines are used when the lower level class is implicitly part of the higher level class. Figure 3.4 is an example of one of these diagrams. In this example, the PROCESS object class is only implicitly a part of the DESIGN object class. The DESIGN class actually contains a reference to the SLOT class in which is stored a set of pointers to instances of the PROCESS object class. Appendix E also includes a detailed description of each of the defined object classes including their corresponding methods. Appendix B contains structure charts for each of the defined high-level methods. Structure charts were not included for the low-level methods (i.e. methods which retrieve or set the value of a single instance variable) because of their simplicity. The

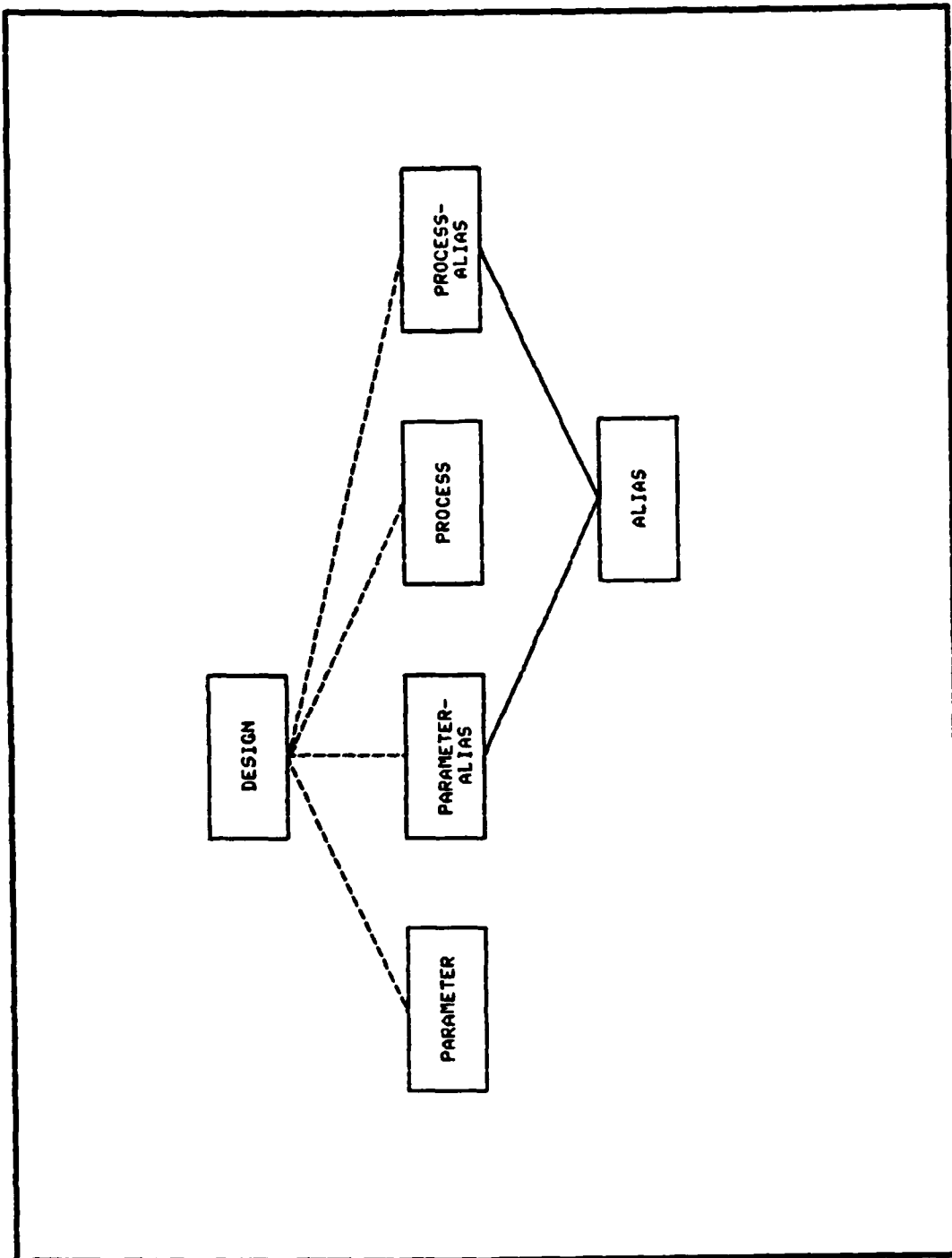


Figure 3.4. Sample Object Hierarchy Chart

remainder of this subsection presents a textual description of the object classes.

ACTIVITY. This class corresponds to the Functional Requirements Analysis Phase 'Activity' data dictionary entry.

ACTIVITY-ALIAS. This class corresponds to the Functional Requirements Analysis Phase 'Alias' data dictionary entry for activities.

ALIAS. This class is a common component of the ACTIVITY-ALIAS, DATA-ELEMENT-ALIAS, PARAMETER-ALIAS, and PROCESS-ALIAS object classes. It consists of the instance variables common to all of these objects.

ASSOC. This is a primitive-level object class used to define the ASSOC data type and its operations. The ASSOC data type is similar to Lisp's association list. An association list consists of a list of pairs. The first element of the pairs can be searched by using the 'assoc' function. The 'assoc' function returns the matching pair or NIL if no match is found.

ATOM. This is a primitive-level object class used to define the ATOM data type and its operations. The ATOM data type is similar to Lisp's atom.

DATA-ELEMENT. This class corresponds to the Functional Requirements Analysis Phase 'Data Element' data dictionary entry.

DATA-ELEMENT-ALIAS. This class corresponds to the Functional Requirements Analysis Phase 'Alias' data dictionary entry for data elements.

DATE. This is a primitive-level object class used to define the DATE data type and its operations.

DESIGN. This is a high-level class which is used to keep track of all the aliases, parameters, and processes in a program design. It also is used to maintain a list of the main processes (i.e. those processes which are not invoked by another process in the design) of a program design.

ENTRY. This is a low-level object class which is used to define instance variables which contain the version number and date of entry of the object instances in which it is included. This class is included as a subclass of the ACTIVITIES, ALIAS, DATA-ELEMENT, DESIGN, IMPLEMENTATION, PARAMETER, PROCESS, and REQUIREMENTS object classes.

HEADER. This is a low-level object class which is used to define instance variables which contain the name, type, and project name of the object instances in which it is included. This class is included as a subclass of the ACTIVITIES, ALIAS, DATA-ELEMENT, DESIGN, IMPLEMENTATION, PARAMETER, PROCESS, and REQUIREMENTS object classes.

IMPLEMENTATION. This is a high-level class which is used to keep track of the modules and variables of a program. It is also used to maintain a list of main modules (i.e. those modules which are not invoked by another module in the implementation) of a program implementation.

MODULE. This class corresponds to the Implementation Phase 'Module' data dictionary entry.

LIST. This is a primitive-level object class used to define the LIST data type and its operations. The LIST data type is similar to Lisp's list.

PARAMETER. This class corresponds to the Systems Design Phase and Detailed Design Phase 'Parameter' data dictionary entries. It is also included as a subclass of the DATA ELEMENT and VARIABLE object classes.

PARAMETER-ALIAS. This class corresponds to the Systems Design Phase and Detailed Design Phase 'Alias' data dictionary entries for parameters.

PROCESS. This class corresponds to the Systems Design Phase and Detailed Design Phase 'Process' data dictionary entries. It is also included as a subclass of the MODULE object class.

PROCESS-ALIAS. This class corresponds to the Systems Design Phase and Detailed Design Phase 'Alias' data dictionary entries for processes.

PROJECT. This is the highest-level object class. It is used as a pointer to the REQUIREMENTS, DESIGN, and IMPLEMENTATION object instances of a project.

REQUIREMENTS. This is a high-level class which is used to keep track of the activities, data elements, and aliases of a requirements analysis.

SLOT. This is a low-level class which is used to store pointers to the primitive value-storing object instances. It is also used to define which instance variables are required to be filled in and which instance variables have been filled in. Finally it is used to store a label to be printed when an instance variable's value is displayed.

TEXT. This is a primitive-level object class used to define the TEXT data type.

VARIABLE. This class corresponds to the Implementation Phase 'Variable' data dictionary entry.

Grammar Design. The design of the grammar consists of two parts. For each desired action type (e.g. add, delete, display, etc.), English-like sentences which describe the action and its object must be defined. This can be a never-ending process, since there are many sentences with the "same meaning" in the English language. The best that can be done is to try to define and implement the sentence structures that most people will use most of the time. The

minimum that must be done is to define at least one English-like way to describe each action that the data dictionary tool is capable of performing. Since the emphasis of this thesis effort was on the grammar constructor and interpreter, the second of these two approaches was taken.

The second part of the grammar definition task is to define the pieces of source code which transform the input English-like requests into a form usable by the grammar-to-DDT interface. From this perspective the grammar is really part of the interface, but, for presentation purposes, it will continue to be described separately from the interface.

Interface Design. The interface design makes use of object-oriented techniques. Here a single object class, called EVENT, is defined. This class is used to store information about what actions are to be executed by the data dictionary tool. The methods of EVENT send messages to the data dictionary to perform the desired actions. Appendix F provides a description of EVENT and its methods.

Conclusion

This chapter has described the design of DDS including the natural language processor, CoIn, and DDS's other subsystems. Structure chart and object-oriented design

methods were introduced and were then used in the design specification. Chapter IV continues the description of this effort by presenting the implementation phase.

IV. Implementation

Introduction

This chapter discusses the implementation of DDS including CoIn. As in previous chapters, the system level is looked at first, followed by a more detailed discussion of each of its subsystems. The final section of this chapter describes the integration of CoIn and DDS into the Software Development Workbench. Volume 2 of this thesis contains complete source code listings of the data dictionary system.

System Implementation

DDS was implemented on the AFIT Information Sciences Laboratory's DEC VAX-11/780 minicomputer under the VMS operating system Version 3.6. Figure 4.1 is a diagram of the configuration of this computer system during the time DDS was being implemented. The ISL VAX was chosen as the target machine for two reasons. First, it was available and not overloaded compared to other machines at AFIT. Second, it is the host machine of the SDW (Hadfield, 1982:17-18).

After deciding upon the target machine, the next major decision made during the implementation phase was the choice of the implementation programming language. Several languages were available on the target machine at the beginning of this phase, including C, Fortran, Lisp, Pascal,

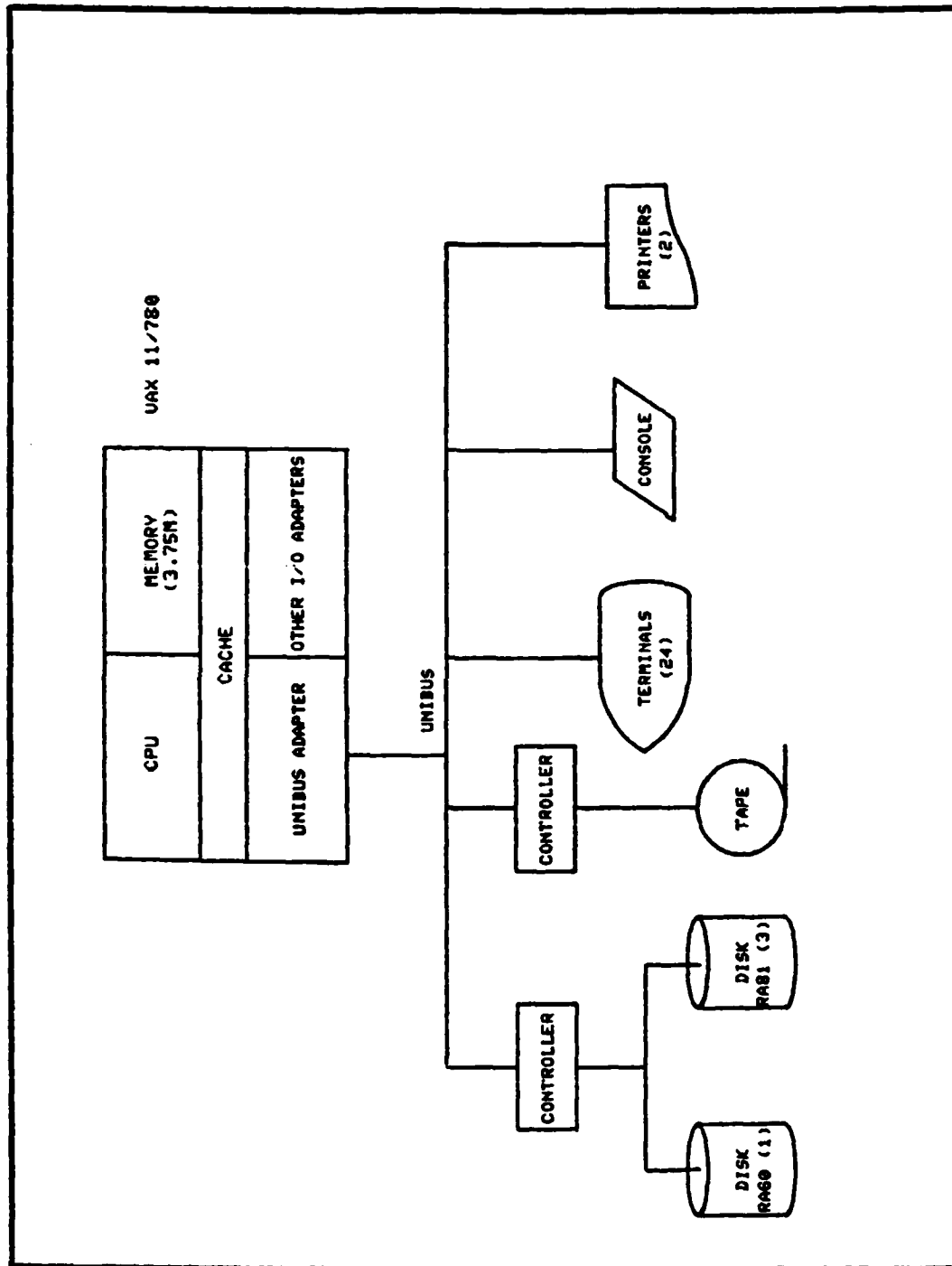


Figure 4.1. AFIT ISL VAX-11/780 Hardware Configuration

and Prolog. Of these, Lisp was chosen for three reasons. These are described in the next paragraphs.

The first and most important reason for choosing Lisp as the implementation language is the development environment provided by Lisp systems. Lisp systems generally provide an interpreter, compiler, editor, and debugger combined into one nicely integrated environment. This allows the Lisp programmer to easily jump back and forth between these different tools. As each subroutine is developed, it can be tested; debugged, modified, or redesigned as necessary; compiled into object code; and integrated into the system, all without leaving the Lisp environment.

The second reason for choosing Lisp is pedagogical in nature: a student of artificial intelligence is generally expected to learn to program in Lisp. Writing code in Lisp is a necessary part of fulfilling this goal. The thesis effort provided an excellent opportunity to pursue this goal.

The existence of a large amount of Lisp code that could possibly be used in DDS was the third reason for choosing Lisp as the implementation language. Prototypes of both the grammar constructor/interpreter and the software design part of the data dictionary tool were built in other projects (Wolfe, 1985a) (Wolfe, 1985b).

The particular implementation of Lisp used is called NIL (Burke, 1984) which is an acronym for New Implementation of Lisp. NIL was developed at MIT and is based on Common Lisp (Steele, 1984). NIL is a fairly complete implementation of Lisp. Its most serious lack is that it does not include a garbage collector (a routine which reclaims discarded memory cells). The lack of a garbage collector means that one must periodically exit NIL, restart it, and then reload DDS and one's database. Fortunately, this needs to be done seldom enough that it should not be a major problem. The reason for choosing NIL was that it was the only version of Lisp installed on the ISL VAX-11 computer at the time this effort was begun.

While the entire DDS is implemented in Lisp, parts of it (the data dictionary tool and the interface) are implemented using an object-oriented language built on top of Lisp. This language is called Flavors and is included in NIL (Burke, 1984:170-178). Flavors is an environment which allows one to define and manipulate objects (Chapter III contains a discussion of objects). Since Flavors is built on top of Lisp, one can still access all of the functions of Lisp. This makes the Flavors system ideal for implementing an object-oriented system which must interface with a system written in Lisp.

CoIn Implementation. This subsection describes the implementation of the natural language processor. CoIn was by far the most difficult part of the data dictionary system to design and implement. Implementation of the grammar data structures was straightforward--the record structures discussed in the last chapter were simply implemented as Lisp lists. However, the grammar constructor routines were difficult to implement properly, and the development of a workable design for the sentence interpreter was an iterative process of prototype development and throwaway.

The Grammar Constructor. The grammar constructor consists of an implementation of all of the subprogram designs described in Chapter III. Each of the subprograms, except for the grammar initialization and saving routines, is implemented as a Lisp macro (along with any needed lower-level subfunctions). Macros were used to simplify the user interface. Unlike a Lisp function, a macro does not evaluate its arguments before the body of its code is executed. Since the parameters that are passed to these macros are not meant to be evaluated, using macros saves the user the trouble of quoting the parameters. Since neither the grammar initialization nor the grammar saving subprograms require any arguments, they were implemented as Lisp functions.

Implementing the grammar constructor presented a couple of problems. One difficulty was the necessity of making the constructor's routines "destructive" in nature. That is, any changes made to the grammar using the constructor must permanently alter the global data structures of the grammar. Otherwise, if the change is not global, it will be lost, and the grammar will remain unchanged. In order to facilitate this requirement, a slightly modified version of the editor presented in Chapter 7 of Artificial Intelligence Programming (Charniak, et al, 1980:84-97) was heavily relied upon in the constructor implementation. This editor is destructive in nature. The original editor was designed to edit Lisp function definitions, but since the data structures of a grammar are not functions but lists, it was necessary to modify the editor to enable it to edit any Lisp symbolic expression. A side benefit of implementing this editor is that not only is it used in the grammar constructor, but it can be used as a standalone tool to edit the grammar data structures or any other Lisp symbolic expressions one wishes to modify.

Keeping track of the positions of the expression editor global pointers within the grammar data structures was a second source of difficulty in the constructor implementation. The data structures of a complex grammar are themselves complex. To overcome these problems, many

hours of tracing code while keeping track of pointers on a hand-drawn representation of the structures was done.

The Sentence Interpreter. As mentioned previously, the design and implementation of the sentence interpreter turned out to be an iterative process. The original version of the interpreter was developed as part of a project in EE 6.23, Artificial Intelligence System Design (Wolfe, 1985a). This version provided insight into the problem but no executing code.

The second design/implementation followed the form of the data structures much more closely. It was capable of correctly interpreting a grammatical sentence (if the grammar had been carefully designed) but could not always recognize a nongrammatical sentence as being invalid. The problem was that not enough information was being returned by the low-level subroutines for the driver subroutine to recognize an error. Correcting this problem was one of the major changes in the third (and present) version of the sentence interpreter.

The present module structure of the sentence interpreter is very similar to the previous one. Most of the differences are in the detailed design and implementation of the mid-level subroutines. Both the high-level driver routines and the low-level data structure access routines are essentially unchanged from the second

version. However, the module interfaces between the high-level and the mid-level routines were modified so as to return the information necessary for the top-level module to detect nongrammatical input sentences.

The interpreter parses an input sentence left-to-right comparing the words of the sentence to the grammar. As it successfully parses the sentence, it executes the associated Lisp source code contained in the grammar. If it is unable to fully parse a sentence using a production of the grammar, it backtracks to a previous branch point in the grammar and tries again. If the interpreter is unable to fully parse a sentence using any of the grammar productions, then the sentence is considered nongrammatical and an error handling routine is invoked. Currently the error handling routine displays only a message that the sentence was invalid within the context of the supplied grammar.

Data Dictionary Implementation. The data dictionary tool (DDT) was implemented as a set of Flavors objects and their associated methods. For most of the defined object classes of the data dictionary tool (not all were implemented due to time constraints), a Flavors object was declared. The slots of the Flavors objects correspond exactly to the instance variables defined in Appendix E. Similarly, the implemented Flavors access methods correspond exactly to the defined methods listed in Appendix E.

Since the low-level objects were needed by all of the higher level objects, they were implemented first (except for the TEXT object which was not implemented in this effort). Next the objects associated with the software design phase were implemented. This choice was made because some of the code already existed from a prior project (discussed above). Finally the highest-level object (PROJECT) was implemented. Also the objects concerned with the requirements phase and implementation phase were minimally implemented at this time.

Grammar Implementation. Once the grammar had been defined, implementing it was quite straightforward. The grammar productions of Appendix C were entered verbatim using the grammar construction routines of CoIn. Figure 4.2 shows an example of one of these productions (the notation used in the figure is defined in Appendix C). Entry errors were corrected either by making use of the expression editor in CoIn or by deleting and reentering the erroneous productions. Implementation of the grammar brought out a need for several more grammar modification subprograms. These are further discussed in Chapter V. The Lisp code that was included within the grammar is really part of the interface, so it is discussed in the next subsection.

```

+-----+
|                                             |
|                                             |
|               <grammar> ::=                |
|               (please) <aux-grammar>      |
|                                             |
|                                             |
+-----+

```

Figure 4.2. Sample Grammar Production

The defined grammar was not completely implemented during this thesis effort. Effort was concentrated on the presentation part of the grammar as it was thought to be simple enough to fully implement, yet complex enough to be a valid demonstration of that CoIn can be used to implement a useful natural language human-computer interface. The presentation part of the grammar was considered relatively simple because during presentation of the data, there is no new information being added. The information that is contained in an input sentence can all be checked word-for-word against the grammar and database. The presentation part of the grammar was considered complex enough because it needs to have the capability of accessing the entire database, and it makes use of all the capabilities of the interpreter.

Other parts of the grammar that were implemented during this thesis effort include the initialization productions and the data save and quit commands. The initialization

productions were implemented because it was quite simple to do so, and the data save and quit commands because of their obvious importance.

Interface Implementation. The interface between the sentence interpreter and DDT was implemented as a single Flavors object class called EVENT. A set of methods to access this class and the internal Lisp code of the grammar, which invokes these methods, constitute the remainder of the interface implementation.

The EVENT object class was modeled after Schank's Conceptual Dependency (CD) theory (Schank and Riesbeck, 1981:10-26). According to the CD theory, every event has an actor, an action, an object, and a direction. The actor is the entity which performs the action. The action is performed upon an object and is oriented in a direction. In the case of the interface, the actor is always the computer, the action is one of the defined commands of the interface, the object is an instance of one of the data dictionary's objects, and the direction is either from the data base to the user's terminal or vice versa.

A second idea of CD theory is best described by Schank and Riesbeck: "When two sentences describe the same event in such a way that these descriptions have the same overall meaning but quite different forms, we expect out CD

representations to be identical for both descriptions." (Schank and Riesbeck, 1981:14). This idea was also used in the design and implementation of the interface. A set of command types corresponding to the operations of DDT was defined in the grammar. The various sentence forms of each of the command types is mapped into an identical representation by the interface. Upon completion of a successful parse, this representation is converted into a command that can be executed directly by the data dictionary tool. Any response by DDT is sent directly to the user's terminal.

Integration into the SDW

Using the Software Development Workbench Executive Maintenance Guide (Hadfield, 1982:355-364) as a guide, DDS and CoIn were integrated into the SDW. The first decision made for each of these systems was the choice of a two letter code to be entered by the SDW user to invoke the tool. The codes 'DD' for DDS and 'NL' for CoIn were chosen because they were available and because of their obvious mnemonic nature.

The second decision made was to determine to which of the SDW's functional groups of tools each system should be added. Since the design phase part of DDS was concentrated upon during implementation, it was decided to initially add DDS to the Design Tools functional group of the SDW. When

the requirements phase part of DDS is completed, DDS should be added to the Requirements Definition Tools functional group. Likewise, when the implementation phase part of DDS is completed, DDS should be added to one of the implementation tools functional groups, possibly the Text Editors group. The grammar constructor portion of CoIn is essentially a grammar editor, so CoIn was added to the Text Editors group.

Conclusion

This chapter discussed the implementation of DDS and its subsystems, including CoIn. The choice of the target machine was discussed, as was the choice of implementation language. This chapter concluded with a discussion concerning the integration into the SDW of DDS and CoIn. The next chapter, Chapter V, concludes this thesis by reviewing and analyzing this thesis effort and providing suggestions for further work.

V. Conclusion and Recommendations

Introduction

This thesis has described the development of a natural language processor, CoIn, and its application to a data dictionary system. DDS, the data dictionary system, consists of several distinct but interacting parts: a natural language human-computer interface implemented using CoIn, a data dictionary access mechanism, and an interface between the grammar and the data dictionary tool. The emphasis in DDS was on the first of these subsystems. Although the other subsystems make DDS a usable system, their primary role was to show that the interface constructor and interpreter is a useful and usable tool. This final chapter first presents a short summary of the system development. Following this, an analysis which relates the current system back to the standards described in Chapter I is presented. Finally, a list of recommendations for the completion and enhancement of the system is included.

Development Summary

CoIn and DDS were built using a variation of the classic software development life cycle. First an extensive literature search was done to gain a better understanding of

natural language systems, the software development process and its problems, and how these problems might be diminished by automation. The information gleaned from this search, along with prior knowledge, was used as the input to the requirements analysis phase. In this phase, sets of requirements were defined for the natural language processor, for the entire data dictionary system, and for each of its subsystems.

After generating the initial set of requirements, an iterative process of design-implement-test had to be done. This process provided necessary feedback about the completeness and consistency of the requirements. This feedback was used to modify the requirements as required. The design and implementation of the sentence interpreter in particular was a cyclic process of prototype development. Throughout the implementation, routines were tested as they were developed, both isolated from and integrated into the system. The testing done is by far inadequate (mainly due to its informality) but does suggest that DDS is reasonably error-free.

Analysis of the Current System

As has been shown in this thesis effort, CoIn does allow one to construct a usable natural language human-computer interface. The constructor does provide at

least the minimum set of routines necessary for building a grammar, and their interfaces do seem consistent with each other. The interpreter is able to correctly parse grammatical sentences within the implemented DDS grammar (Figure 5.1). The interpreter is also able to recognize and report non-grammatical sentences (Figure 5.2). Therefore, from the point of view of the natural language processor, this project should be considered a success.

The data dictionary itself is not complete and needs to be extended. This is discussed more in the next section. Enough of the system is implemented to show that it is a reasonable project. What is implemented does show the utility of the natural language processor, so the data dictionary should be considered fairly successful. Finishing DDT should be easy, albeit time-consuming.

Finally, DDS does operate in a reasonable amount of time. Several timing test were made with an moderate load on the computer system (4 users logged in). DDS was able to consistently respond in 1 to 3 seconds.

Recommendations for Future Work

DDS is incomplete. Not all of the requirements have been fulfilled by the design. Not all of the design has been implemented. More, formal testing, in accordance with the test plan (see Appendix G), needs to be done. The

```

+-----+
| Data Dictionary System (DDS) |
| Type (help) for help.      |
|                               |
| DDS-> (please show me the version and date of process |
|         is-process)        |
|                               |
| Entry Version:             1 |
| Entry Date:                29 Aug 85 |
|                               |
| DDS->                       |
+-----+

```

Figure 5.1. Example of a Valid Sentence Entered into DDS

```

+-----+
| DDS-> (what are the calling processes of foo alias1) |
| *** Error - Can't parse sentence ***                |
| DDS->                                                |
+-----+

```

Figure 5.2. Example of an Invalid Sentence Entered into DDS

following subsections describe some of the work that needs to be done for DDS to become a usable system.

CoIn. Of the subsystems, CoIn is probably the closest to being a "complete" implementation. This is natural since the emphasis of the thesis work was on this part of the system. The most severe lack in CoIn is in its handling of nongrammatical input sentences. The sentence interpreter's error handling routine needs to be greatly enhanced. As it

is implemented now, it provides virtually no information to a user of the system. This is a serious limitation and needs to be investigated and corrected as soon as possible.

In Chapter IV, the statement was made that several grammar modification subprograms need to be added to the grammar constructor. There are at least two such routines. First, a routine to remove a production from a production record is needed. This can currently be done either by destroying the production record and reentering all but the production to be deleted or by using the s-expression error. Neither of these solutions is good a good one. The first can be very time consuming; the second, due to the recursive nature of production records, is difficult and error-prone.

The second subprogram that needs to be added to the grammar constructor is one that would allow the modification of the Lisp source code contained in the grammar. The need for this routine is not as critical as the need for the production deletion routine--modification of the Lisp code is fairly easy using the s-expression editor. This is more of a "nice to have" routine.

DDT. The emphasis in DDT was on the objects needed during the design phase of the software development life cycle. Therefore, the implementation of these objects is closer to being complete than the objects associated with the requirements or implementation phase. However, even the

design phase objects still need work. Files for source code have been created and instance variables have been defined for all of the object classes. Several of these object definitions have not yet been entered. While the defined access methods have all been implemented, the access methods of several object classes have not yet been defined. Before DDS can be used throughout the life cycle, these deficiencies must be eliminated.

DDT was implemented using the Flavors language of NIL. A subject worth investigating is the rehosting of DDT using a data base management system (DBMS). Unfortunately, NIL provides no means for interfacing to an external DBMS. However, DEC's VAX Lisp is now available on the ISL VAX. VAX Lisp may provide the capability needed to access external systems. Whether this is indeed the case needs to be determined.

The Grammar. It is felt that the grammar as defined is adequate for interacting with DDS. This is not to imply that the grammar is all inclusive, but that it does provide at least one way to specify each of the possible operations of DDT. Extending and refining the DDS's grammar could probably be a thesis project all by itself.

One of the primary deficiencies of the grammar is that it was not completely implemented during this thesis effort. Completing the implementation should be possible for a Lisp

programmer with reasonable knowledge of the structure of DDS.

The Interface. How close the interface is to being complete is difficult to judge. As the grammar is implemented, the corresponding Lisp interface code must be included. Whether changes will be necessary to the rest of the interface depends on how this code is implemented.

Appendix A
System Data Dictionary

Introduction

This appendix contains the data dictionary for DDS and CoIn. Currently there are four entry types: ACTIVITY, ALIAS, DATA FLOW, and PROCESS. The ACTIVITY, ALIAS, and DATA FLOW types are all associated with the functional requirements analysis phase of the software development lifecycle. The PROCESS type is associated with the design phase.

DDS System Data Dictionary

NAME: ACCESS DATA DICTIONARY
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: DATA DICTIONARY ACCESS COMMAND
DATA DICTIONARY DATA
OUTPUTS: DATA DICTIONARY DATA
DATA DICTIONARY ERROR MESSAGE
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: DATA DICTIONARY SYSTEM
VERSION: 1
DATE: 22 OCT 85

NAME: ADD-PRODUCTION (MACRO)
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION
CODE TO EXECUTE
PRODUCTION RECORD NAME

INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *EDIT-EXP*
GLOBAL DATA CHANGED: *EDIT-EXP*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: IS-PRODUCTION
ED-RESET
AUX-ADD-PRODUCTION
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: ADDITION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: DATA DICTIONARY ACCESS COMMAND
COMPOSITION:
ALIASES:
SOURCES: UPDATE DATA DICTIONARY
DESTINATIONS: UPDATE DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

NAME: AUX-ADD-PRODUCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION
CODE TO EXECUTE

INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *EDIT-PTR*
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: ED-RIGHT
INSERT-PRODUCTION
ED-DOWN
LOOP&
AUX-ADD-PRODUCTION (RECURSIVE)
ED-UP

RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF:
COMPOSITION: CONSTRUCTION COMMAND
SENTENCE
ALIASES:
SOURCES: USER
DESTINATIONS: DATA DICTIONARY SYSTEM
VERSION: 2
DATE: 22 OCT 85

NAME: CONSTRUCT GRAMMAR
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: CONSTRUCTION COMMAND
GRAMMAR
OUTPUTS: CONSTRUCTION ERROR MESSAGE
GRAMMAR
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: DATA DICTIONARY SYSTEM
VERSION: 2
DATE: 22 OCT 85

NAME: CONSTRUCTION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: COMMAND
COMPOSITION: GRAMMAR LOAD COMMAND
GRAMMAR MODIFICATION COMMAND
GRAMMAR SAVE COMMAND
ALIASES:
SOURCES:
DESTINATIONS: CONSTRUCT GRAMMAR
VERSION: 2
DATE: 22 OCT 85

NAME: CONSTRUCTION ERROR MESSAGE
TYPE: DATA FLOW
PROJECT: DDS
PART OF: RESPONSE
COMPOSITION:
ALIASES:
SOURCES: CONSTRUCT GRAMMAR
MODIFY GRAMMAR
DESTINATIONS:
VERSION: 2
DATE: 22 OCT 85

NAME: CREATE-FUNCTION (MACRO)
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION RECORD NAME
INPUT FLAGS: FUNCTION CODE
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: CREATE-META-SYMBOL
SETF*
MSG
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: CREATE-META-SYMBOL
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: META-SYMBOL NAME
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *META-SYMBOLS*
GLOBAL DATA CHANGED: *META-SYMBOLS*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: CREATE-PRODUCTION (MACRO)
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION RECORD NAME
CODE TO EXECUTE
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: CREATE-META-SYMBOL
SETF*
MSG
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: DATA DICTIONARY ACCESS COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF:
COMPOSITION:

ADDITION COMMAND
DELETION COMMAND
INITIALIZATION COMMAND
MODIFICATION COMMAND
PRESENTATION COMMAND

ALIASES:
SOURCES:

GENERATE DD COMMAND
INTERPRET SENTENCE
ACCESS DATA DICTIONARY
1
22 OCT 85

DESTINATIONS:
VERSION:
DATE:

NAME:
TYPE:
PROJECT:
PART OF:
COMPOSITION:
ALIASES:
SOURCES:

DATA DICTIONARY DATA
DATA FLOW
DDS
RESPONSE

DESTINATIONS:

ACCESS DATA DICTIONARY
DISPLAY DATA DICTIONARY
UPDATE DICTIONARY
ACCESS DATA DICTIONARY
DISPLAY DATA DICTIONARY
UPDATE DATA DICTIONARY

VERSION:
DATE:

1
22 OCT 85

NAME:
TYPE:
PROJECT:
PART OF:
COMPOSITION:
ALIASES:
SOURCES:

DATA DICTIONARY ERROR MESSAGE
DATA FLOW
DDS
RESPONSE

DESTINATIONS:
VERSION:
DATE:

ACCESS DATA DICTIONARY
DISPLAY DATA DICTIONARY
UPDATE DATA DICTIONARY
1
22 OCT 85

NAME: DATA DICTIONARY SYSTEM
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION: COMMAND
INPUTS: DATA DICTIONARY
DATA DICTIONARY
OUTPUTS: RESPONSE
CONTROLS:
MECHANISMS:
ALIASES: DDS
PARENT ACTIVITY: (NONE)
VERSION: 2
DATE: 22 OCT 85

NAME: DDS
TYPE: ALIAS
DD TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION: DATA DICTIONARY SYSTEM
SYNONYM:
VERSION: 2
DATE: 22 OCT 85

NAME: DELETION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: DATA DICTIONARY ACCESS COMMAND
COMPOSITION:
ALIASES:
SOURCES: UPDATE DATA DICTIONARY
DESTINATIONS: UPDATE DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

NAME: DESTROY-FUNCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION RECORD NAME
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: IS-FUNCTION
DESTROY-META-SYMBOL
MSG

RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: DESTROY-META-SYMBOL
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: META-SYMBOL NAME
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *META-SYMBOLS*
GLOBAL DATA CHANGED: *META-SYMBOLS*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: IS-META-SYMBOL
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: DESTROY-PRODUCTION (MACRO)
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION RECORD NAME
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ: '
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: IS-PRODUCTION
DESTROY-META-SYMBOL
MSG

RELATED ACTIVITY:
VERSION: 1
DATE:

NAME: DISPLAY DATA DICTIONARY
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: DATA DICTIONARY DATA
PRESENTATION COMMAND
OUTPUTS: DATA DICTIONARY DATA
DATA DICTIONARY ERROR MESSAGE

CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: ACCESS DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

NAME: DISPLAY-PARSE-ERROR
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA:
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: MSG
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: FTN-CODE
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION RECORD
INPUT FLAGS:
OUTPUT DATA: FUNCTION CODE
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: FTN-NAME
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION RECORD
INPUT FLAGS:
OUTPUT DATA: FUNCTION NAME
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: FTN-PARSE
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: SENTENCE
FUNCTION RECORD
INPUT FLAGS:
OUTPUT DATA: RESULT
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: FTN-NAME
FTN-CODE
RELATED ACTIVITY:
VERSION: 1
DATE:

NAME: GENERATE DD COMMAND
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: PARSE RESULTS
OUTPUTS: DATA DICTIONARY ACCESS COMMAND
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: INTERPRET SENTENCE
VERSION: 1
DATE: 22 OCT 85

NAME: GRAMMAR
TYPE: DATA FLOW
PROJECT: DDS
PART OF:
COMPOSITION:
ALIASES:
SOURCES: CONSTRUCT GRAMMAR
LOAD GRAMMAR
MODIFY GRAMMAR
PERMANENT GRAMMAR
SAVE GRAMMAR
WORKING GRAMMAR
DESTINATIONS: CONSTRUCT GRAMMAR
INTERPRET SENTENCE
LOAD GRAMMAR
MODIFY GRAMMAR
PARSE SENTENCE
PERMANENT GRAMMAR
SAVE GRAMMAR
WORKING GRAMMAR
VERSION: 2
DATE: 22 OCT 85

NAME: GRAMMAR LOAD COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: CONSTRUCTION COMMAND
COMPOSITION:
ALIASES:
SOURCES:
DESTINATIONS: LOAD GRAMMAR
VERSION: 1
DATE: 22 OCT 85

NAME: GRAMMAR MODIFICATION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: CONSTRUCTION COMMAND
COMPOSITION:
ALIASES:
SOURCES:
DESTINATIONS: MODIFY GRAMMAR
VERSION: 1
DATE: 22 OCT 85

NAME: GRAMMAR-CODE
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION RECORD
INPUT FLAGS:
OUTPUT DATA: PRODUCTION CODE
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: GRAMMAR-LIST
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION RECORD
INPUT FLAGS:
OUTPUT DATA: SUB-GRAMMAR LIST
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: GRAMMAR-NAME
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION RECORD
INPUT FLAGS:
OUTPUT DATA: PRODUCTION NAME
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME:	GRAMMAR-PARSE
TYPE:	PROCESS
PROJECT:	DDS
NUMBER:	
DESCRIPTION:	
INPUT DATA:	SENTENCE SUB-GRAMMAR LIST
INPUT FLAGS:	
OUTPUT DATA:	RESULT
OUTPUT FLAGS:	
GLOBAL DATA USED:	
GLOBAL DATA CHANGED:	
FILES READ:	
FILES WRITTEN:	
HARDWARE READ:	
HARDWARE WRITTEN:	
ALIASES:	
CALLING PROCESSES:	
PROCESSES CALLED:	GRAMMAR-CODE GRAMMAR-NAME GRAMMAR-LIST IS-PRODUCTION SUB-PARSE IS-FUNCTION FTN-PARSE RESULT-LIST GRAMMAR-PARSE (RECURSIVE) RESULT-SENTENCE
RELATED ACTIVITY:	
VERSION:	1
DATE:	04 NOV 85

NAME:	GRAMMAR SAVE COMMAND
TYPE:	DATA FLOW
PROJECT:	DDS
PART OF:	CONSTRUCTION COMMAND
COMPOSITION:	
ALIASES:	
SOURCES:	
DESTINATIONS:	SAVE GRAMMAR
VERSION:	1
DATE:	22 OCT 85

NAME: INITIALIZATION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: DATA DICTIONARY ACCESS COMMAND
COMPOSITION:
ALIASES:
SOURCES:
DESTINATIONS: UPDATE DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

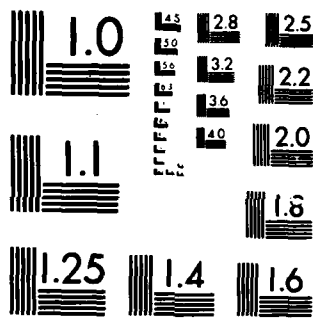
NAME: INITIALIZE-GRAMMAR
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA:
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *META-SYMBOLS*
GLOBAL DATA CHANGED: *META-SYMBOLS*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: FOR&
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: INSERT-FUNCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION CODE
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *EDIT-PTR*
GLOBAL DATA CHANGED: *EDIT-PTR*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: SETF*
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: INSERT-PRODUCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: PRODUCTION
CODE TO EXECUTE
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *EDIT-PTR*
GLOBAL DATA CHANGED: *EDIT-PTR*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: ED-DOWN
AUX-ADD-PRODUCTION
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: INTERPRET SENTENCE
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: GRAMMAR
SENTENCE
OUTPUTS: DATA DICTIONARY ACCESS COMMAND
INTERPRETER ERROR MESSAGE
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: DATA DICTIONARY SYSTEM
VERSION: 2
DATE: 22 OCT 85

NAME: INTERPRETER ERROR MESSAGE
TYPE: DATA FLOW
PROJECT: DDS
PART OF: RESPONSE
COMPOSITION: PARSE ERROR MESSAGE
ALIASES:
SOURCES: INTERPRET SENTENCE
DESTINATIONS:
VERSION: 2
DATE: 22 OCT 85



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAME: IS-FUNCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: SYMBOL TO CHECK
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS: IS-FUNCTION
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: IS-META-SYMBOL
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: SYMBOL TO CHECK
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS: IS-META-SYMBOL
GLOBAL DATA USED: *META-SYMBOLS*
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: IS-PRODUCTION
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: SYMBOL TO CHECK
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS: IS-PRODUCTION
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: LOAD GRAMMAR
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: GRAMMAR
GRAMMAR SAVE COMMAND
GRAMMAR
OUTPUTS:
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: CONSTRUCT GRAMMAR
VERSION: 2
DATE: 22 OCT 85

NAME: MODIFICATION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: DATA DICTIONARY ACCESS COMMAND
COMPOSITION:
ALIASES:
SOURCES: UPDATE DATA DICTIONARY
DESTINATIONS: UPDATE DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

NAME: MODIFY-FUNCTION (MACRO)
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: FUNCTION CODE
FUNCTION RECORD NAME

INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *EDIT-PTR*
GLOBAL DATA CHANGED: *EDIT-PTR*
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:

IS-FUNCTION
ED-RESET
INSERT-FUNCTION
MSG

RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: MODIFY GRAMMAR
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: GRAMMAR
GRAMMAR MODIFICATION COMMAND
CONSTRUCTION ERROR MESSAGE
GRAMMAR
OUTPUTS:
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: CONSTRUCT GRAMMAR
VERSION: 1
DATE: 22 OCT 85

NAME: PARSE
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: SENTENCE
GRAMMAR

INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: GRAMMAR-LIST
GRAMMAR-CODE
GRAMMAR-PARSE
SUCCESSFUL
DISPLAY-PARSE-ERROR

RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: PARSE ERROR MESSAGE
TYPE: DATA FLOW
PROJECT: DDS
PART OF: INTERPRETER ERROR MESSAGE
COMPOSITION:
ALIASES:
SOURCES: PARSE SENTENCE
DESTINATIONS:
VERSION: 1
DATE: 22 OCT 85

NAME: PARSE RESULTS
TYPE: DATA FLOW
PROJECT: DDS
PART OF:
COMPOSITION:
ALIASES:
SOURCES: PARSE SENTENCE
DESTINATIONS: GENERATE DD COMMAND
VERSION: 1
DATE: 22 OCT 85

NAME: PARSE SENTENCE
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: GRAMMAR
SENTENCE
OUTPUTS: PARSE ERROR MESSAGE
PARSE RESULTS
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: INTERPRET SENTENCE
VERSION: 1
DATE: 22 OCT 85

NAME: PRESENTATION COMMAND
TYPE: DATA FLOW
PROJECT: DDS
PART OF: DATA DICTIONARY ACCESS COMMAND
COMPOSITION:
ALIASES:
SOURCES:
DESTINATIONS: DISPLAY DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

NAME: RESPONSE
TYPE: DATA FLOW
PROJECT: DDS
PART OF:
COMPOSITION: CONSTRUCTION ERROR MESSAGE
DATA DICTIONARY DATA
DATA DICTIONARY ERROR MESSAGE
INTERPRETER ERROR MESSAGE

ALIASES:
SOURCES: DATA DICTIONARY SYSTEM
DESTINATIONS: USER
VERSION: 2
DATE: 22 OCT 85

NAME: RESULT-LIST
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION: RESULT
INPUT DATA:
INPUT FLAGS:
OUTPUT DATA: SUB-GRAMMAR LIST
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: RESULT-SENTENCE
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: RESULT
INPUT FLAGS:
OUTPUT DATA: SENTENCE
OUTPUT FLAGS:
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: SAVE GRAMMAR
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: GRAMMAR
GRAMMAR SAVE COMMAND
GRAMMAR
OUTPUTS:
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: CONSTRUCT GRAMMAR
VERSION: 2
DATE: 22 OCT 85

NAME: SAVE-GRAMMAR
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA:
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS:
GLOBAL DATA USED: *META-SYMBOLS*
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN: GRAMMAR.LSP
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED:
RELATED ACTIVITY:
VERSION: 1
DATE: 29 OCT 85

NAME: SENTENCE
TYPE: DATA FLOW
PROJECT: DDS
PART OF: COMMAND
COMPOSITION:
ALIASES:
SOURCES:
DESTINATIONS: INTERPRET SENTENCE
PARSE SENTENCE
VERSION: 2
DATE: 22 OCT 85

NAME:	SUB-GRAMMAR-PARSE
TYPE:	PROCESS
PROJECT:	DDS
NUMBER:	
DESCRIPTION:	
INPUT DATA:	SENTENCE SUB-GRAMMAR LIST
INPUT FLAGS:	
OUTPUT DATA:	RESULT
OUTPUT FLAGS:	
GLOBAL DATA USED:	
GLOBAL DATA CHANGED:	
FILES READ:	
FILES WRITTEN:	
HARDWARE READ:	
HARDWARE WRITTEN:	
ALIASES:	
CALLING PROCESSES:	
PROCESSES CALLED:	GRAMMAR-CODE GRAMMAR-NAME GRAMMAR-LIST IS-PRODUCTION SUB-PARSE IS-FUNCTION FTN-PARSE RESULT-LIST SUB-GRAMMAR-PARSE (RECURSIVE) RESULT-SENTENCE
RELATED ACTIVITY:	
VERSION:	1
DATE:	04 NOV 85

NAME:	SUB-PARSE
TYPE:	PROCESS
PROJECT:	DDS
NUMBER:	
DESCRIPTION:	
INPUT DATA:	SENTENCE GRAMMAR
INPUT FLAGS:	
OUTPUT DATA:	RESULT
OUTPUT FLAGS:	
GLOBAL DATA USED:	
GLOBAL DATA CHANGED:	
FILES READ:	
FILES WRITTEN:	
HARDWARE READ:	
HARDWARE WRITTEN:	
ALIASES:	
CALLING PROCESSES:	
PROCESSES CALLED:	GRAMMAR-LIST GRAMMAR-CODE SUB-GRAMMAR-PARSE RESULT-LIST
RELATED ACTIVITY:	
VERSION:	1
DATE:	04 NOV 85

NAME: SUCCESSFUL
TYPE: PROCESS
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUT DATA: RESULT
INPUT FLAGS:
OUTPUT DATA:
OUTPUT FLAGS: SUCCESSFUL
GLOBAL DATA USED:
GLOBAL DATA CHANGED:
FILES READ:
FILES WRITTEN:
HARDWARE READ:
HARDWARE WRITTEN:
ALIASES:
CALLING PROCESSES:
PROCESSES CALLED: RESULT-SENTENCE
RESULT-LIST

RELATED ACTIVITY:
VERSION: 1
DATE: 04 NOV 85

NAME: UPDATE DATA DICTIONARY
TYPE: ACTIVITY
PROJECT: DDS
NUMBER:
DESCRIPTION:
INPUTS: ADDITION COMMAND
DATA DICTIONARY DATA
DELETION COMMAND
INITIALIZATION COMMAND
MODIFICATION COMMAND

OUTPUTS: ADDITION COMMAND
DATA DICTIONARY DATA
DATA DICTIONARY ERROR MESSAGE
DELETION COMMAND
MODIFICATION COMMAND

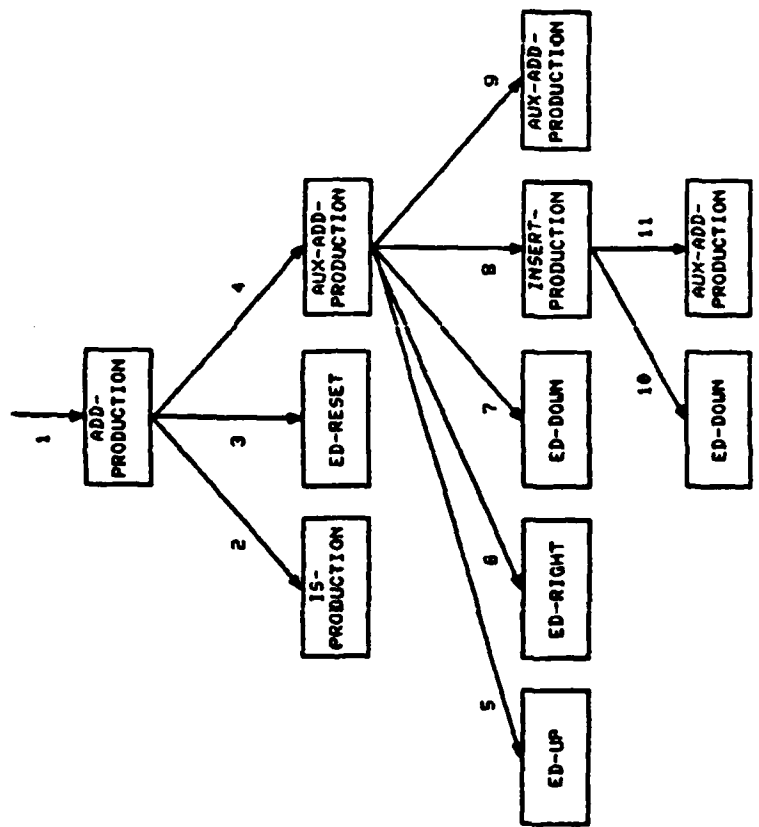
CONTROLS:
MECHANISMS:
ALIASES:
PARENT ACTIVITY: ACCESS DATA DICTIONARY
VERSION: 2
DATE: 22 OCT 85

Appendix B
Structure Charts

Appendix B
Table of Contents

	Page
Grammar Constructor	B- 3
Sentence Interpreter	B- 20
S-expression Editor	B- 23
DESIGN Methods	B- 44
EVENT Methods	B- 51
IMPLEMENTATION Methods	B- 59
PARAMETER Methods	B- 64
PARAMETER-ALIAS Methods	B- 71
PROCESS Methods	B- 78
PROCESS-ALIAS Methods	B- 93
PROJECT Methods	B-100
REQUIREMENTS Methods	B-107

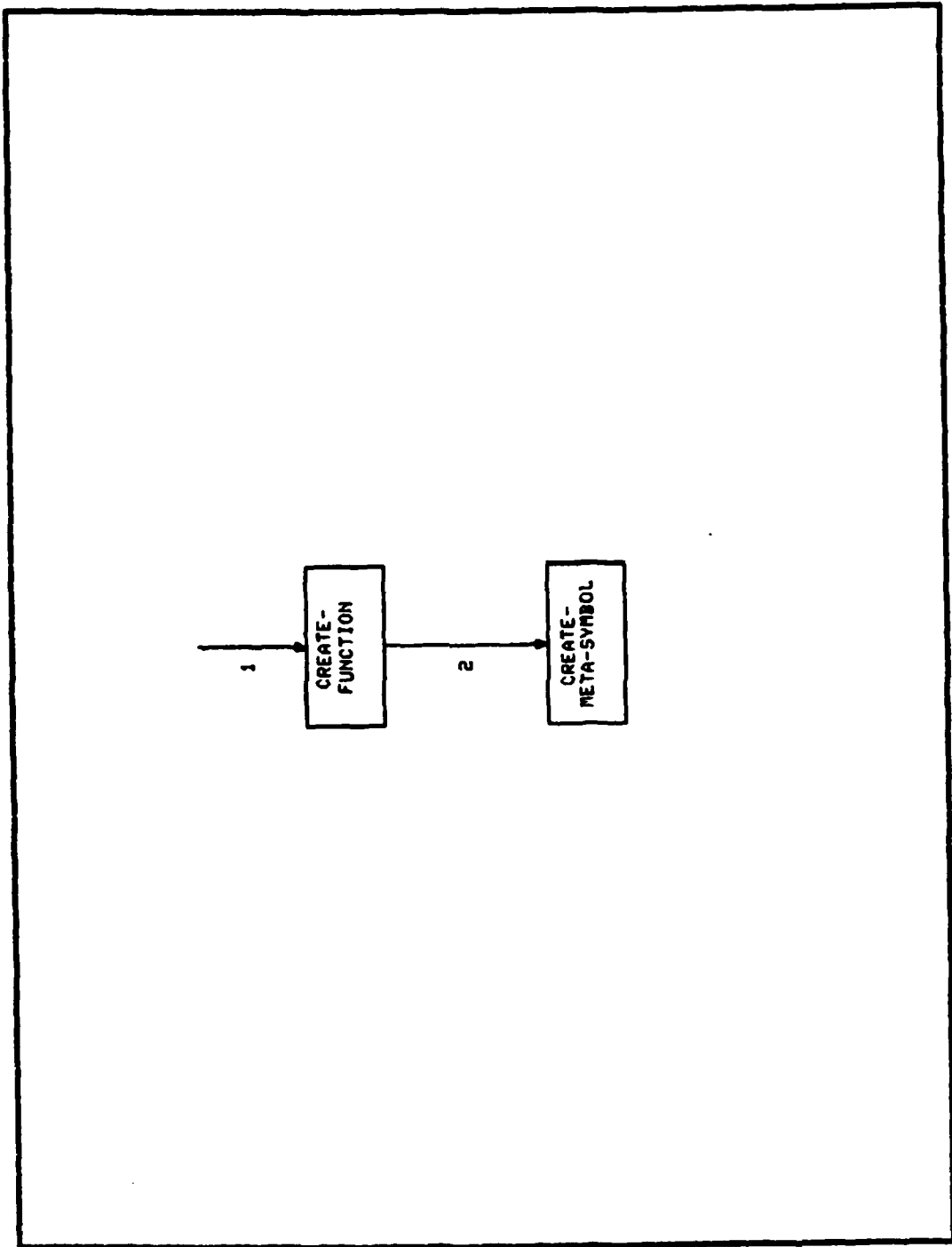
Grammar Constructor



add-production Interfaces

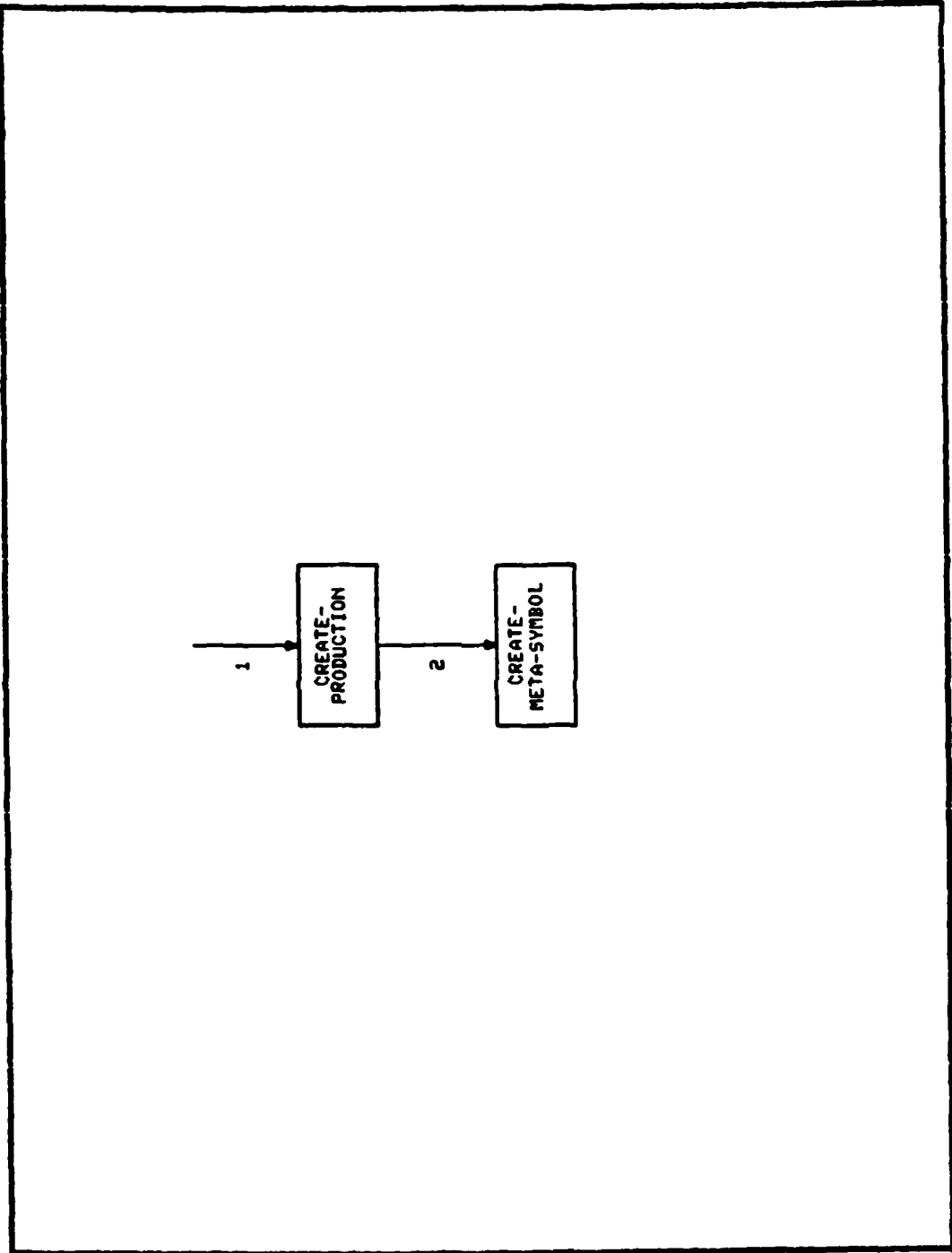
#	Passed Parameters	Type	Returned Parameters	Type
1	sentence code production name	data data data		
2	production-name	data	is production?	flag
3	*edit-exp* *edit-exp* nil	data data data		
4	sentence code	data data		
5	(none)			
6	(none)			
7	(none)			
8	sentence code	data data		
9	cdr*(sentence) code	data data		
10	(none)			
11	cdr*(sentence) code	data data		

* The cdr function returns a list of all but the first element of a list passed to it. For instance, the cdr of the list (a b c) is the list (b c).



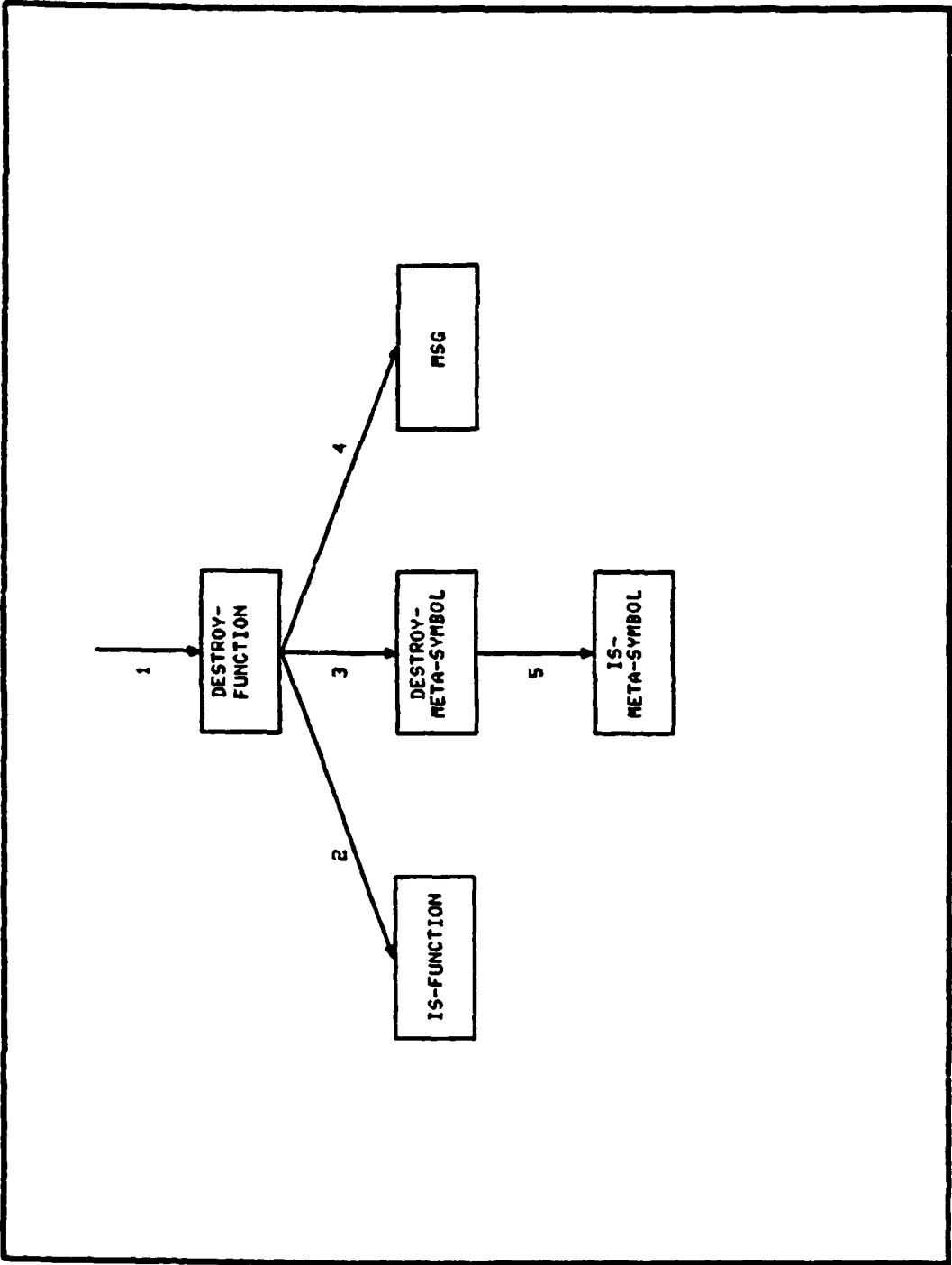
create-function Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	function-name code	data data		
2	function-name	data	function created?	flag



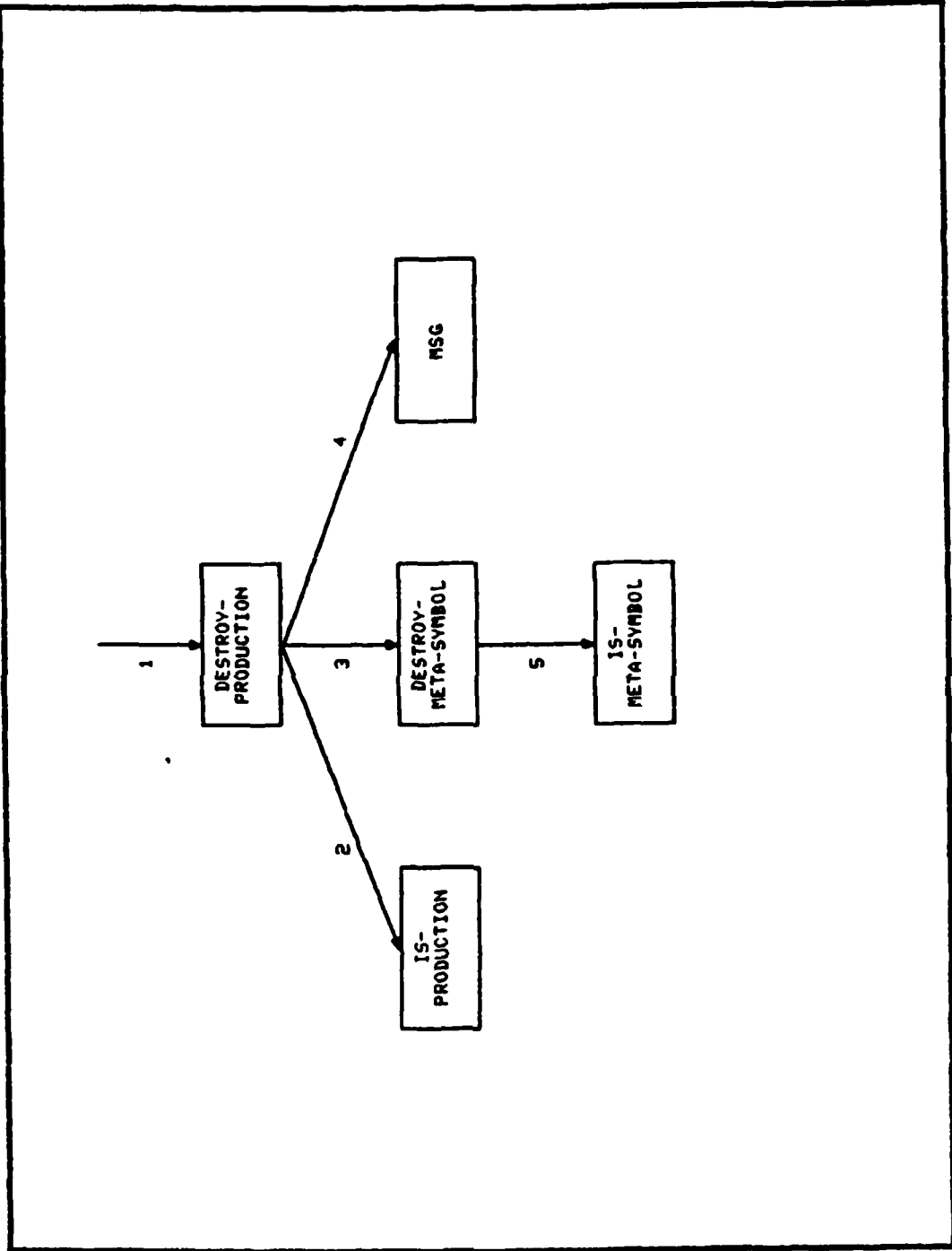
create-production Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	production-name code	data data		
2	production-name	data	production created?	flag



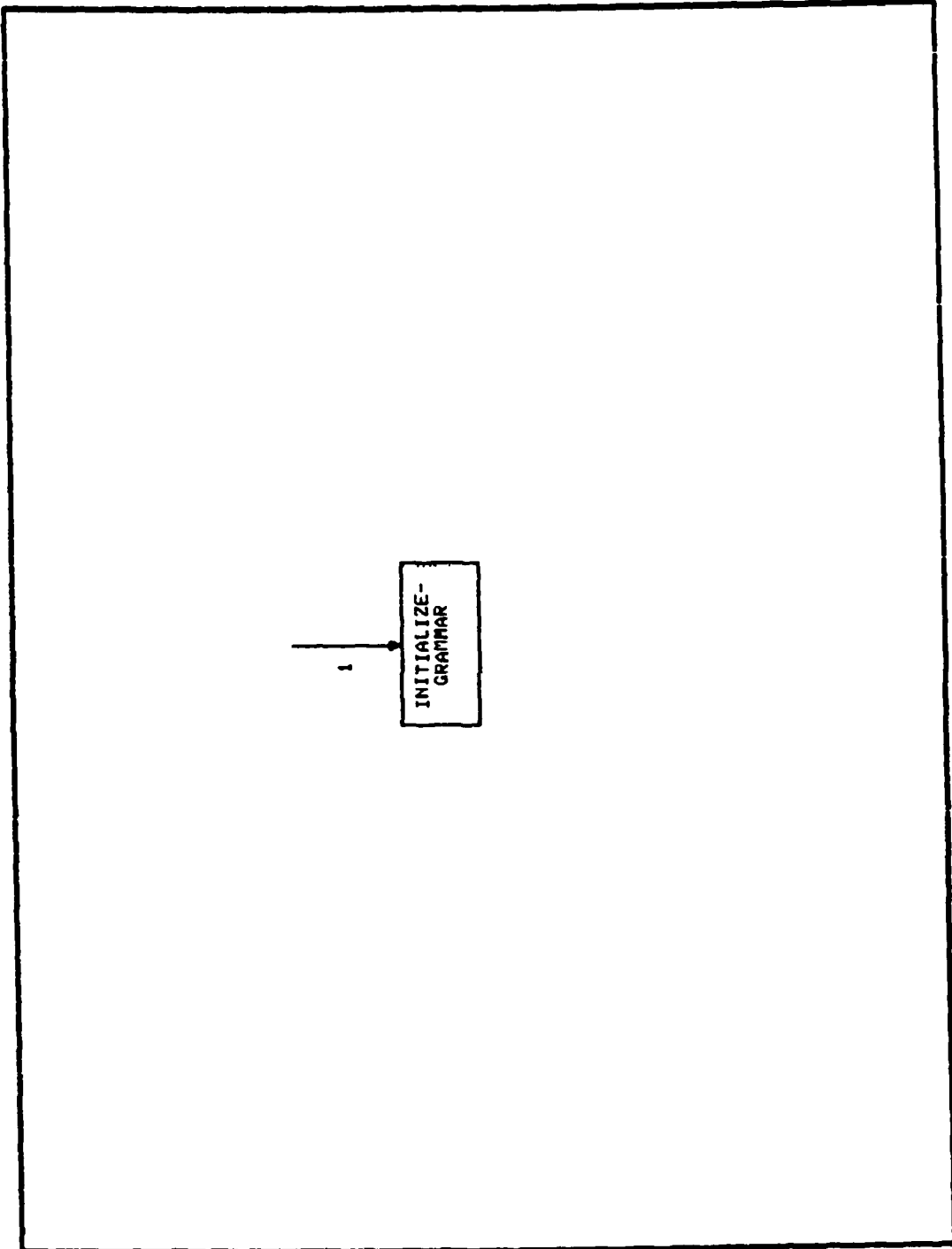
destroy-function Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	function name	data		
2	function name	data	is function?	flag
3	function name	data		
4	error message text	data		
5	function name	data	is meta-symbol?	flag



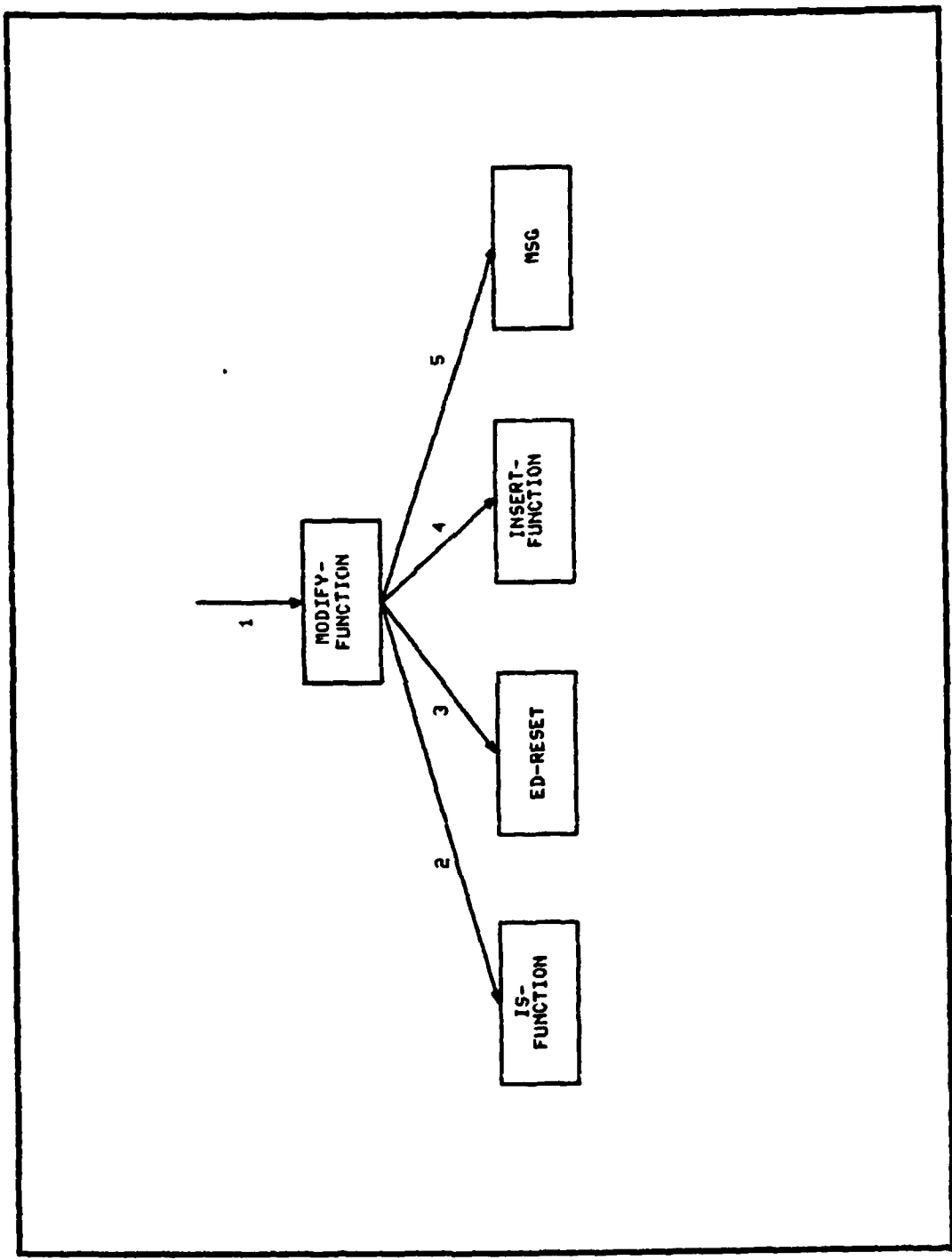
destroy-production Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	production name	data		
2	production name	data	is production?	flag
3	production name	data		
4	error message text	data		
5	production name	data	is meta-symbol?	flag



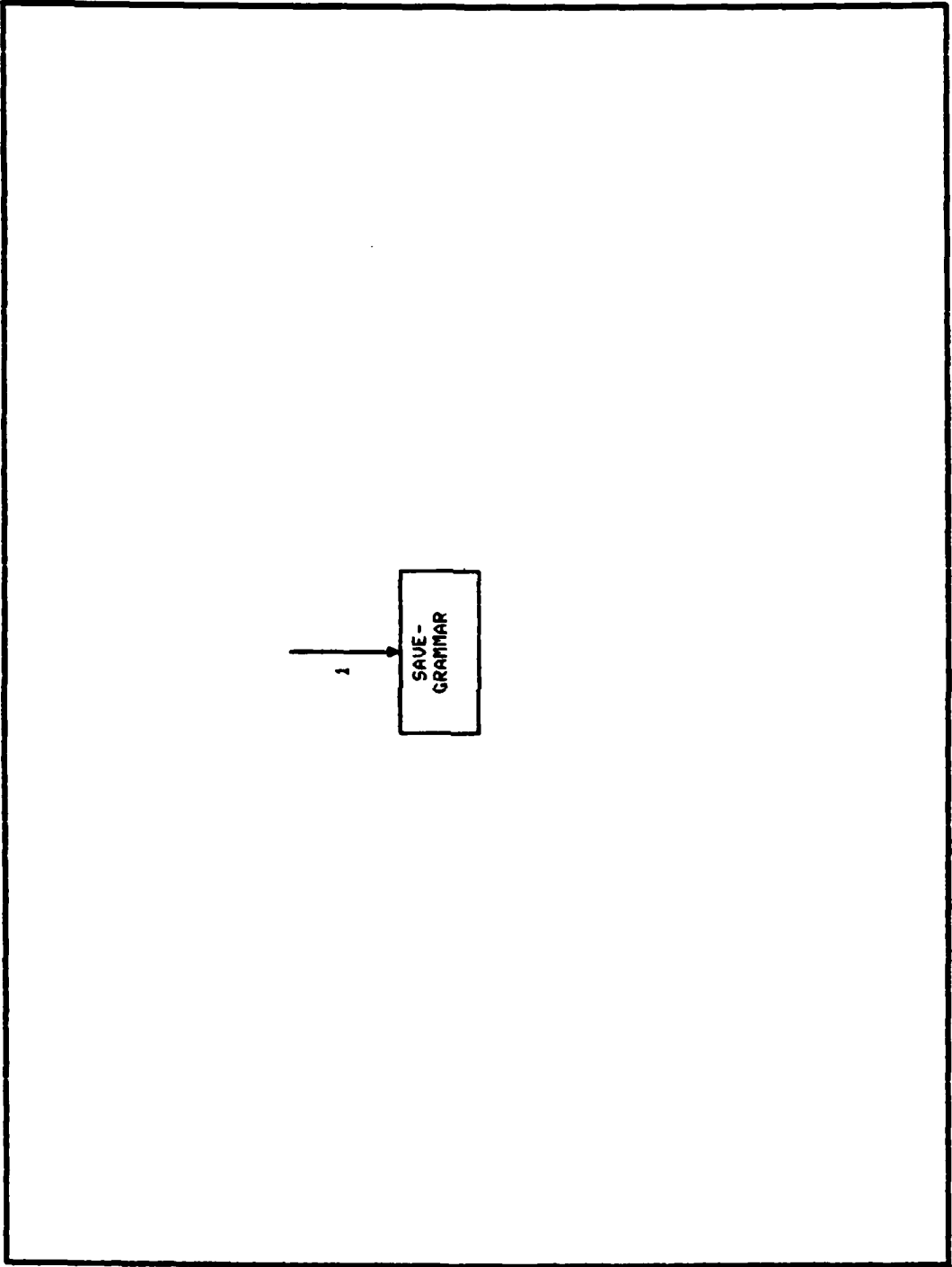
initialize-grammar Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			



modify-function Interfaces

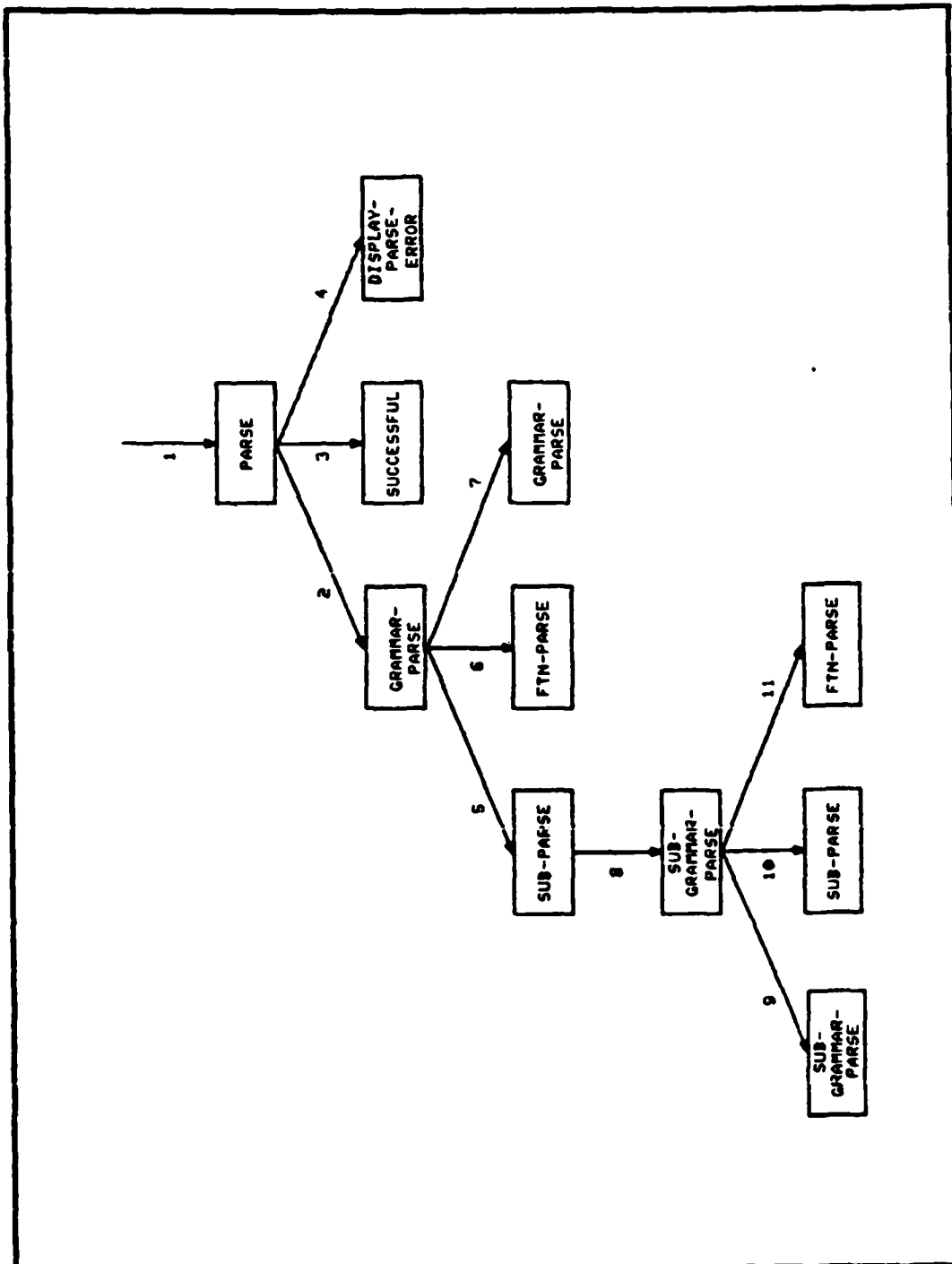
#	Passed Parameters	Type	Returned Parameters	Type
1	function name	data		
2	function name	data	is function?	flag
3	*edit-ptr*	data		
	edit-ptr	data		
	nil			
4	code	data		
5	error message			
	text	data		



save-grammar Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			

Sentence Interpreter

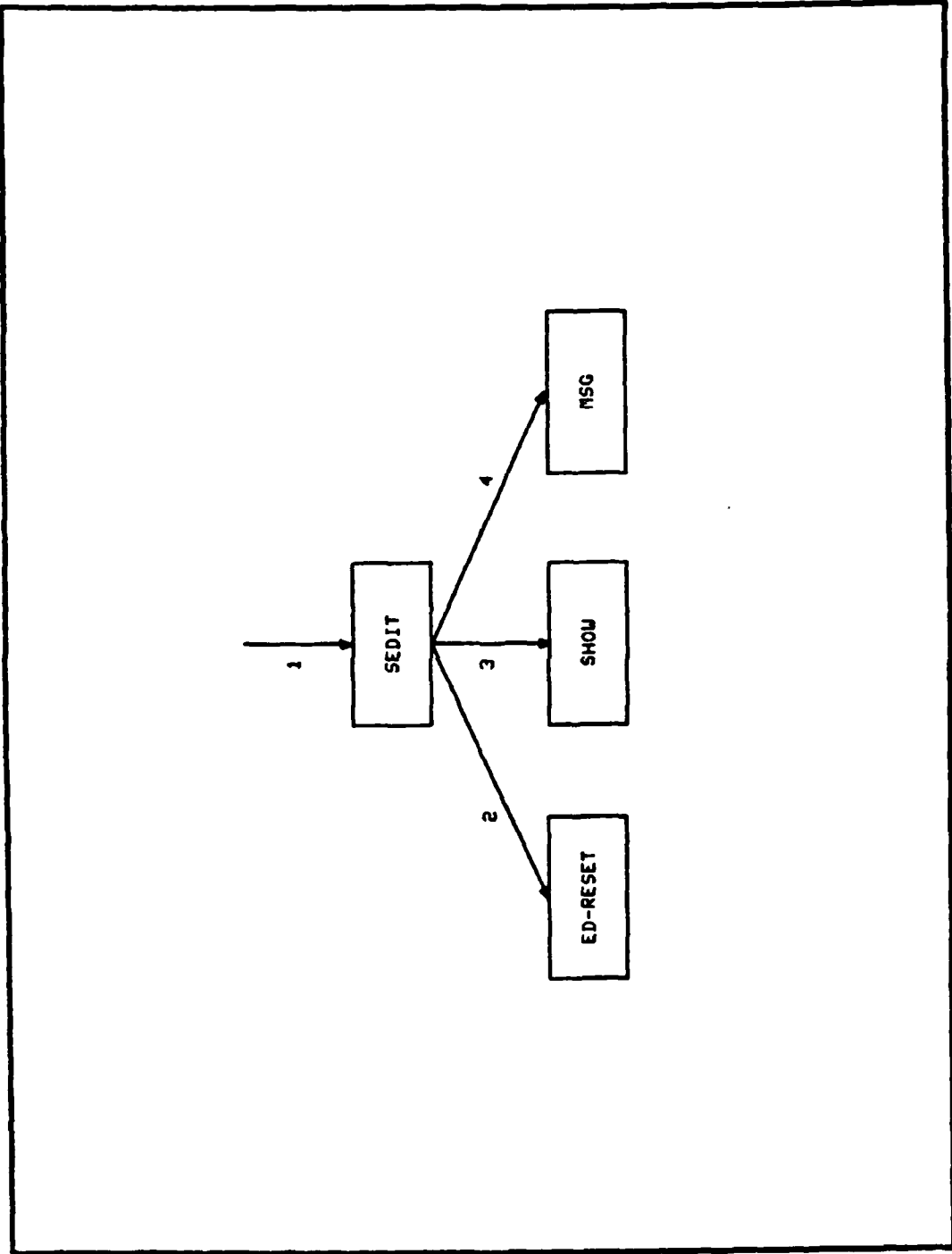


parse Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	sentence grammar	data data		
2	sentence sub-grammar-list	data data	result	data
3	result	data	successful parse?	flag
4	(none)			
5	sentence sub-grammar	data data	result	data
6	sentence function	data data	result	data
7	cdr*(sentence) sub-grammar-list	data data	result	data
8	sentence sub-grammar-list	data data	result	data
9	cdr*(sentence) sub-grammar-list	data data	result	data
10	sentence sub-grammar	data data	result	data
11	sentence function	data data	result	data

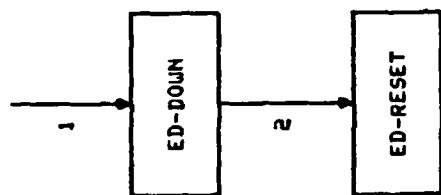
* The cdr function returns a list of all but the first element of a list passed to it. For instance, the cdr of the list (a b c) is the list (b c).

S-expression Editor



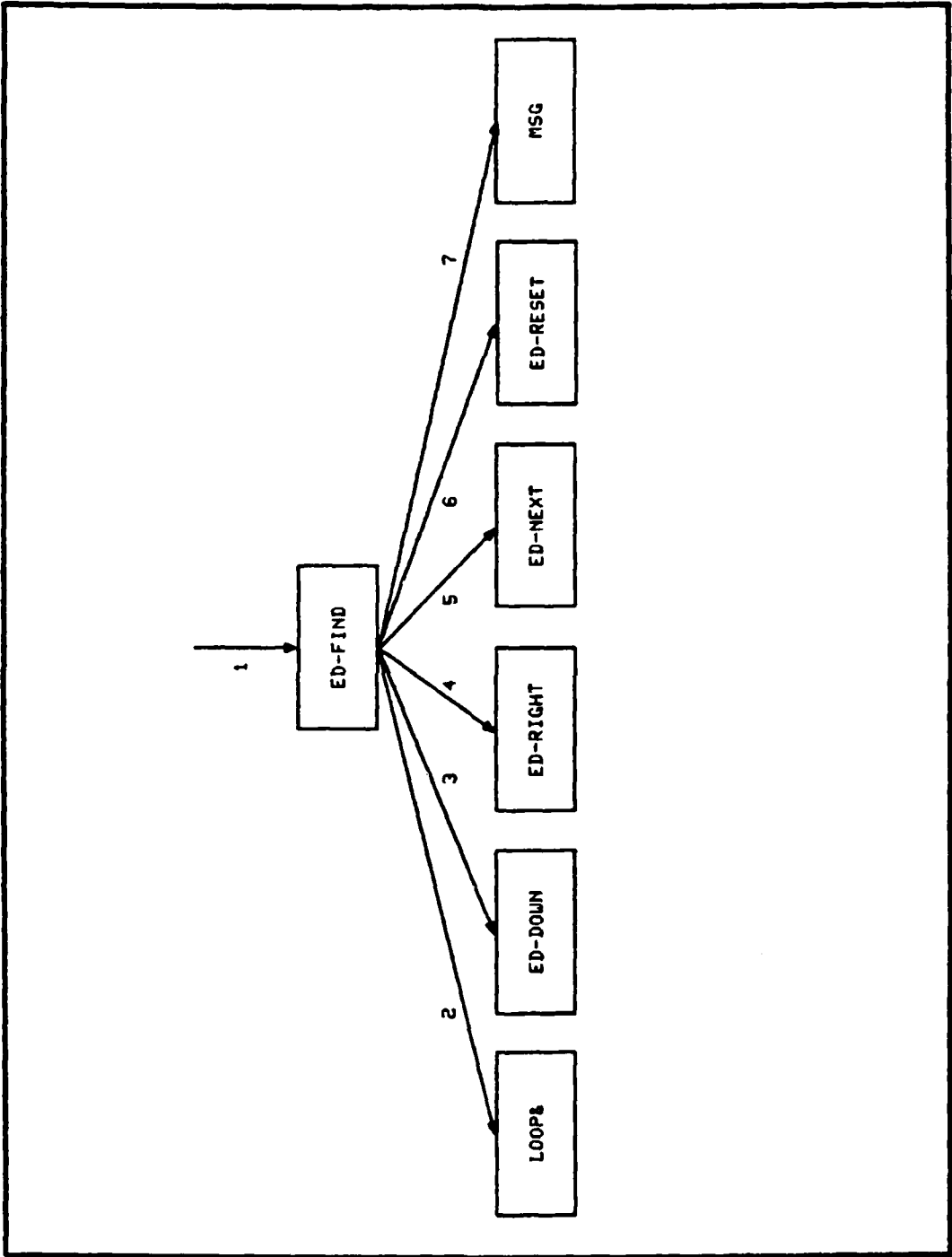
sedit Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	expression	data		
2	expression (edit pointer)	data		
	expression (edit line)	data		
	nil (edit stack)	data		
3	number of levels to display	data		
4	error message text	data		



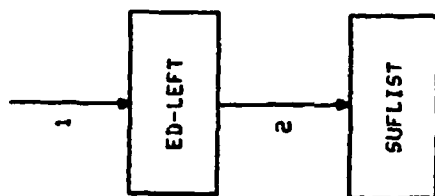
ed-down Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	new edit pointer	data		
	new edit line	data		
	new edit stack	data		



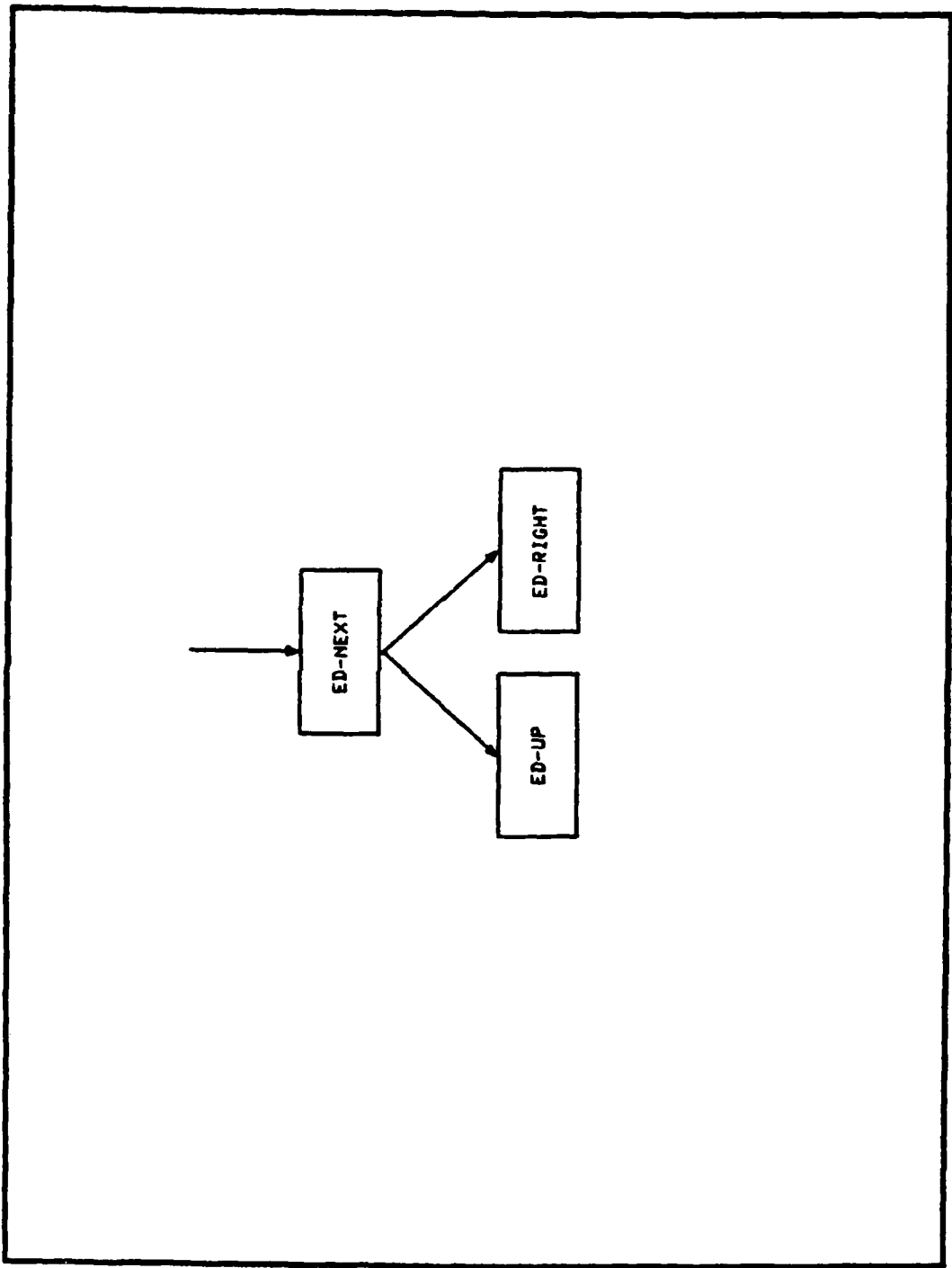
ed-find Interfaces

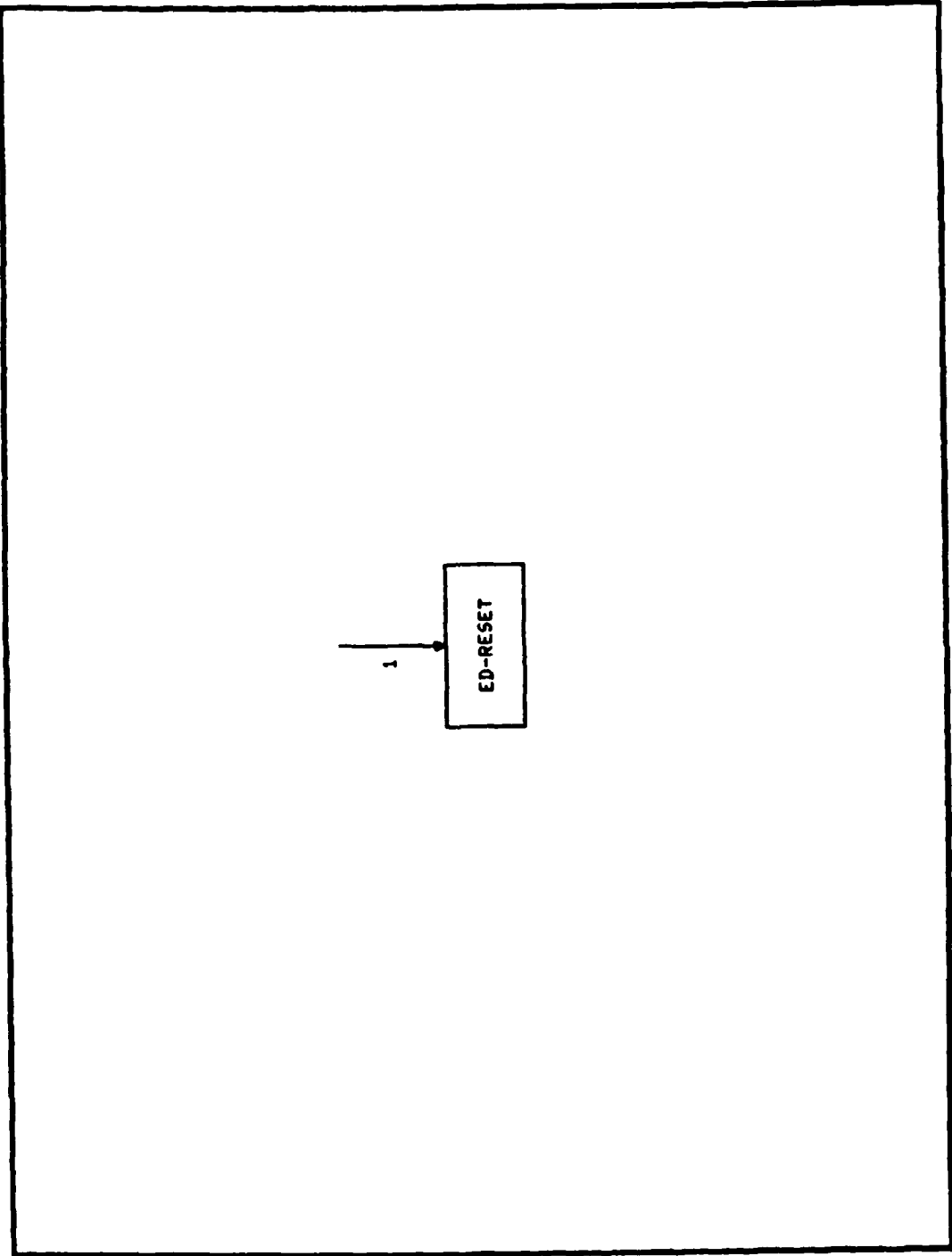
#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	*key*	data		
	edit pointer	data		
	edit line	data		
	edit stack	data		
3	(none)			
4	(none)			
5	(none)			
6	edit pointer	data		
	edit line	data		
	edit stack	data		
7	error message text	data		



ed-left Interfaces

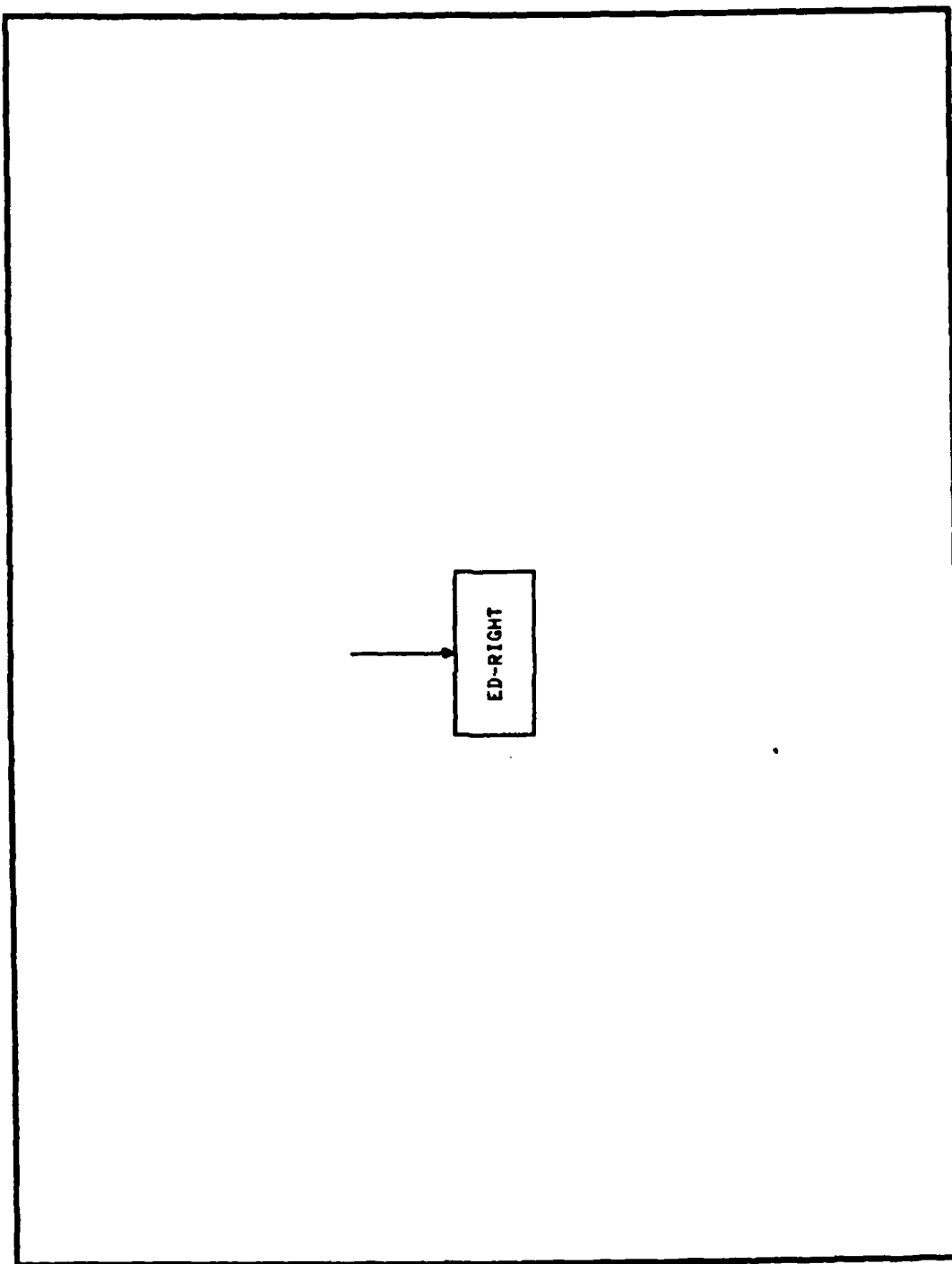
#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	edit line	data	new edit pointer	data
	current edit pointer position	data		

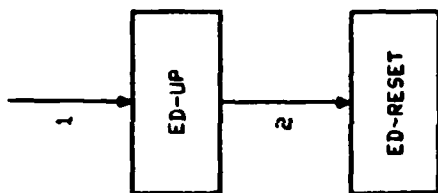




ed-reset Interfaces

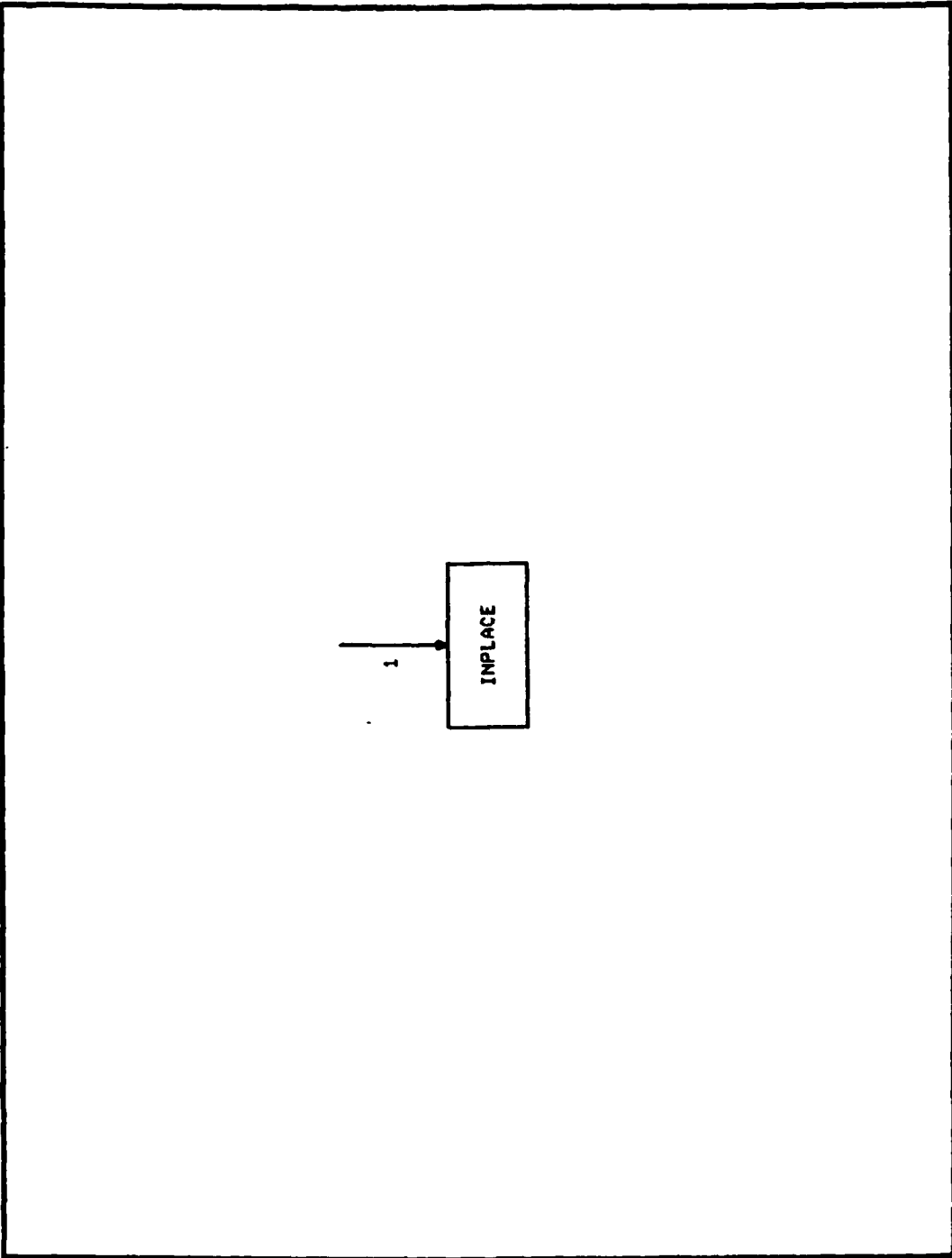
#	Passed Parameters	Type	Returned Parameters	Type
1	new edit pointer new edit line new edit stack	data data data		





ed-up Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	new edit pointer	data		
	new edit line	data		
	new edit stack	data		

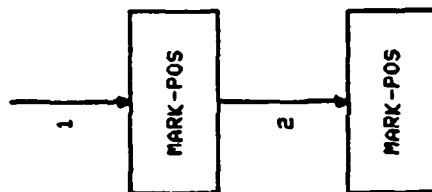


inplace Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	car*(new edit pointer)	data		
	cdr+(new edit pointer)	data		

* The car function returns the first element of a list passed to it. For instance, the car of the list (a b c) is a.

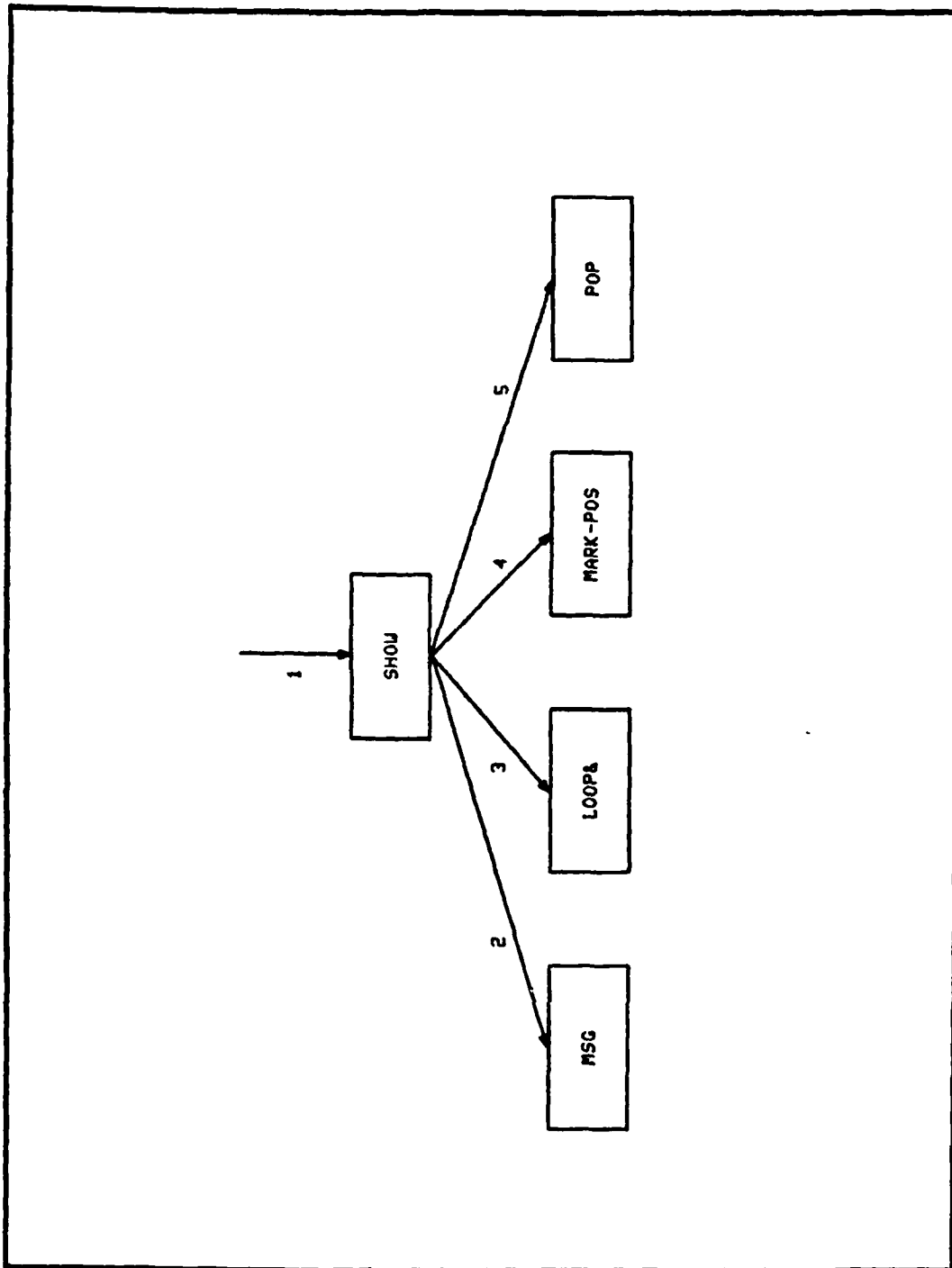
+ The cdr function returns a list of all but the first element of a list passed to it. The cdr of the above list is the list (b c).



mark-pos Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	position to mark list	data data		
2	position to mark cdr*(list)	data data		

* The cdr function returns a list of all but the first element of a list passed to it. For instance, the cdr of the list (a b c) is the list (b c).

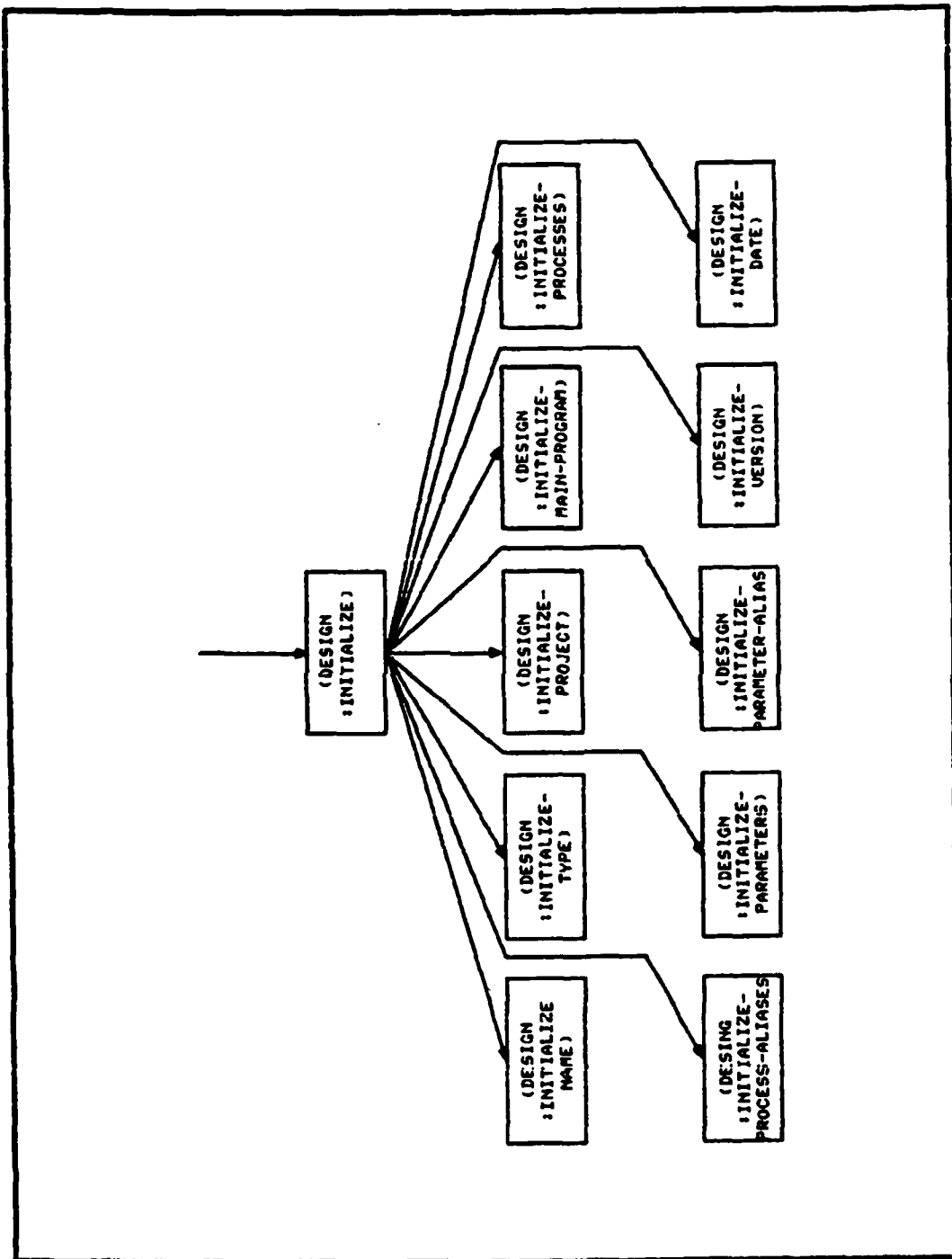


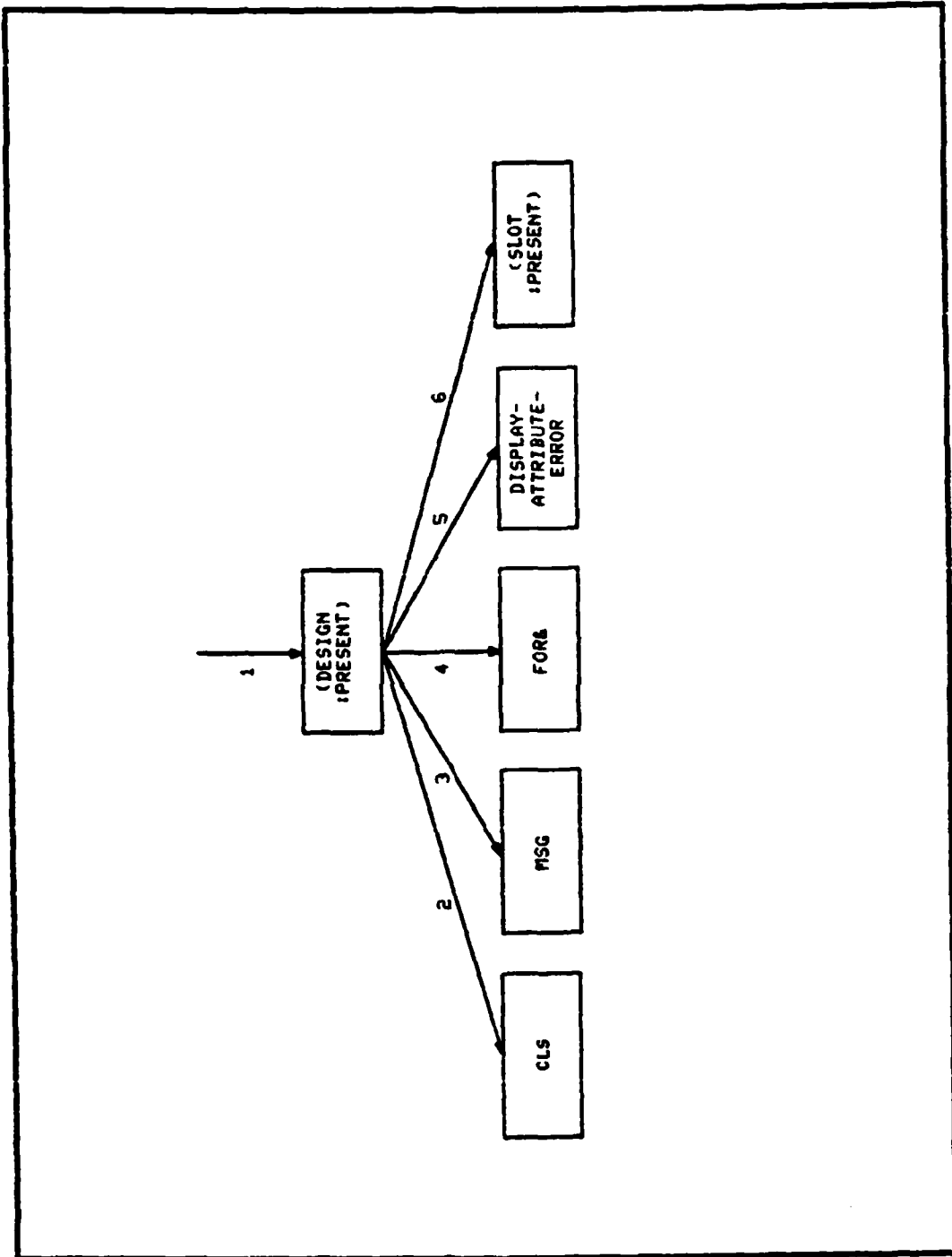
show Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	number of levels of the edit stack to display	data		
2	car*(edit pointer)	data		
3	edit pointer edit line edit stack	data data data		
4	edit pointer edit line	data data		
5	stack	data	top of stack	data

* The car function returns the first element of a list passed to it. For instance, the car of the list (a b c) is a.

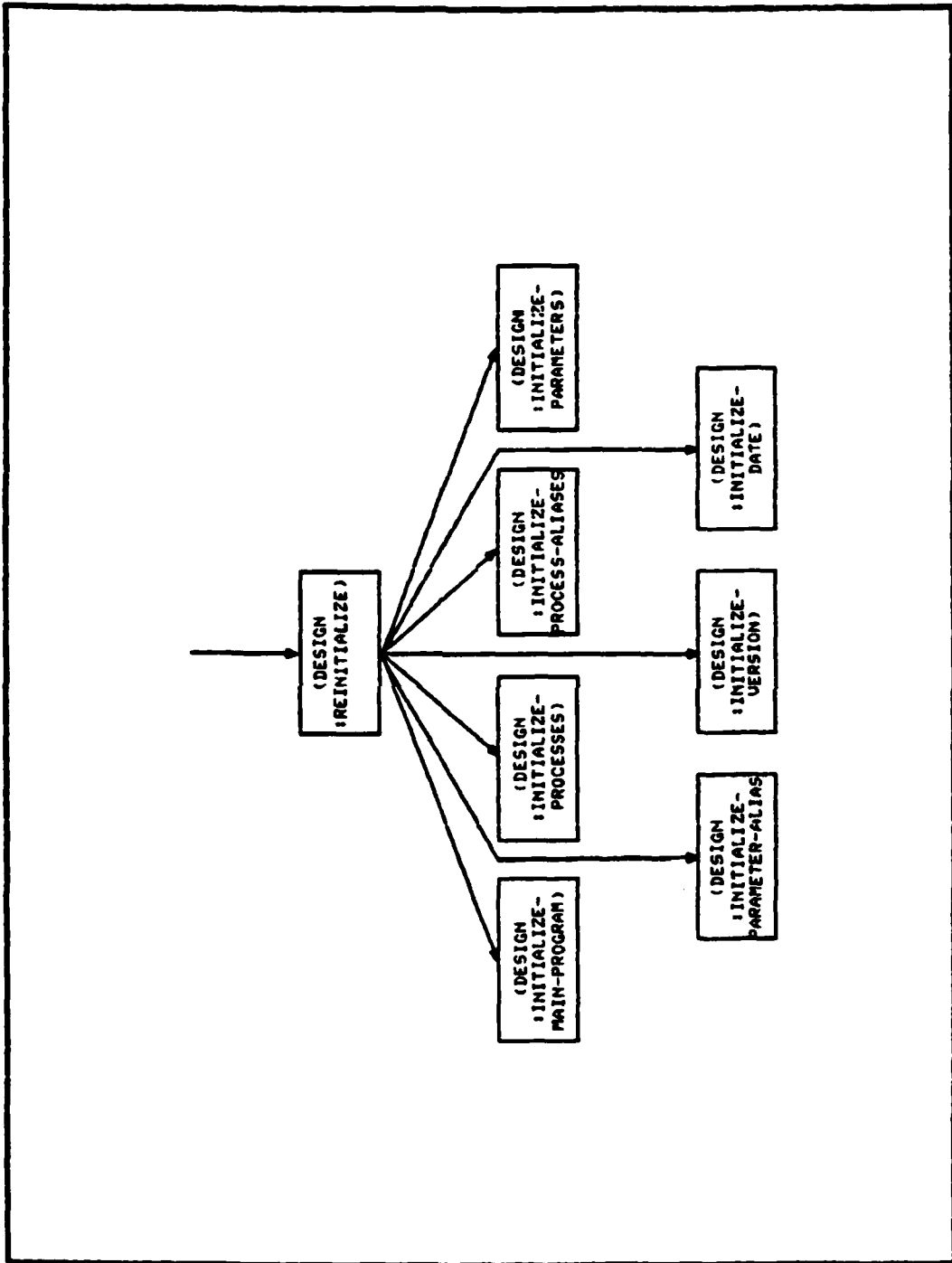
DESIGN Methods

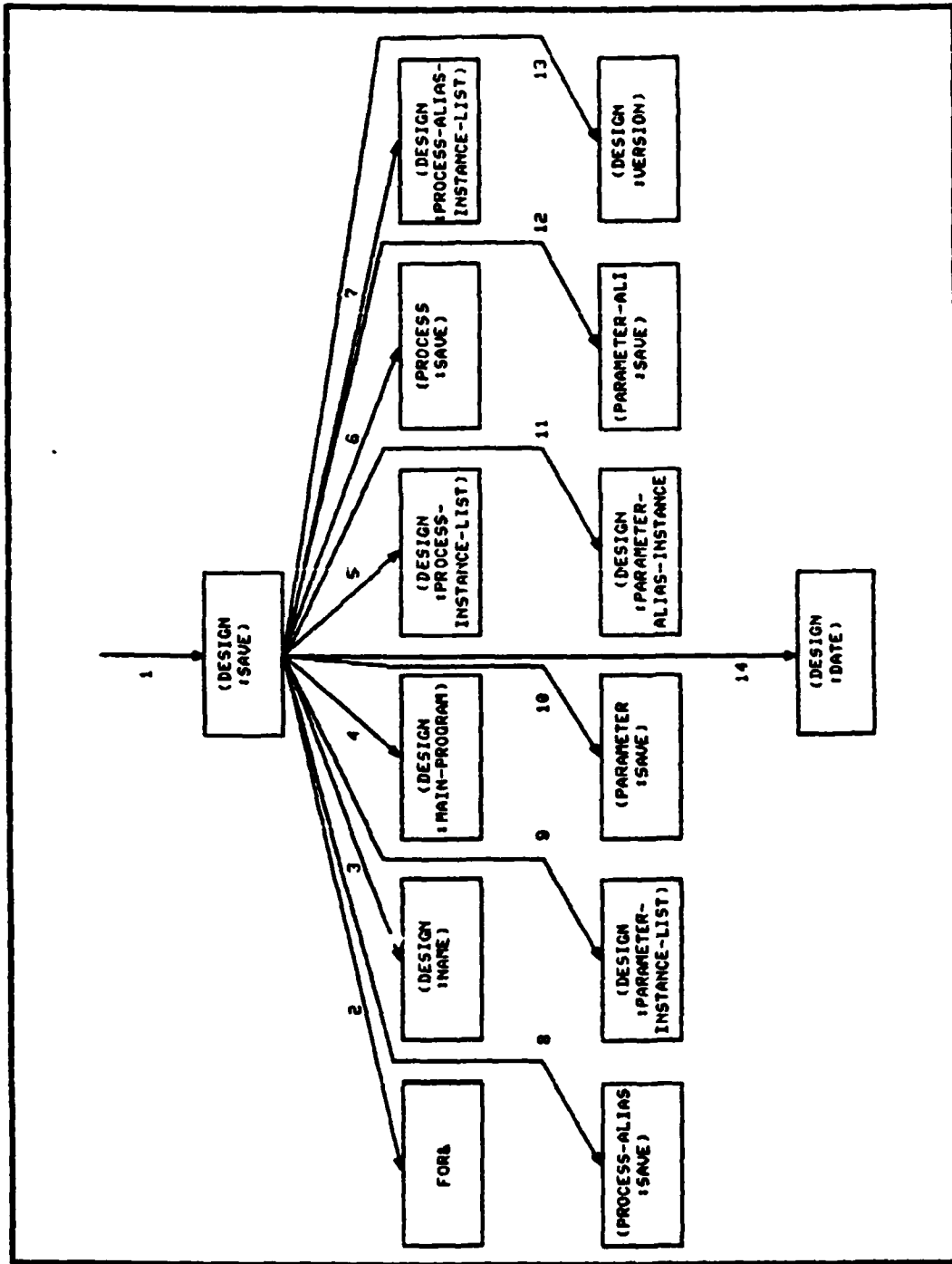




(design :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables		data	
3	(none)			
4	list of instance variables		data	
5	list of invalid instance variables		data	
6	(none)			

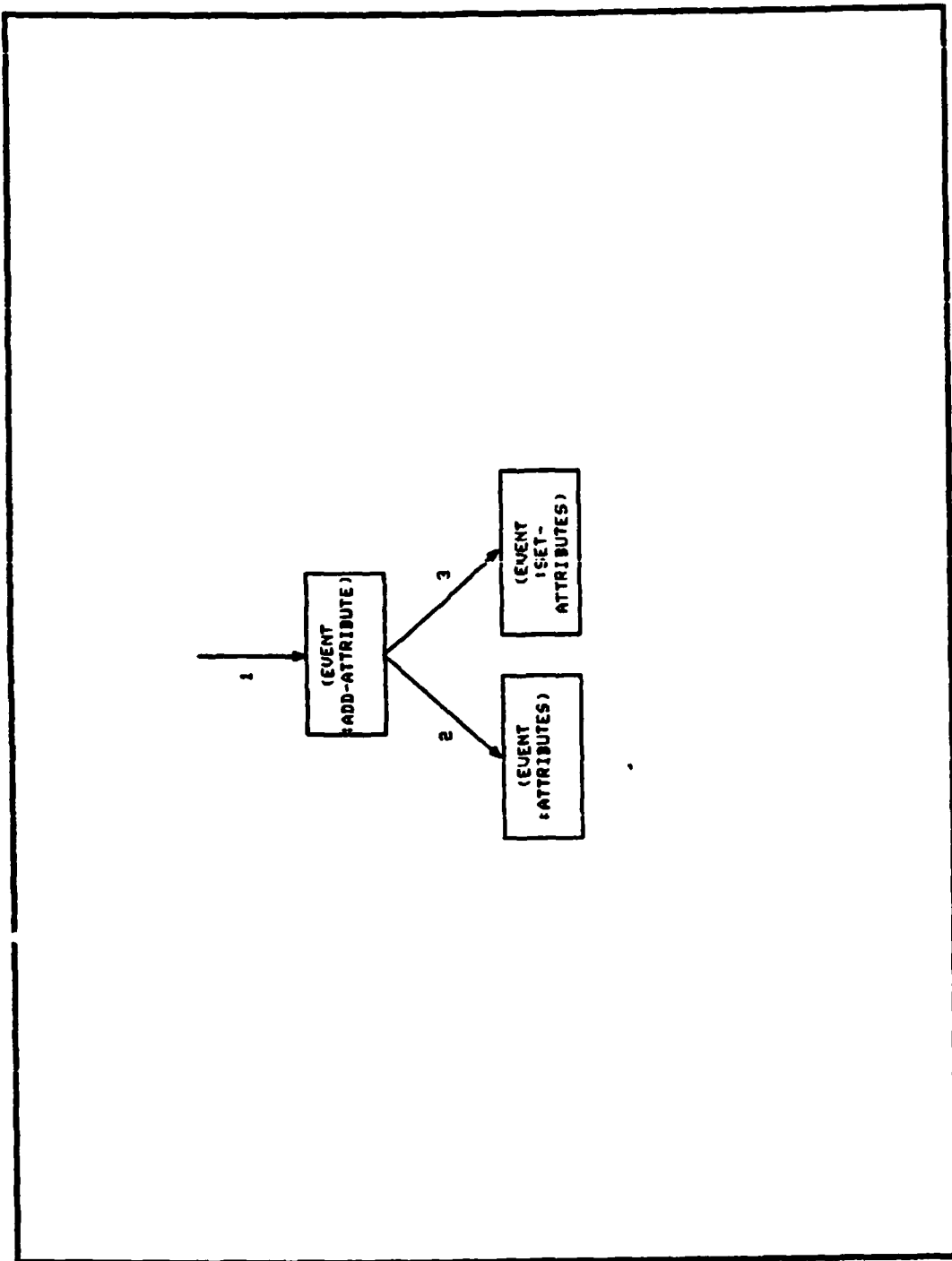




(design :save) Interfaces

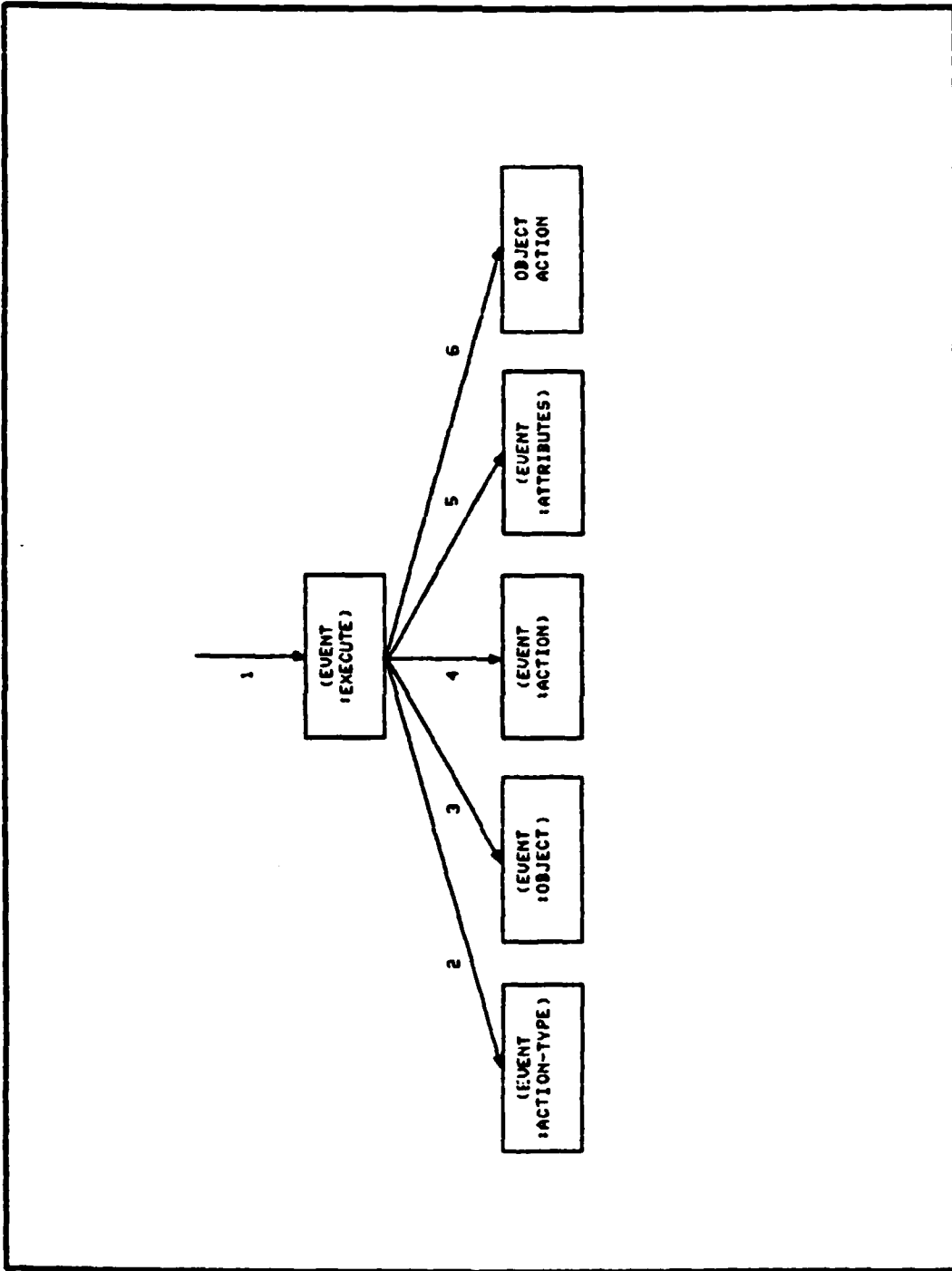
#	Passed Parameters	Type	Returned Parameters	Type
1	project instance pointer stream	data data		
2	list of instance pointers	data		
3	(none)		name	data
4	(none)		main-program	data
5	(none)		list of process instance pointers	data
6	process instance pointer	data		
7	(none)		list of process- alias instance pointers	data
8	process-alias instance pointer	data		
9	(none)		list of parameter instance pointers	data
10	parameter instance pointer	data		
11	(none)		list of parameter- alias instance pointers	data
12	parameter-alias instance pointer	data		
13	(none)		version	data
14	(none)		date	data

EVENT Methods



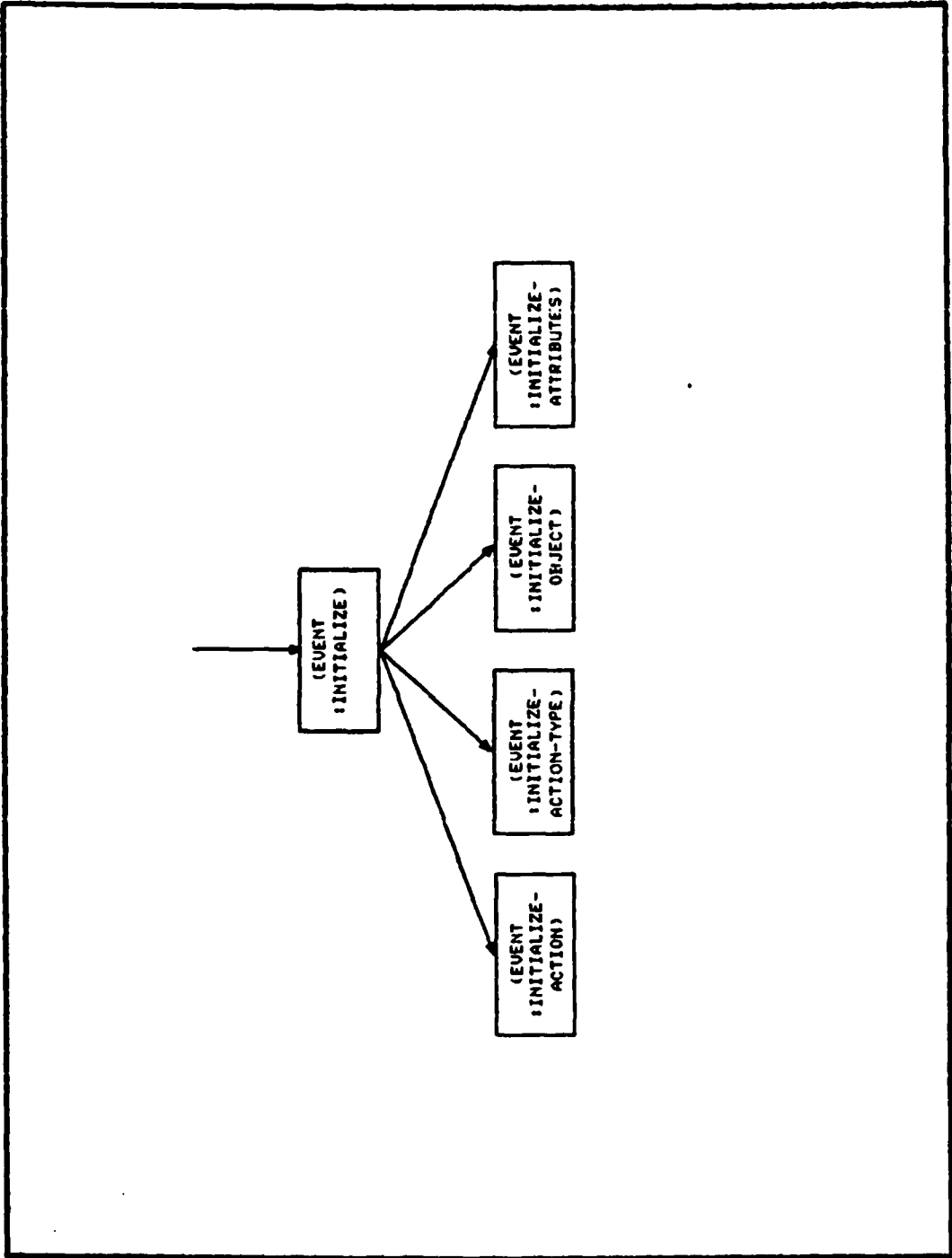
(event :add-attribute) Interfaces

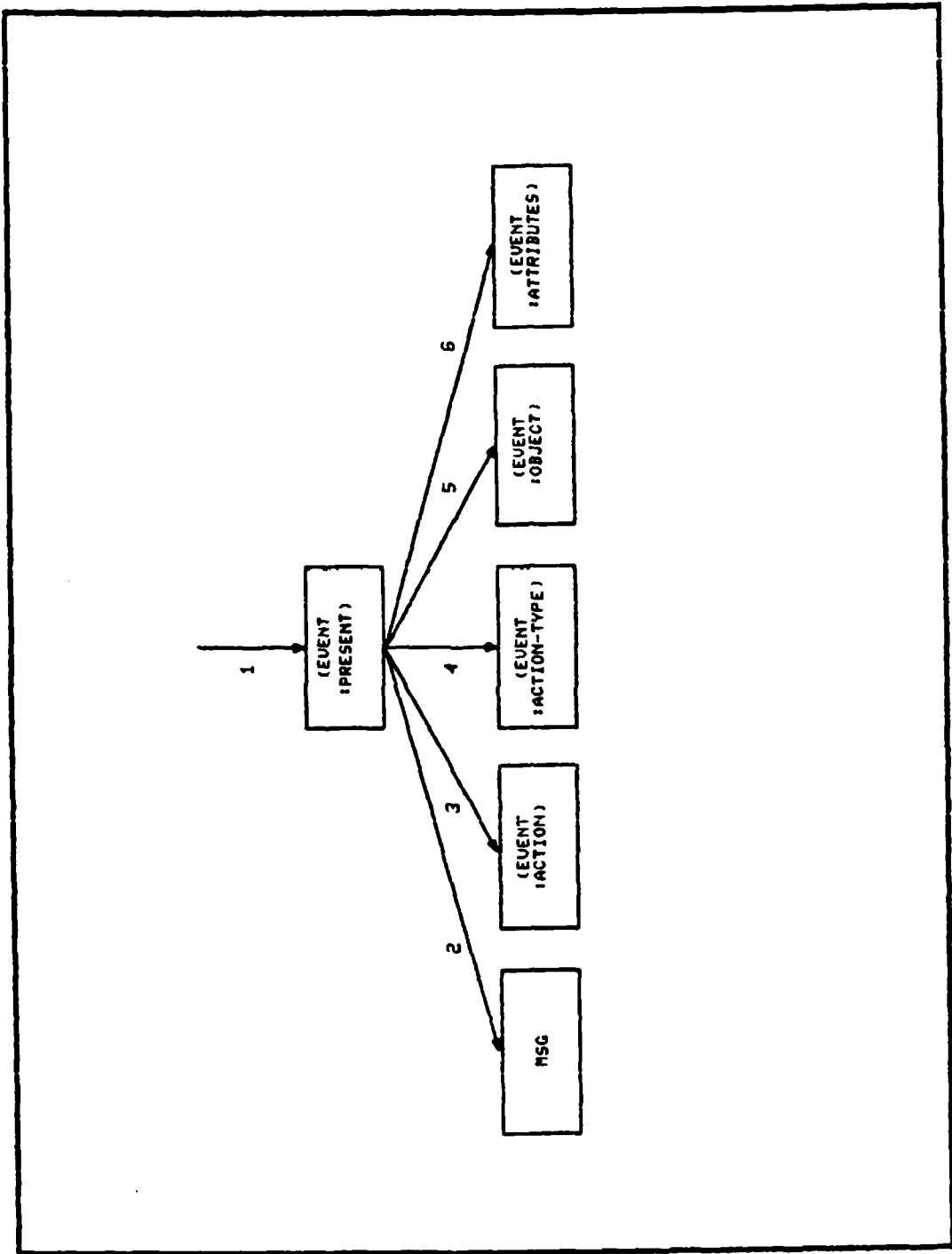
#	Passed Parameters	Type	Returned Parameters	Type
1	attribute	data		
2	(none)		list of attributes	data
3	list of attributes	data		



(event :execute) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	(none)		action type	data
3	(none)		object	data
4	(none)		action	data
5	(none)		attributes	data
6	object action attributes	data data data		

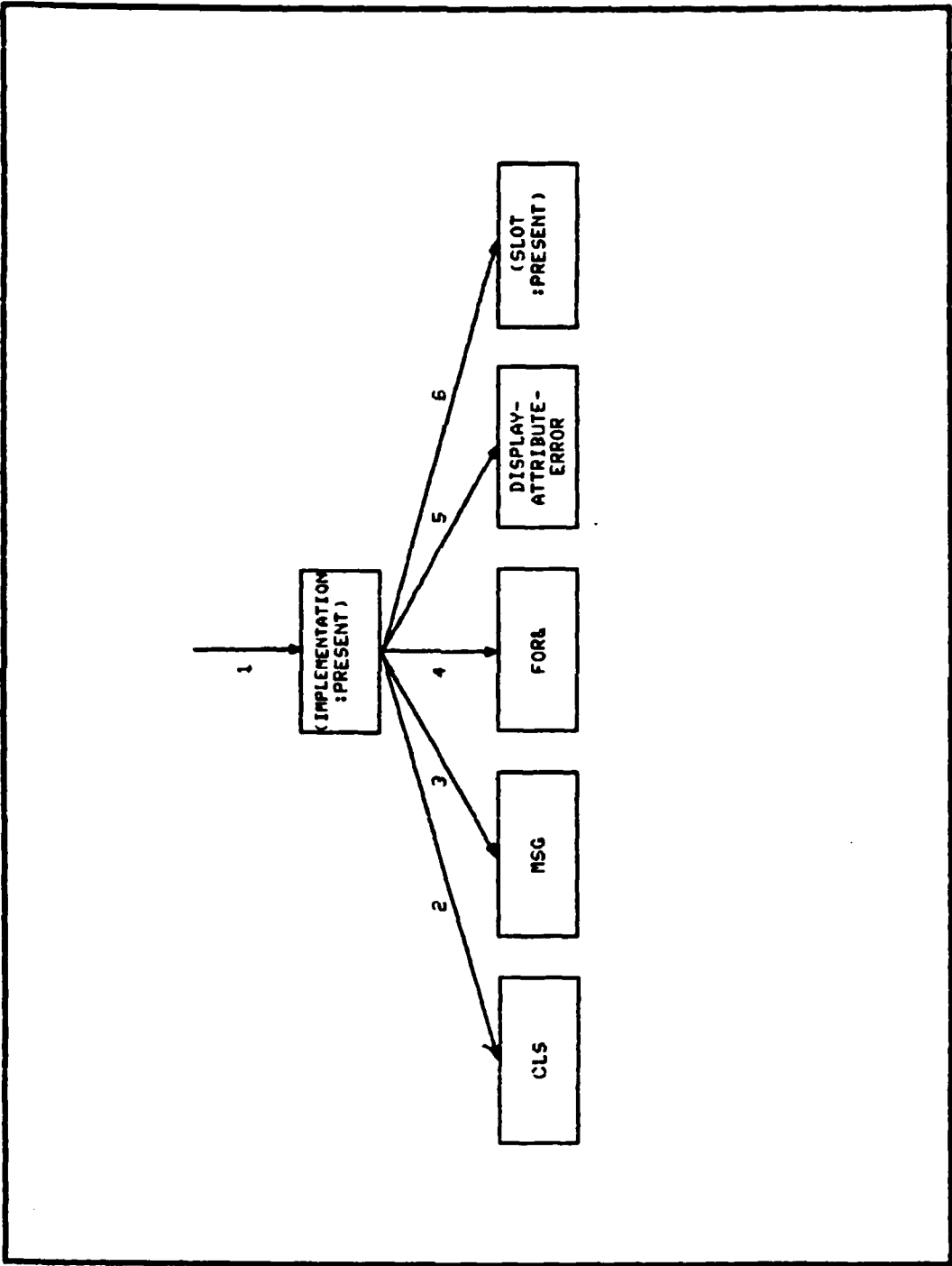




(event :present) Interfaces

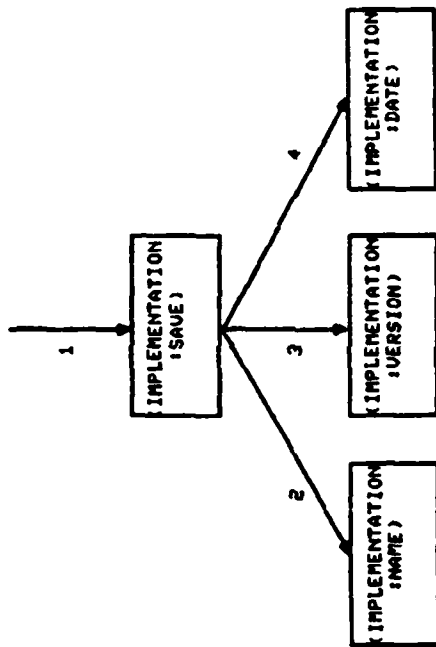
#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	data to display	data		
3	(none)		action	data
4	(none)		action-type	data
5	(none)		object	data
6	(none)		list of attributes	data

IMPLEMENTATION Methods



(implementation :present) Interfaces

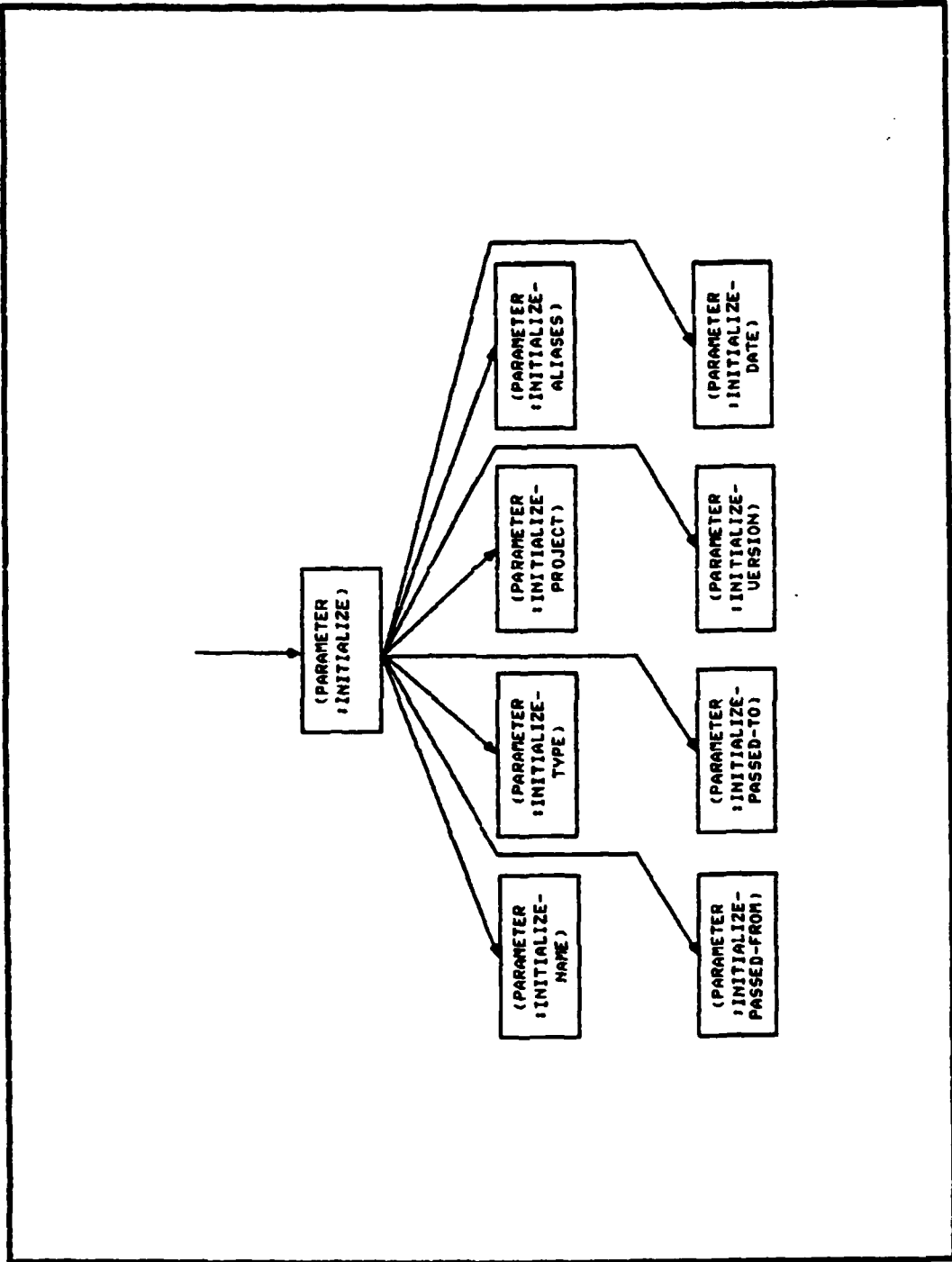
#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables		data	
3	(none)			
4	list of instance variables		data	
5	list of invalid instance variables		data	
6	(none)			

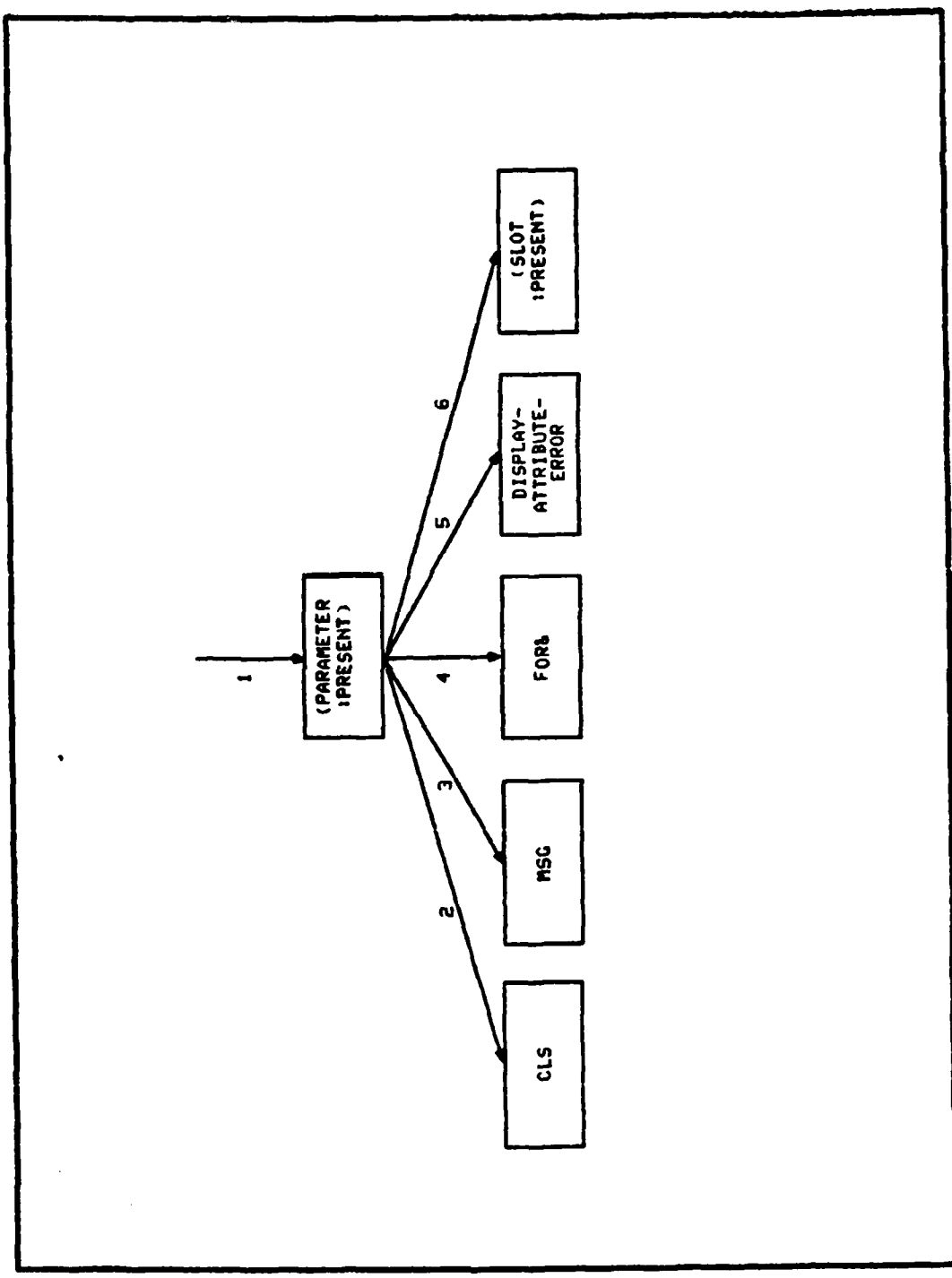


(implementation :save) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	project instance pointer stream	data data		
2	(none)		name	data
3	(none)		version	data
4	(none)		date	data

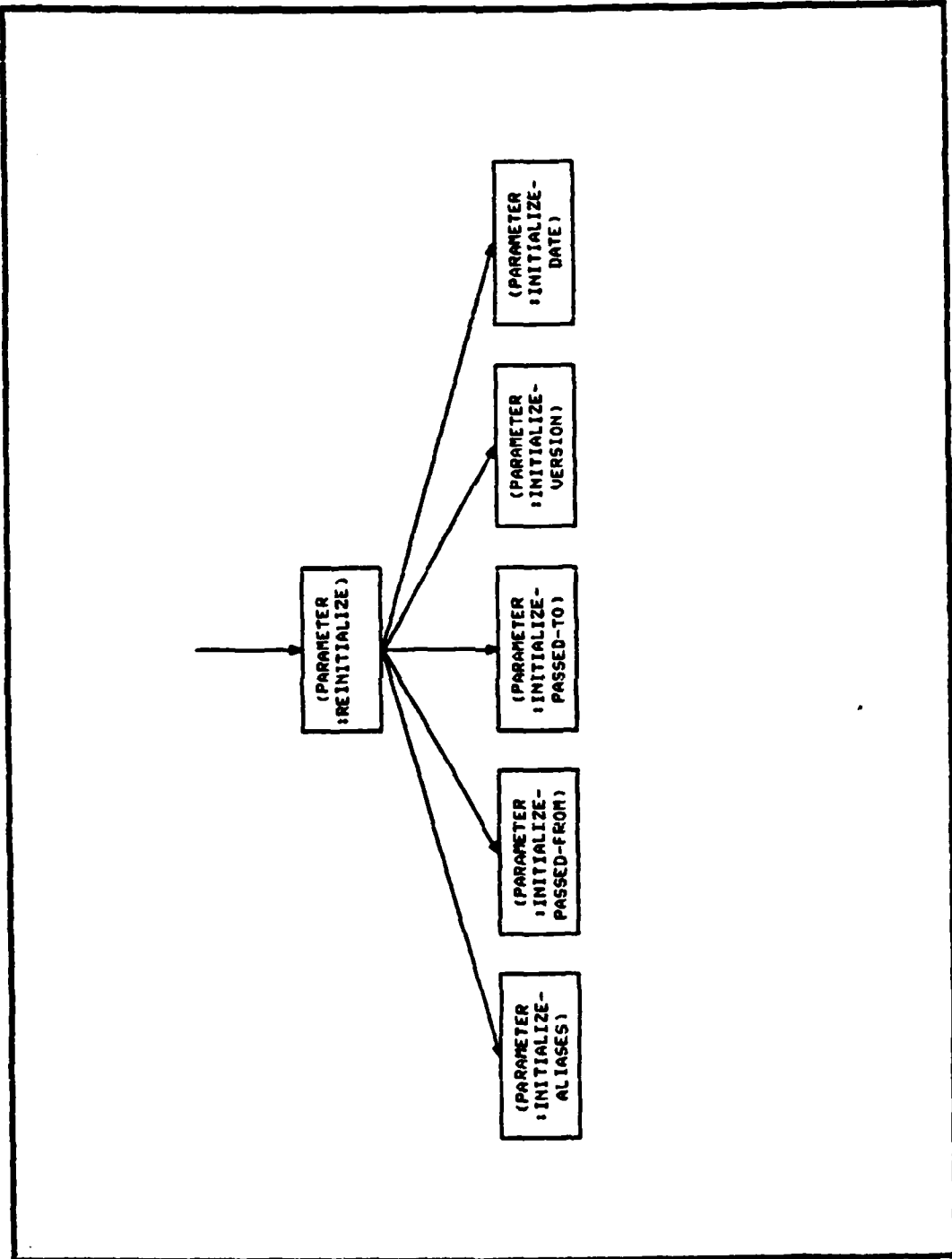
PARAMETER Methods

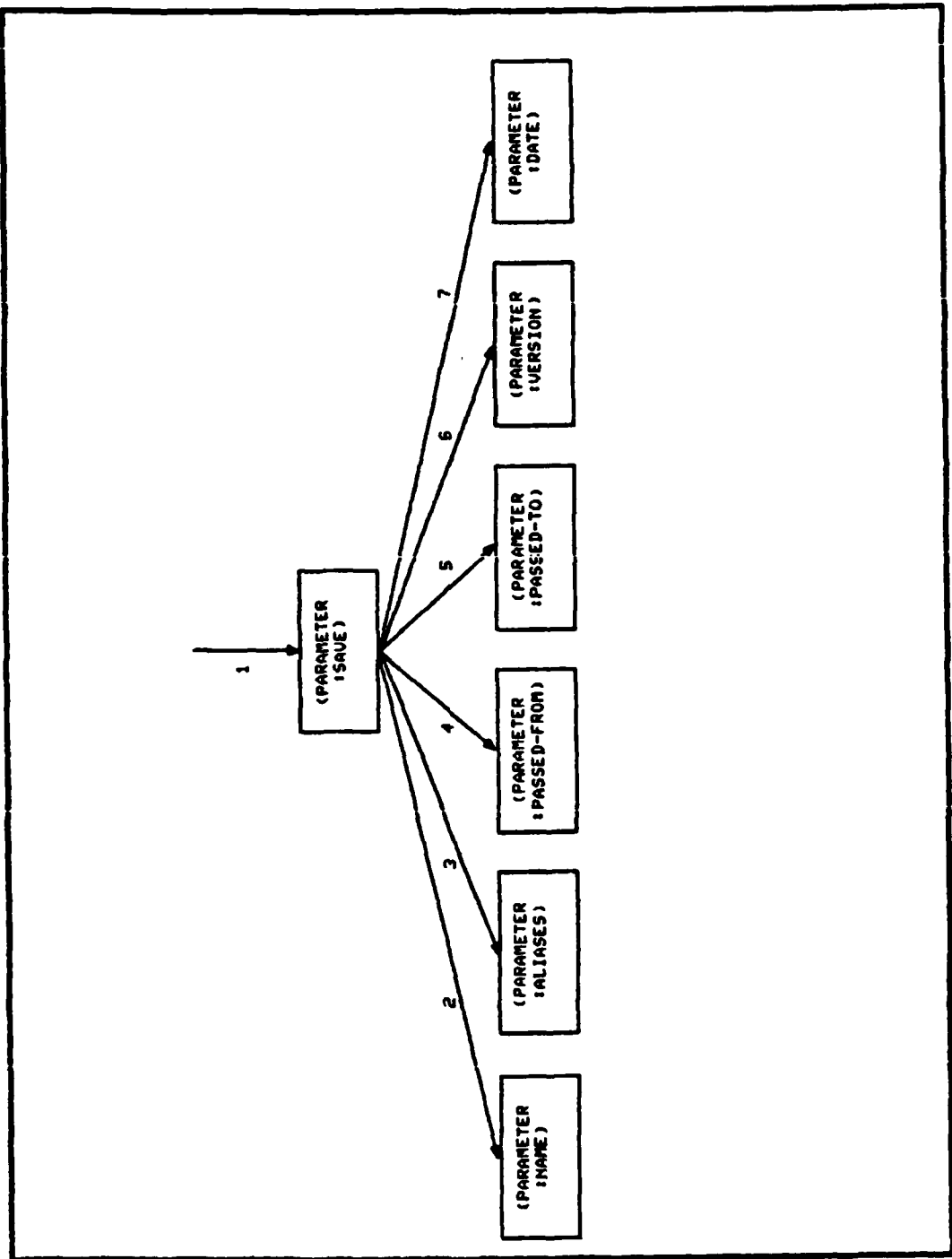




(parameter :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables		data	
3	(none)			
4	list of instance variables		data	
5	list of invalid instance variables		data	
6	(none)			

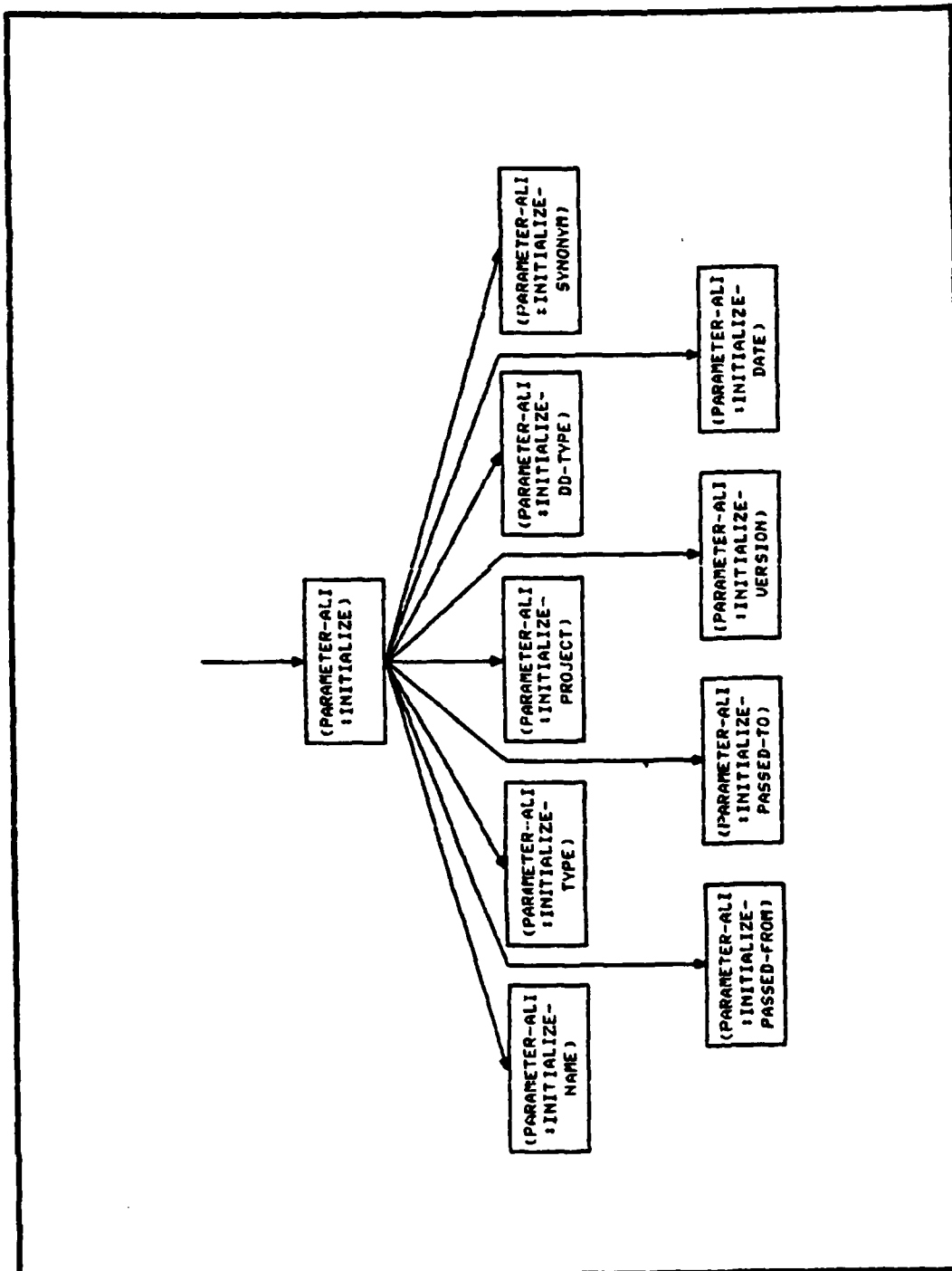


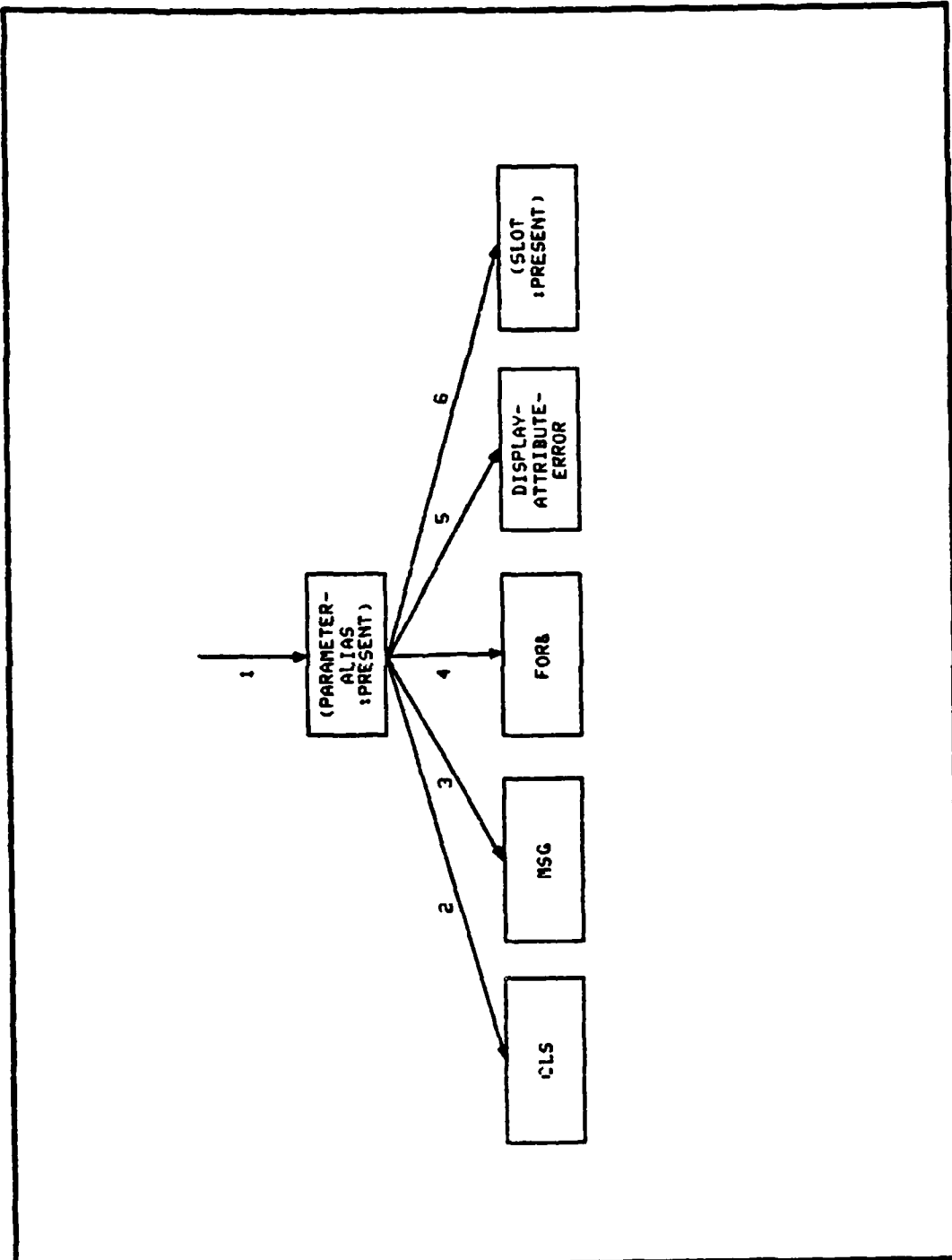


(parameter :save) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	design instance pointer stream	data data		
2	(none)		name	data
3	(none)		aliases	data
4	(none)		passed-from	data
5	(none)		passed-to	data
6	(none)		version	data
7	(none)		date	data

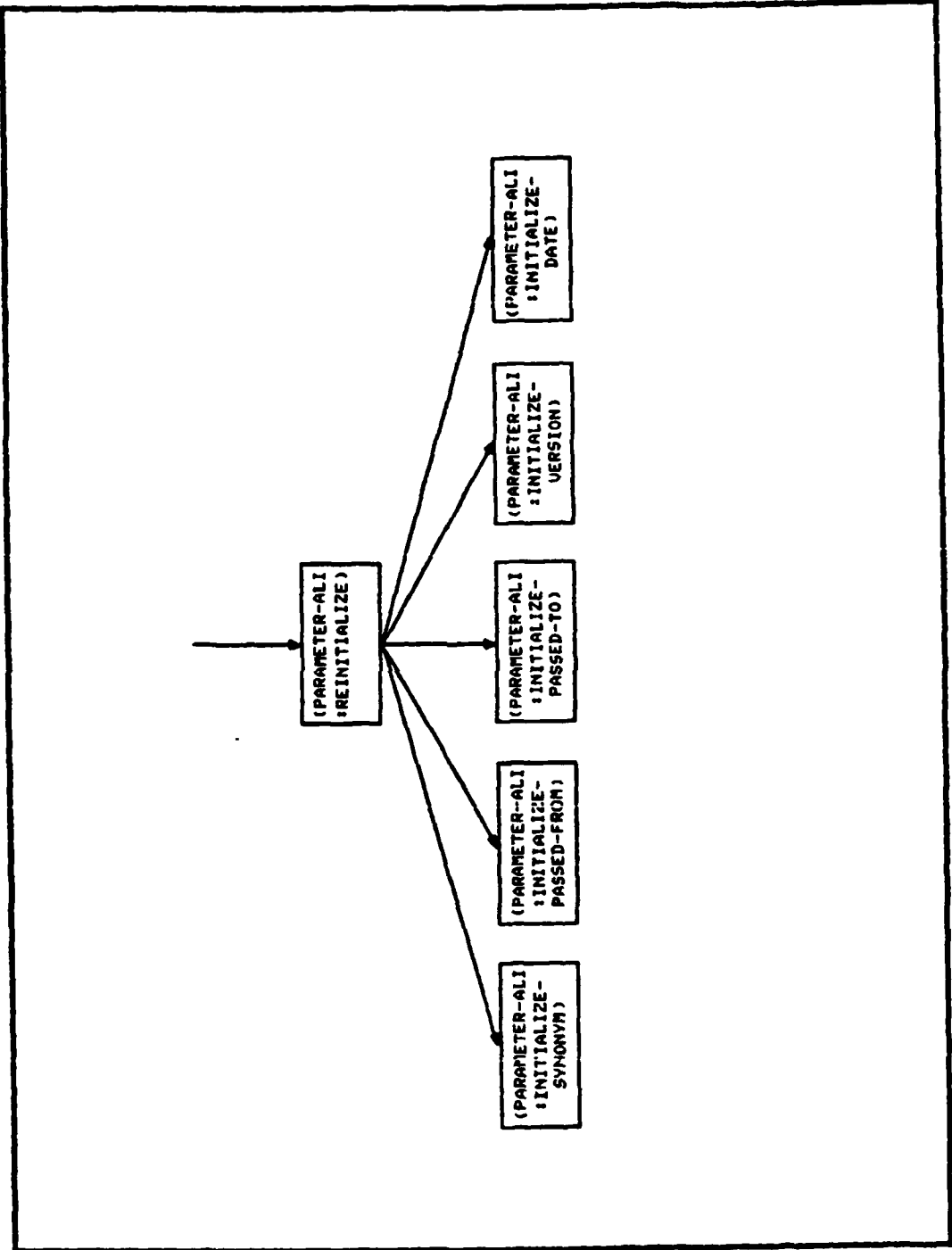
PARAMETER-ALIAS Methods

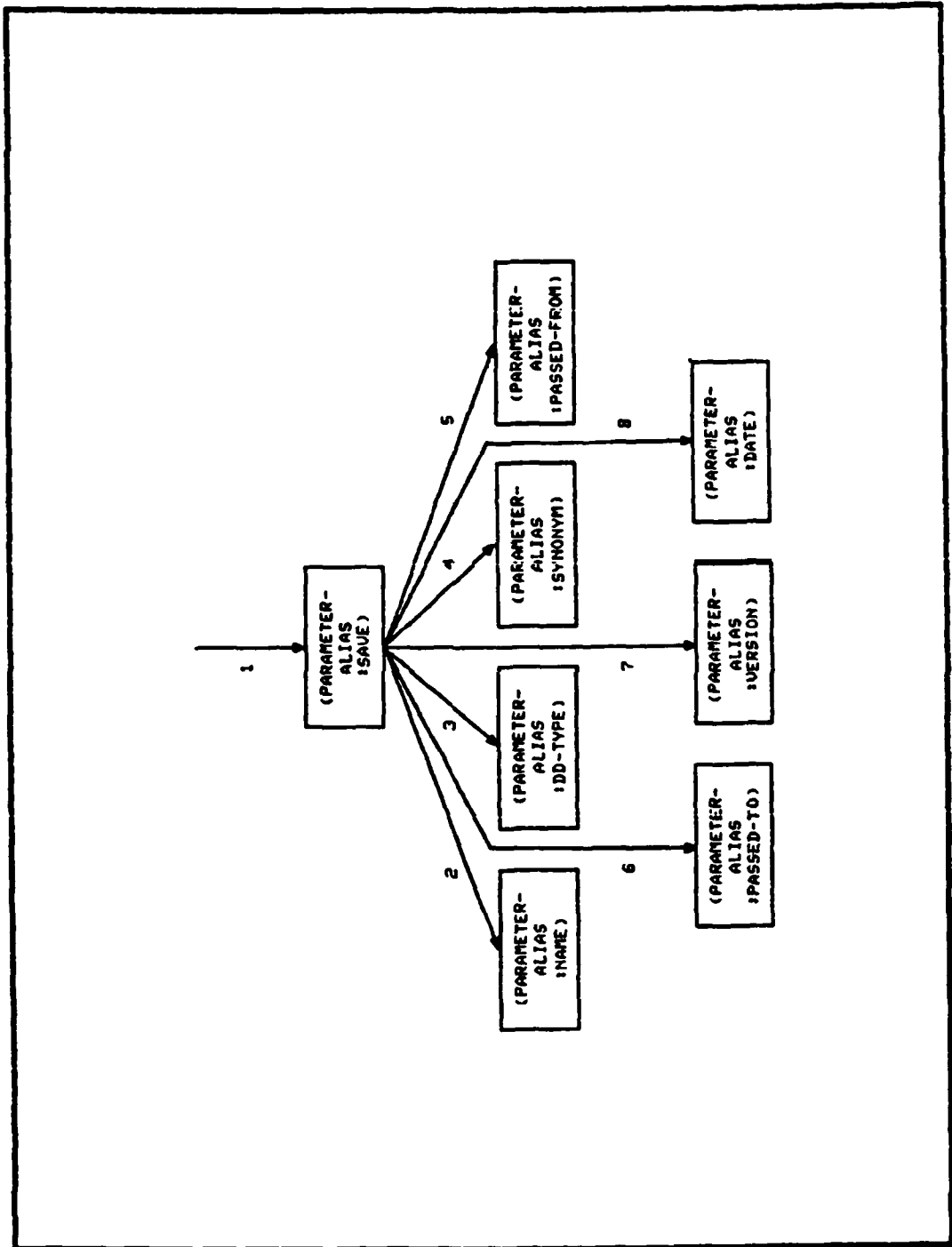




(parameter-alias :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables	data		
3	(none)			
4	list of instance variables	data		
5	list of invalid instance variables	data		
6	(none)			

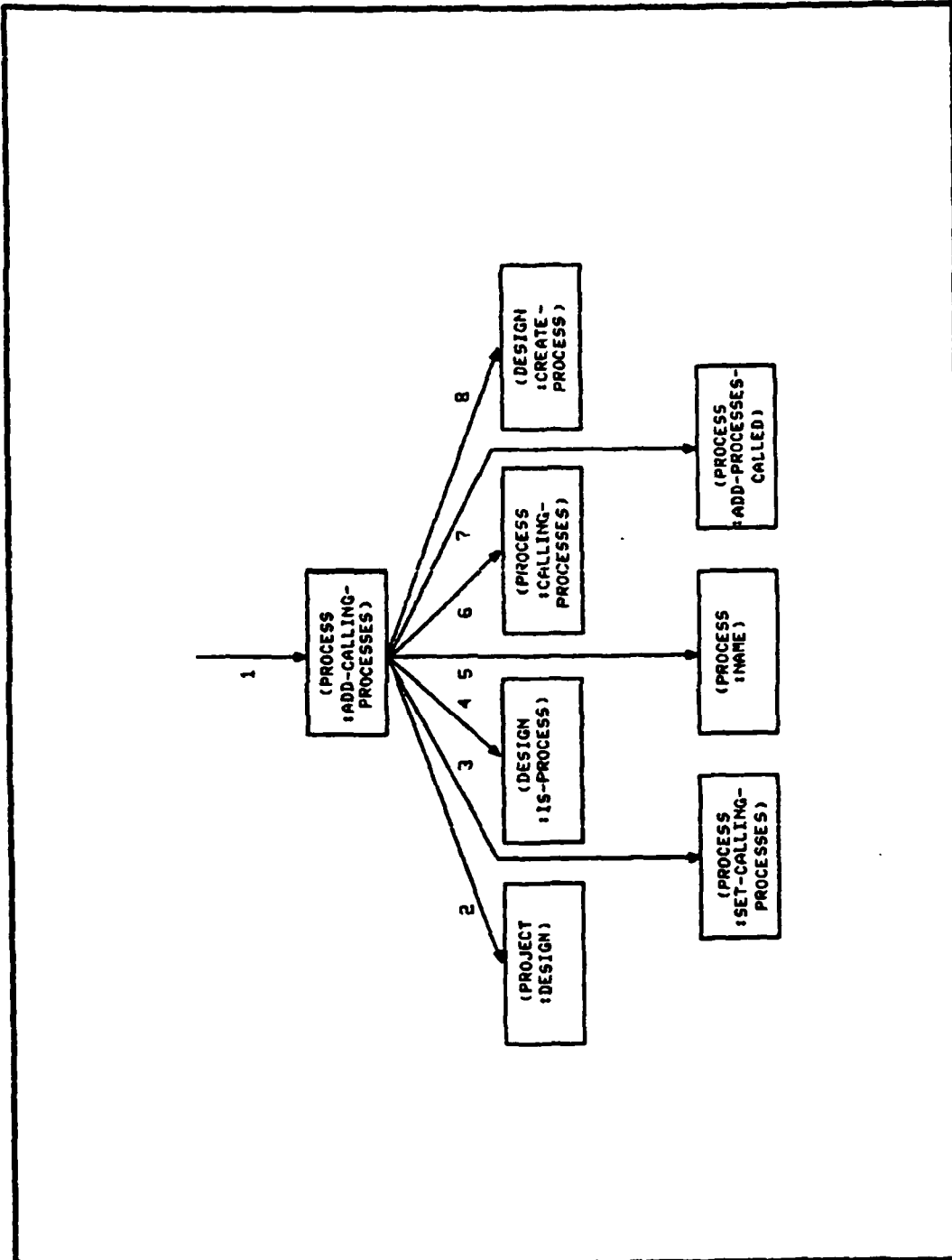




(parameter-alias :save) Interfaces

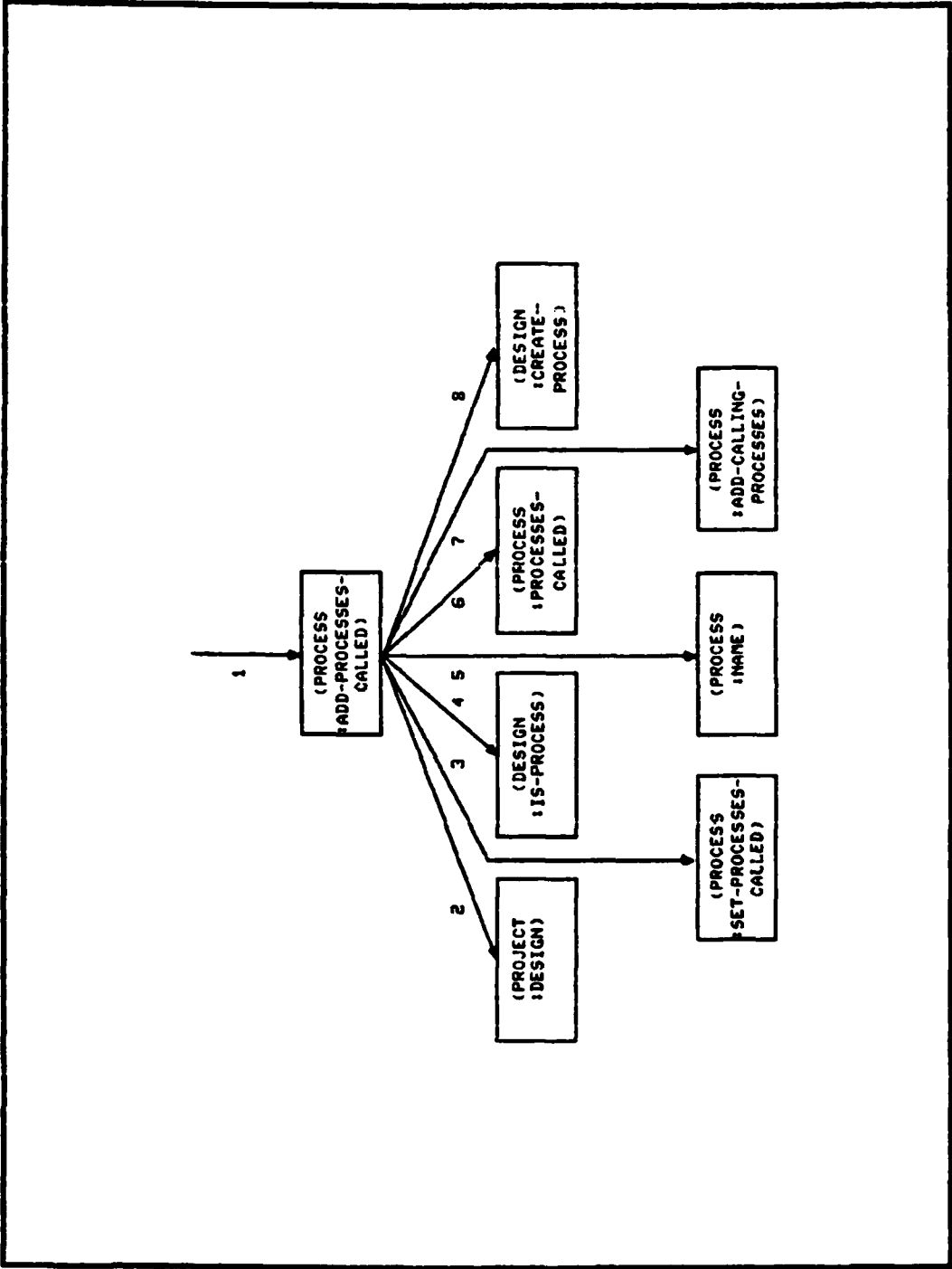
#	Passed Parameters	Type	Returned Parameters	Type
1	design instance pointer stream	data data		
2	(none)		name	data
3	(none)		dd-type	data
4	(none)		synonym	data
5	(none)		passed-from	data
6	(none)		passed-to	data
7	(none)		version	data
8	(none)		date	data

PROCESS Methods



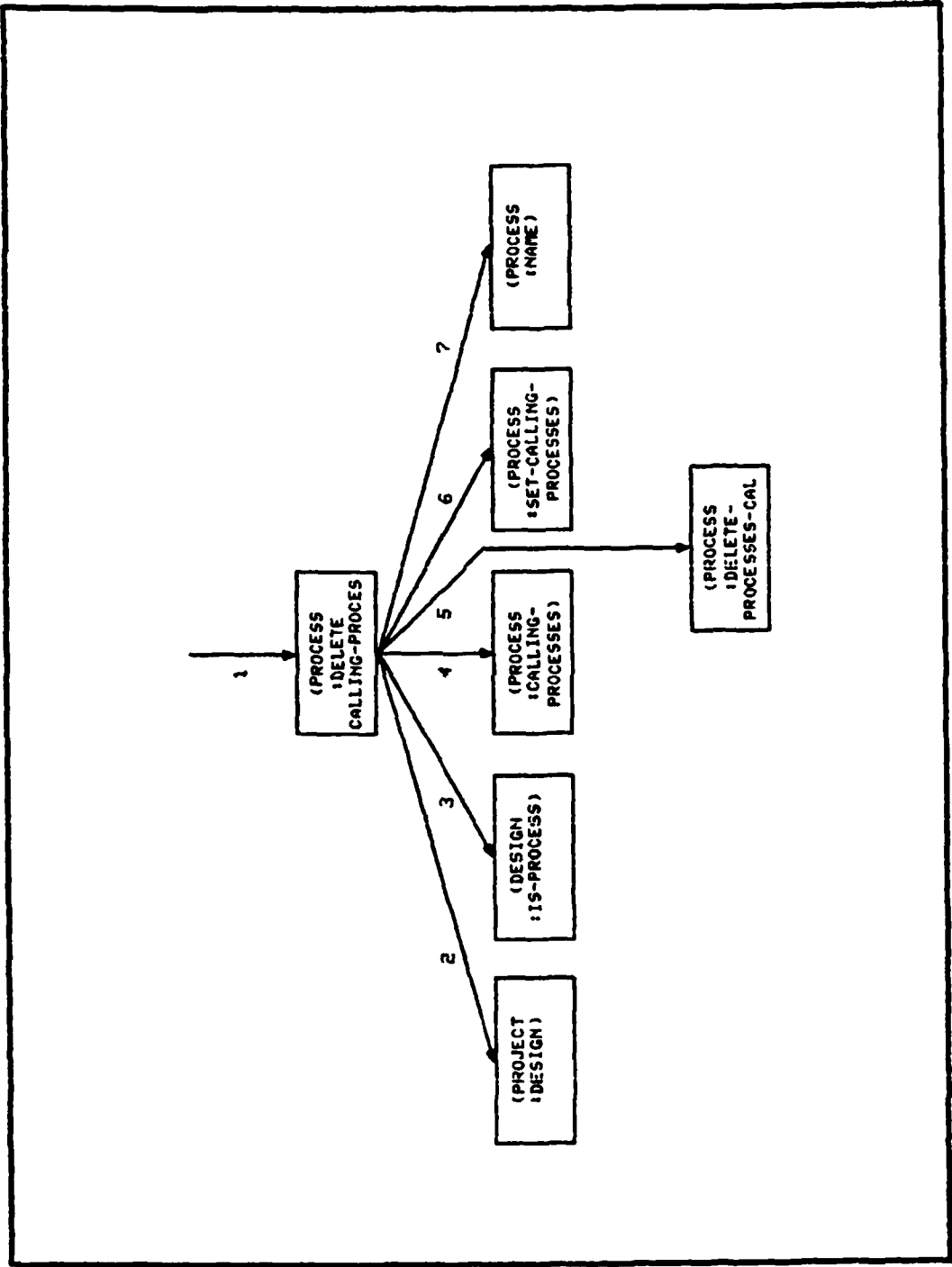
(process :add-calling-processes) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	list of process names	data		
2	(none)		pointer to design instance	data
3	list of process names	data		
4	process name	data	is process?	flag
5	(none)		name of current process	data
6	(none)		list of calling processes	data
7	name of current process	data		
8	process name	data	pointer to new process instance	data



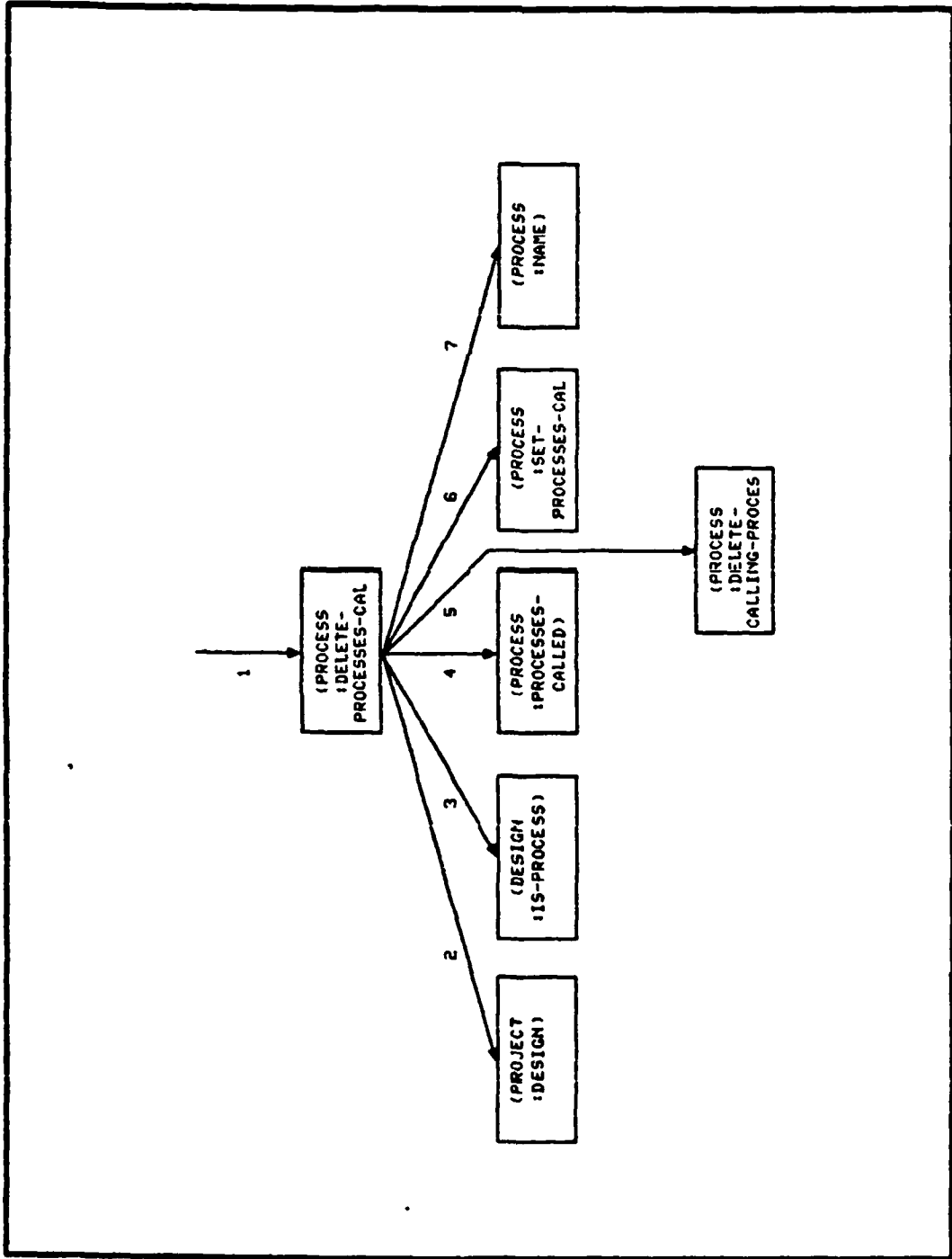
(process :add-processes-called) Interfaces

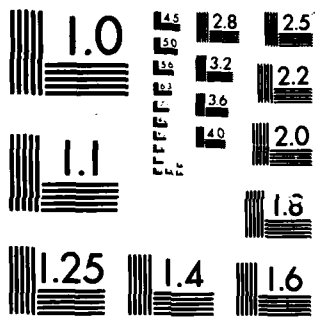
#	Passed Parameters	Type	Returned Parameters	Type
1	list of process names	data		
2	(none)		pointer to design instance	data
3	list of process names	data		
4	process name	data	is process?	flag
5	(none)		name of current process	data
6	(none)		list of calling processes	data
7	name of current process	data		
8	process name	data	pointer to new process instance	data



(process :delete-calling-processes) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	list of process names	data		
2	(none)		pointer to design instance	data
3	process name	data	is process?	flag
4	(none)		list of calling processes	data
5	name of current process	data		
6	list of process names	data		
7	(none)		name of current process	data

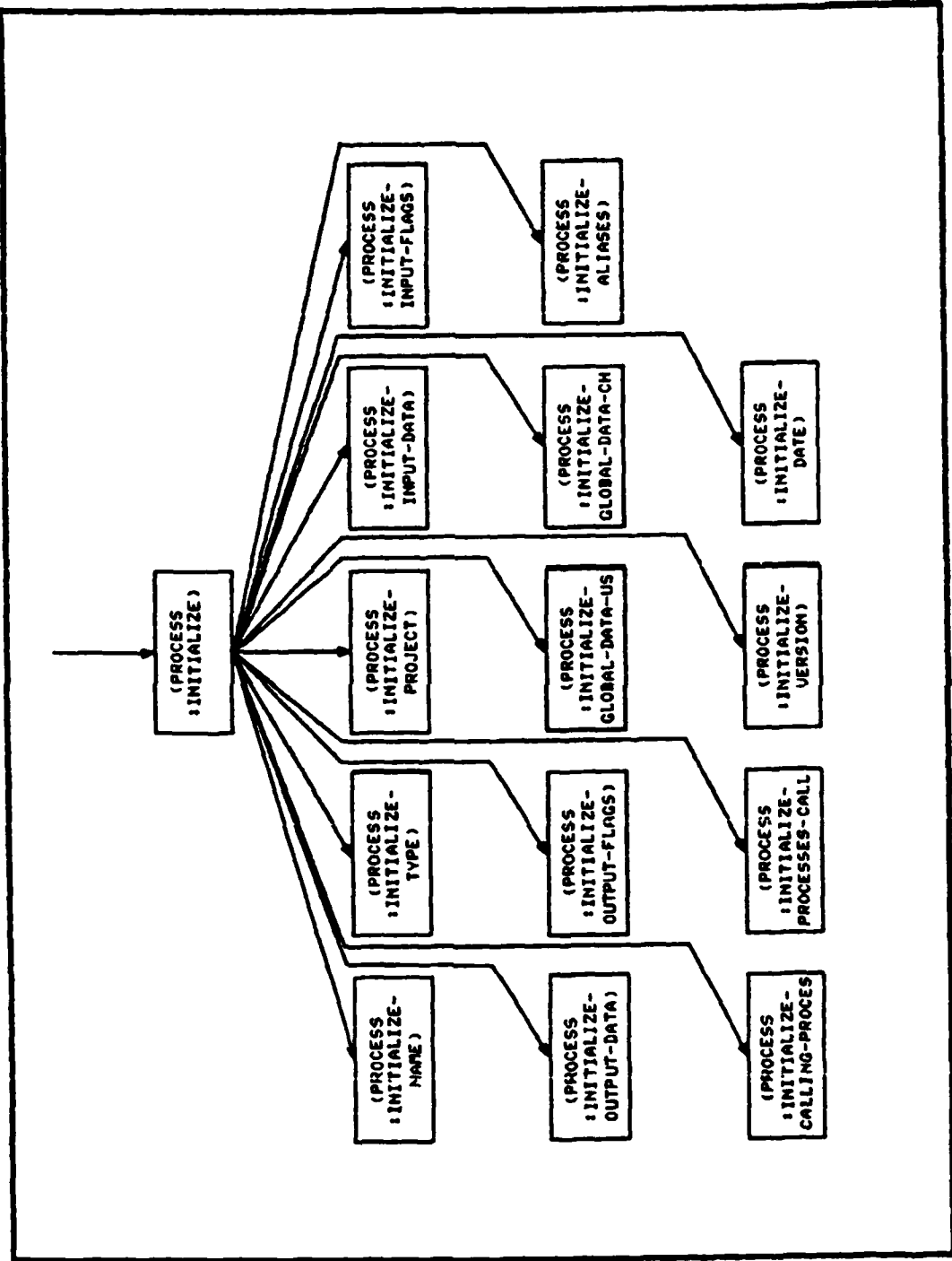


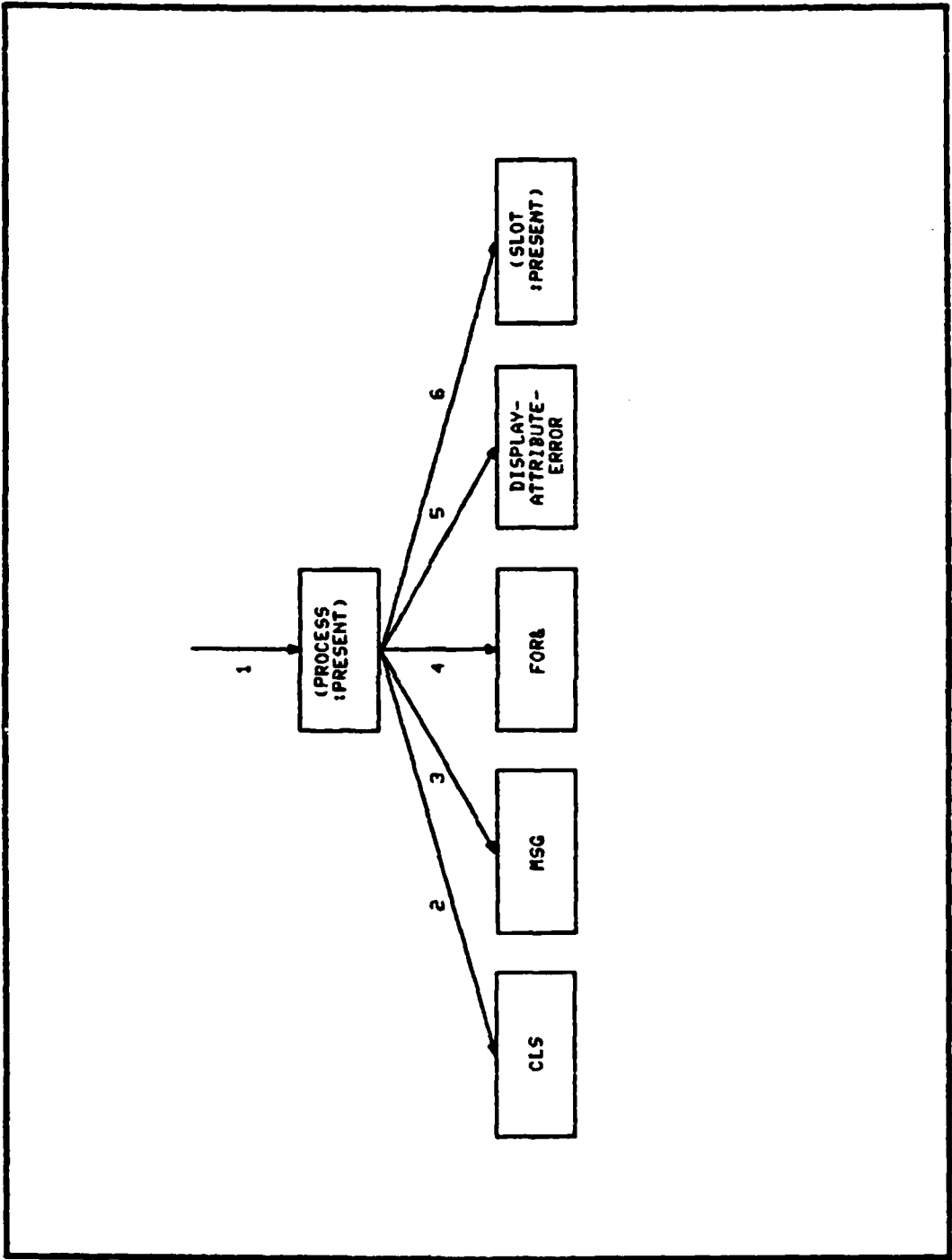


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

(process :delete-processes-called) Interfaces

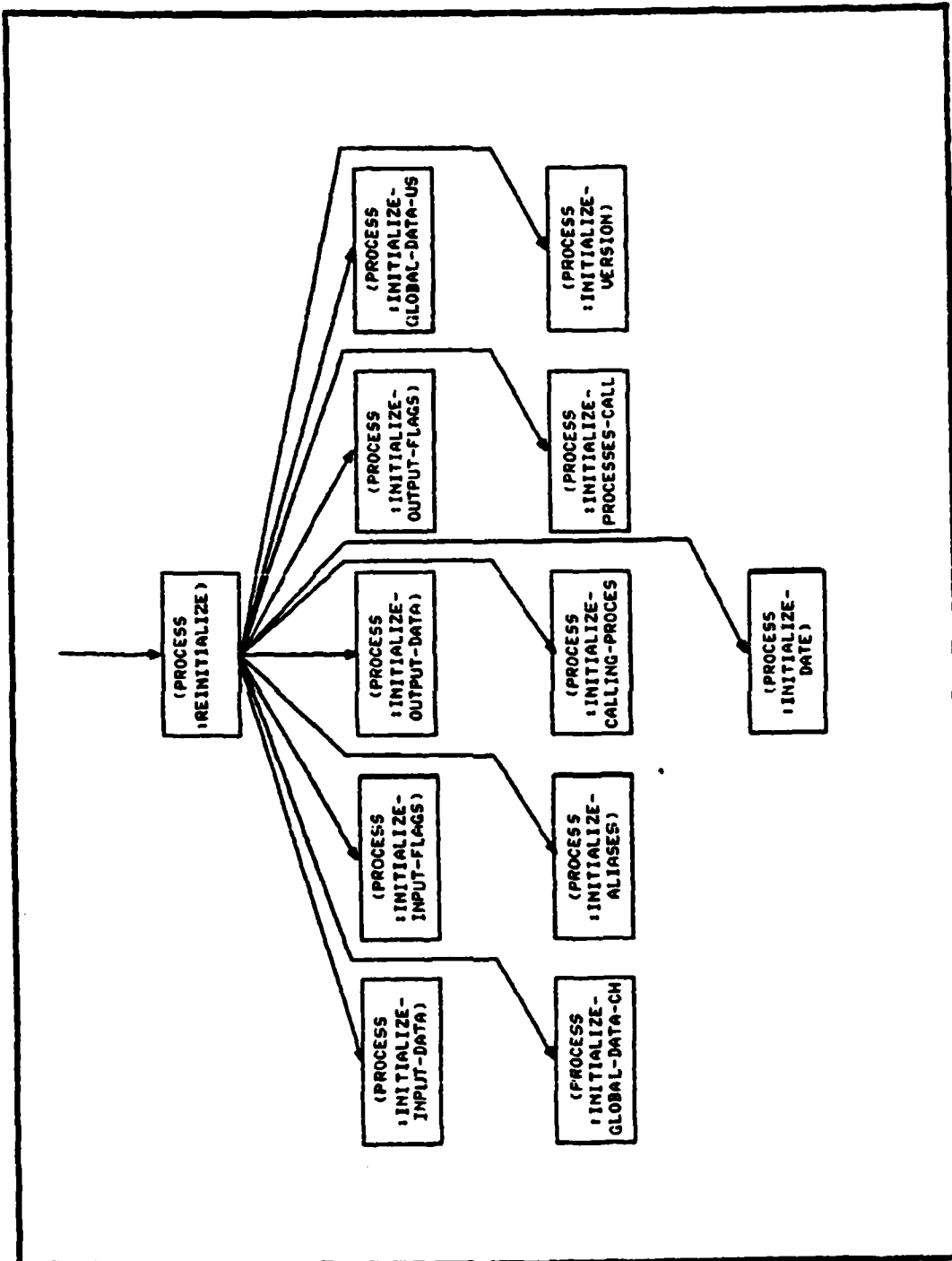
#	Passed Parameters	Type	Returned Parameters	Type
1	list of process names	data		
2	(none)		pointer to design instance	data
3	process name	data	is process?	flag
4	(none)		list of calling processes	data
5	name of current process	data		
6	list of process names	data		
7	(none)		name of current process	data

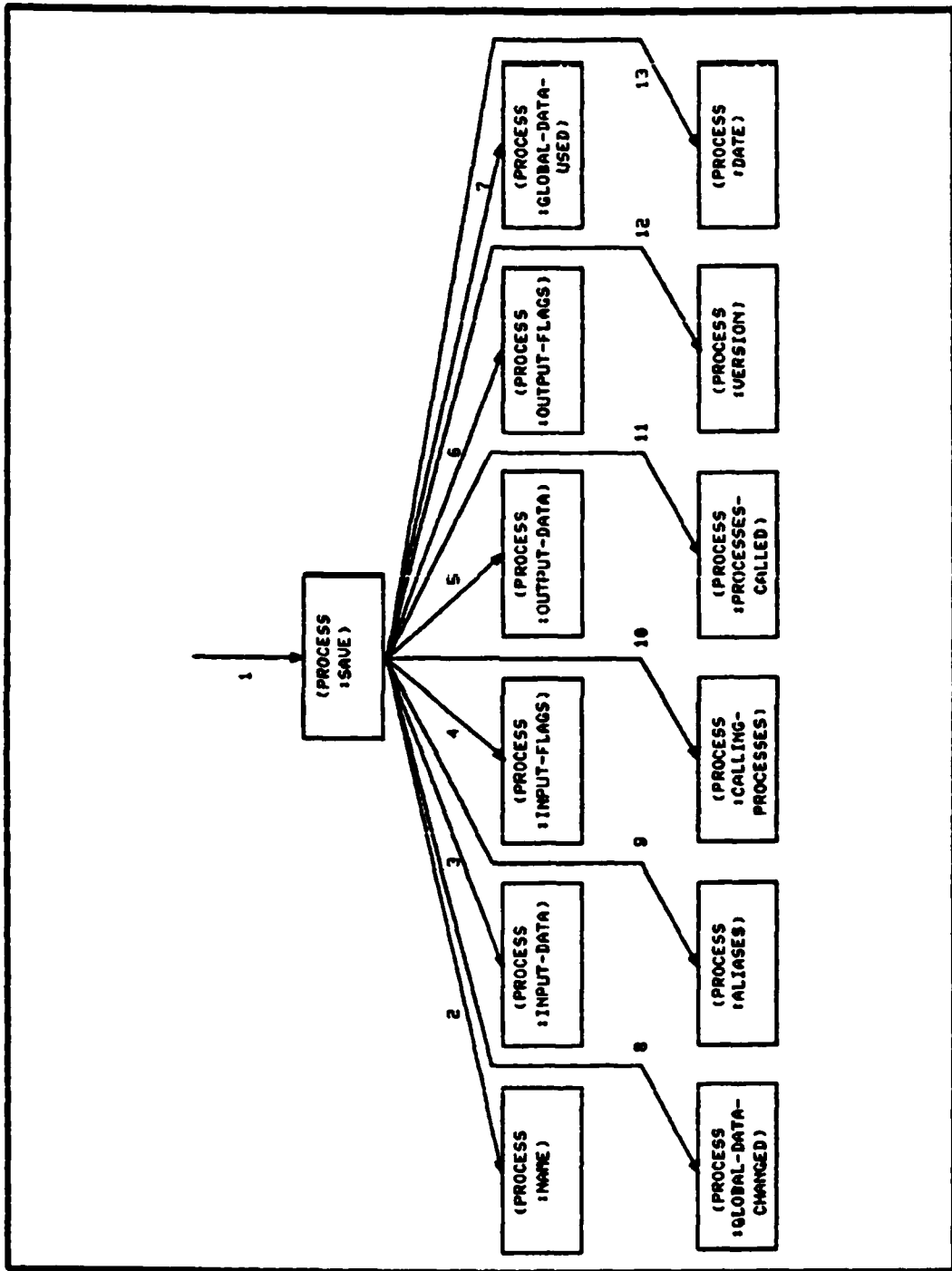




(process :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables	data		
3	(none)			
4	list of instance variables	data		
5	list of invalid instance variables	data		
6	(none)			

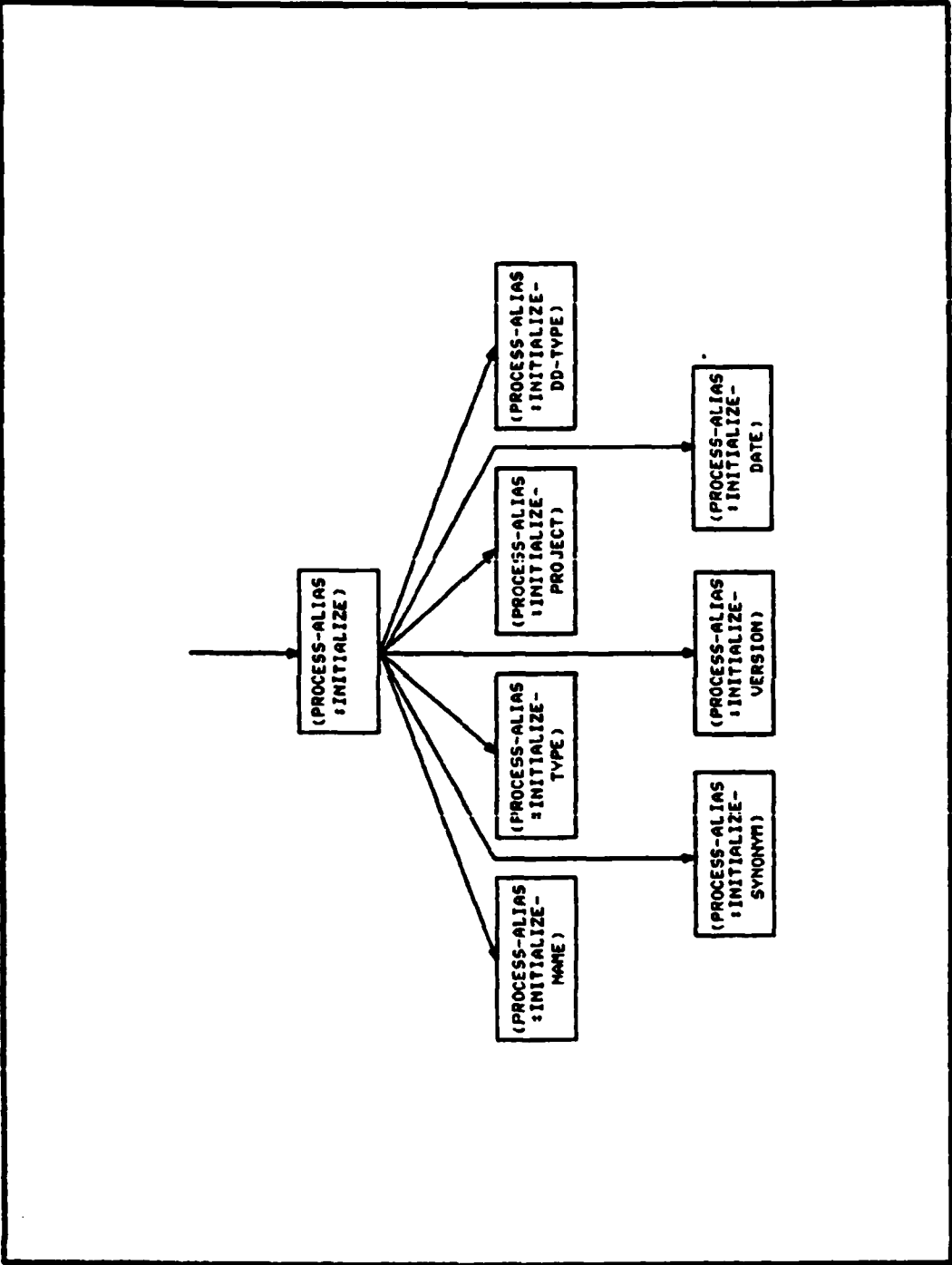


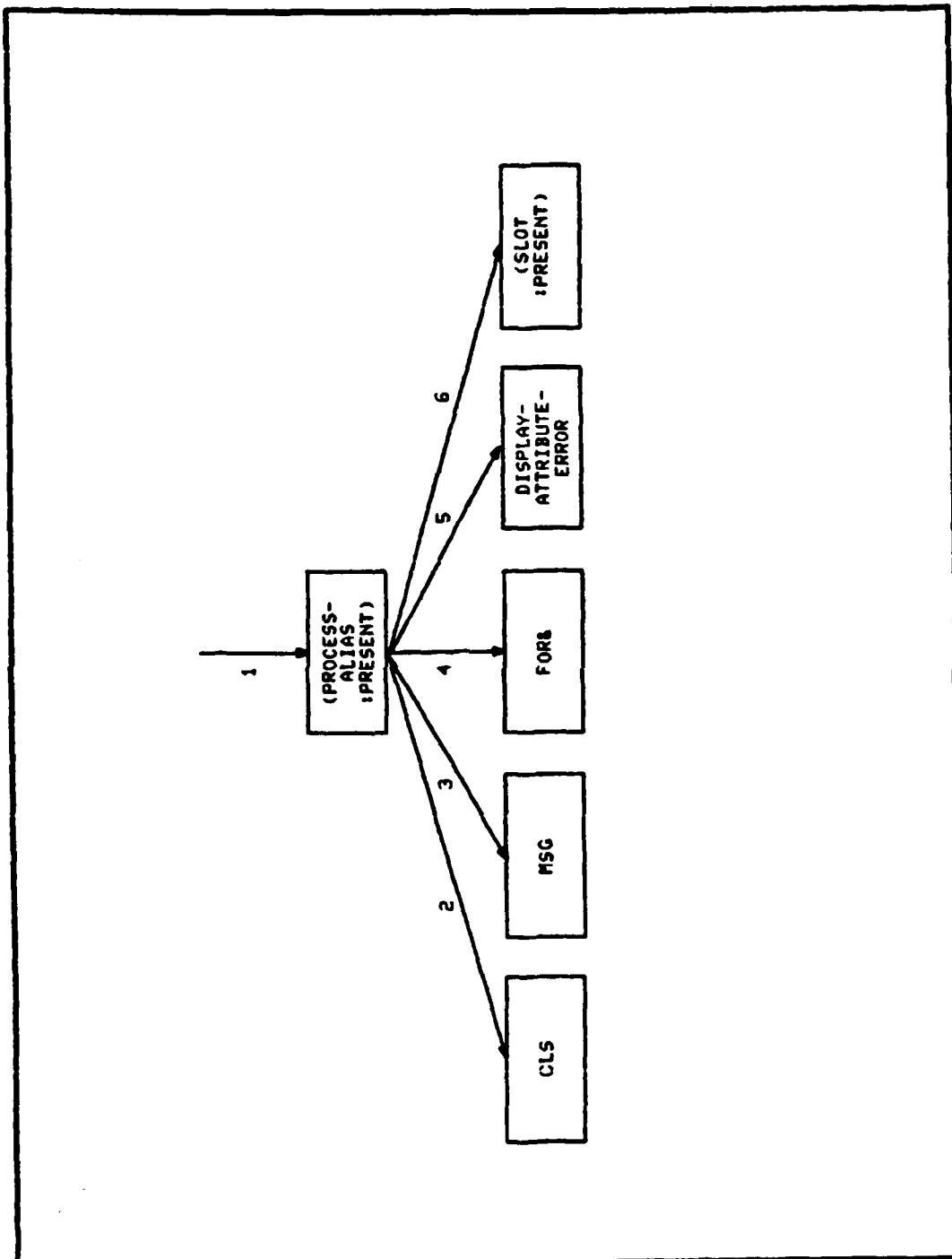


(process :save) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	design instance pointer stream	data data		
2	(none)		name	data
3	(none)		input-data	data
4	(none)		input-flags	data
5	(none)		output-data	data
6	(none)		output-flags	data
7	(none)		global-data-used	data
8	(none)		global-data-changed	data
9	(none)		aliases	data
10	(none)		calling-processes	data
11	(none)		processes-called	data
12	(none)		version	data
13	(none)		date	data

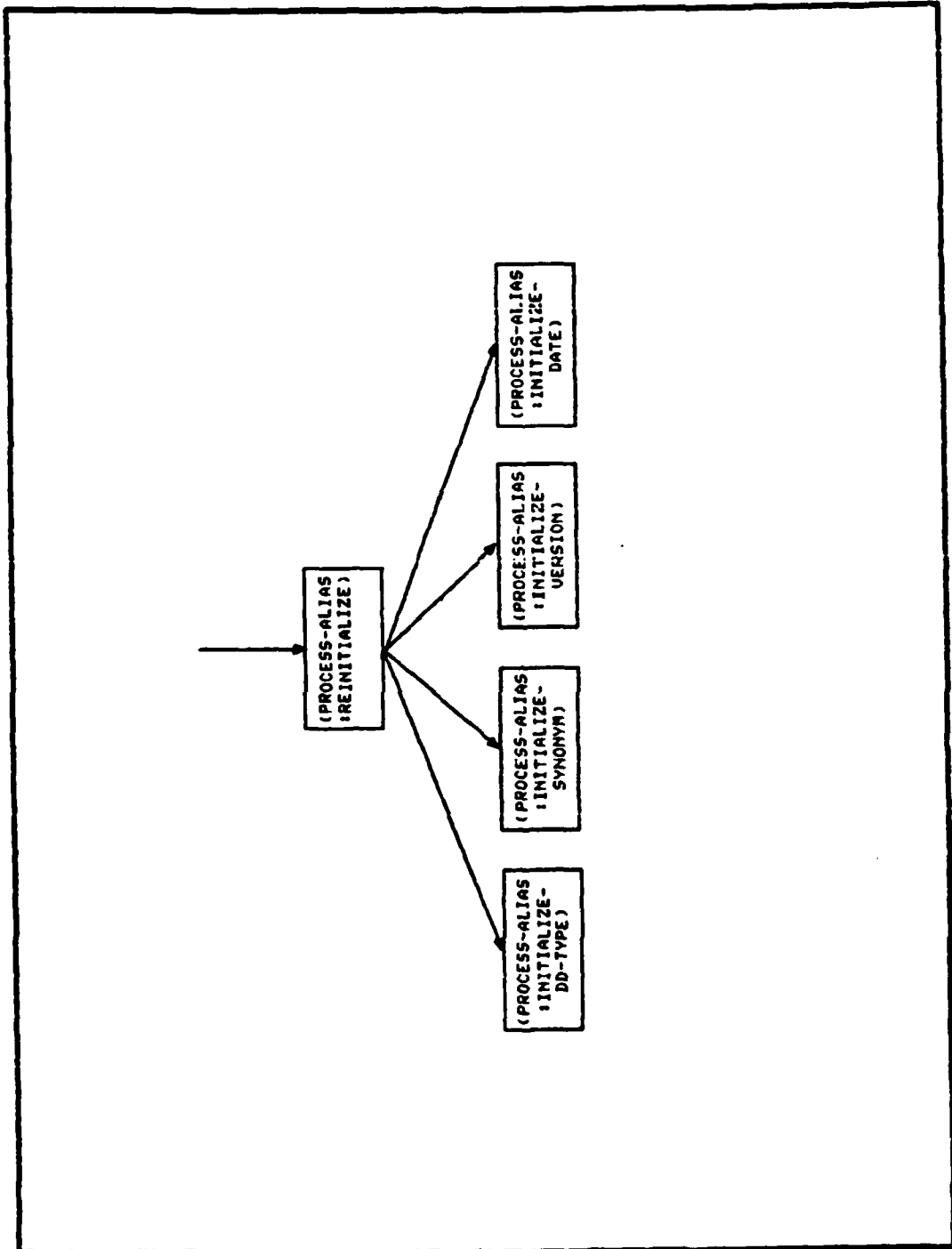
PROCESS-ALIAS Methods

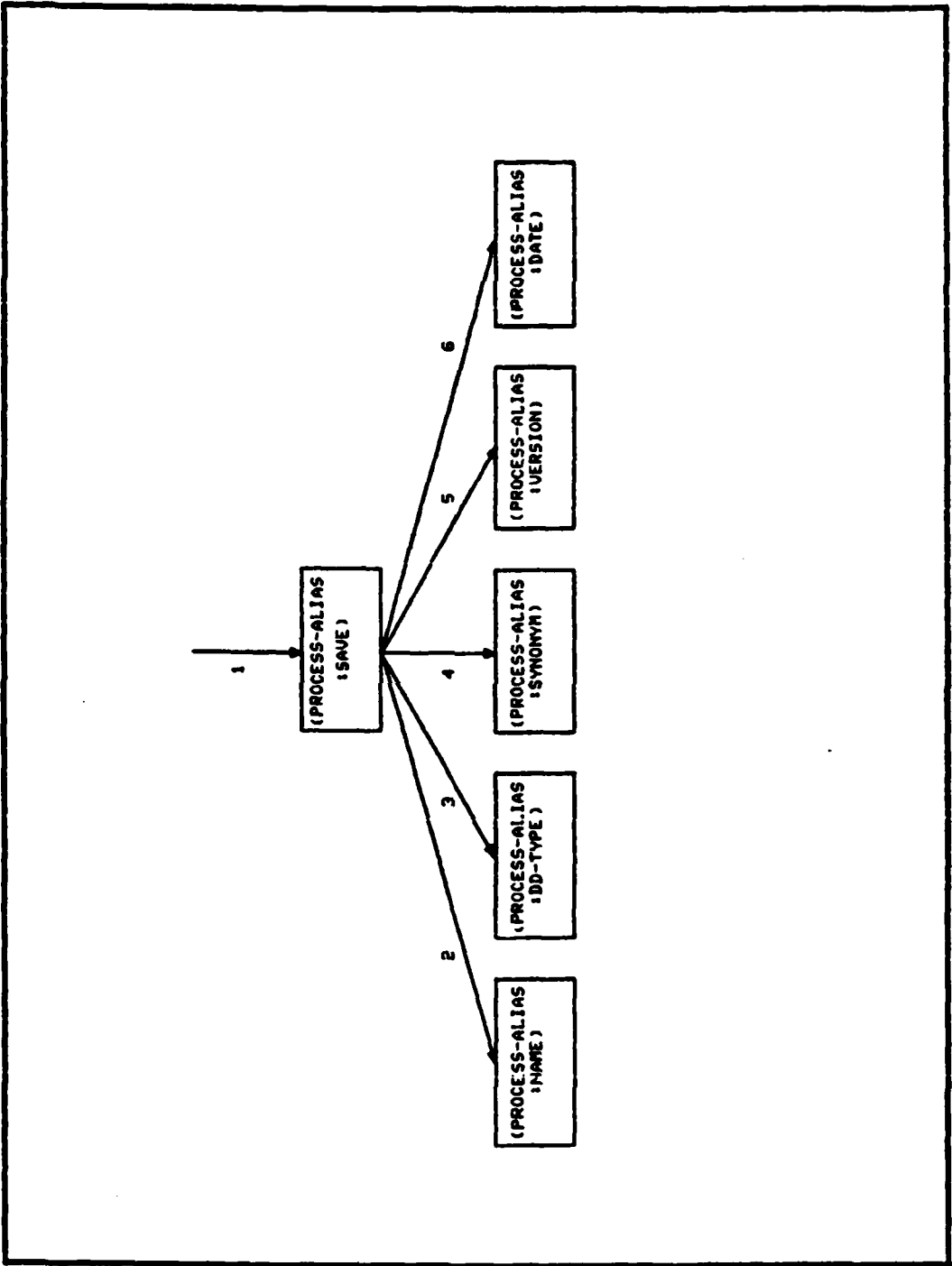




(process-alias :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables	data		
3	(none)			
4	list of instance variables	data		
5	list of invalid instance variables	data		
6	(none)			

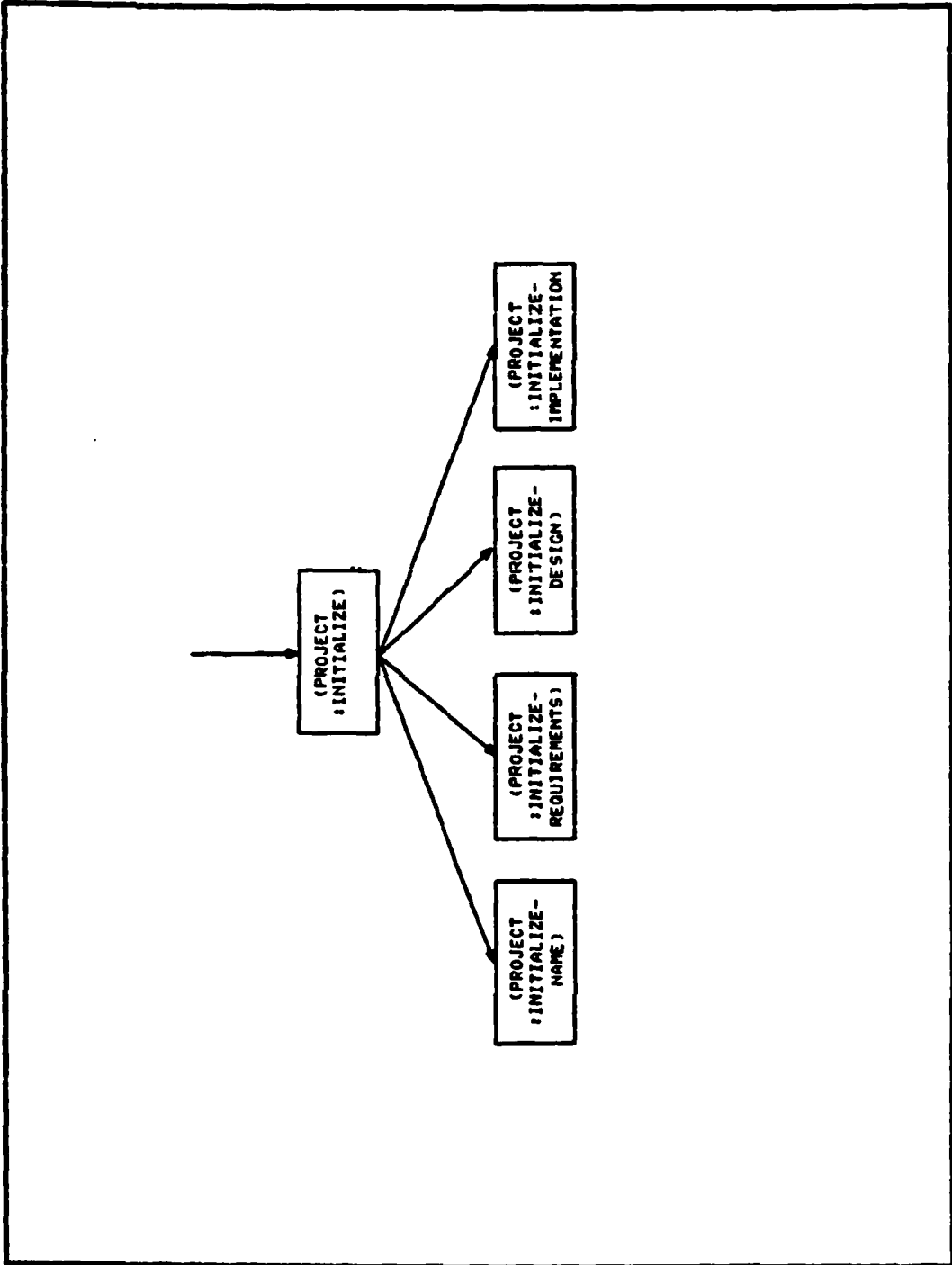


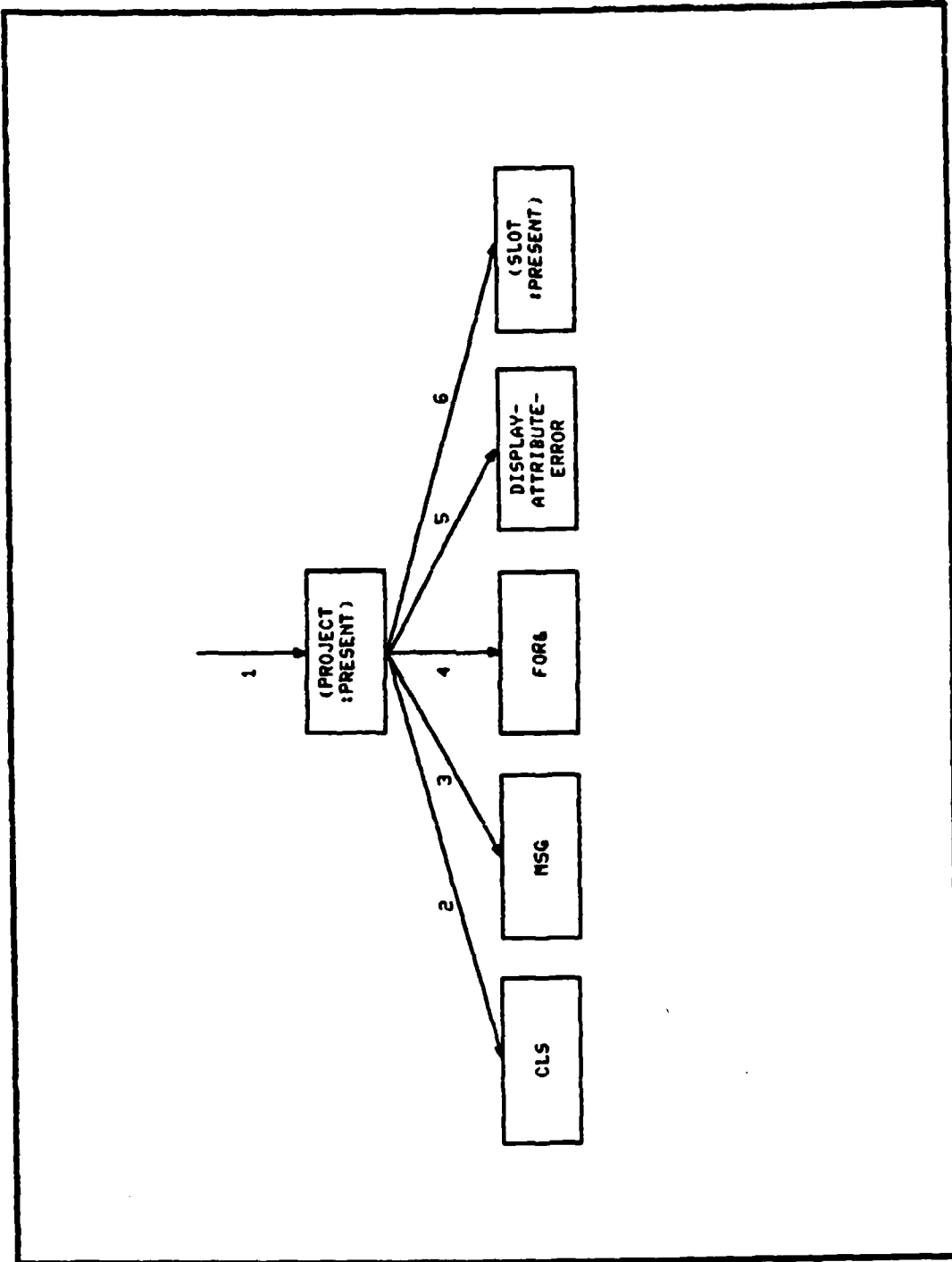


(process-alias :save) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	design instance pointer stream	data data		
2	(none)		name	data
3	(none)		dd-type	data
4	(none)		synonym	data
5	(none)		version	data
6	(none)		date	data

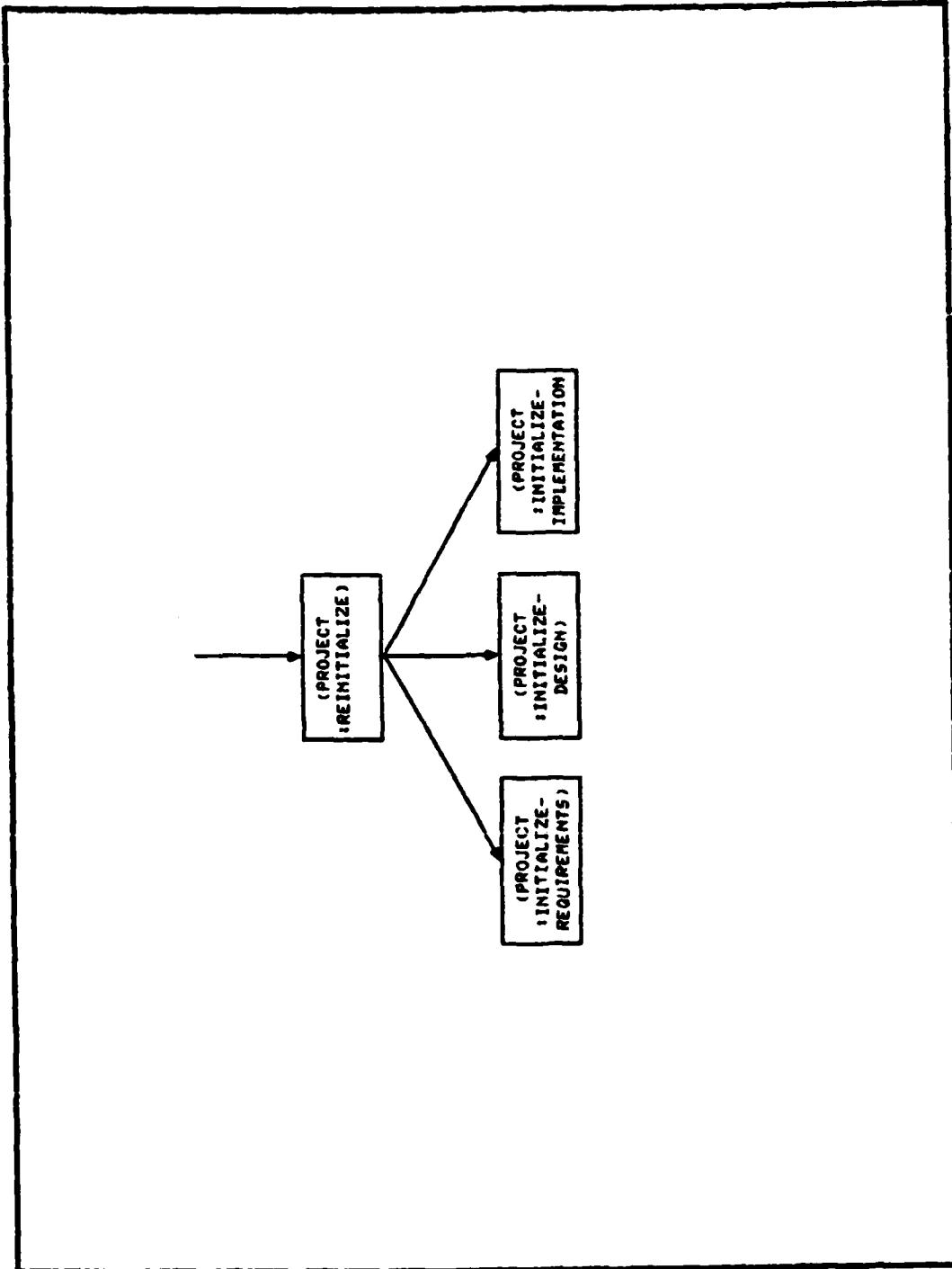
PROJECT Methods

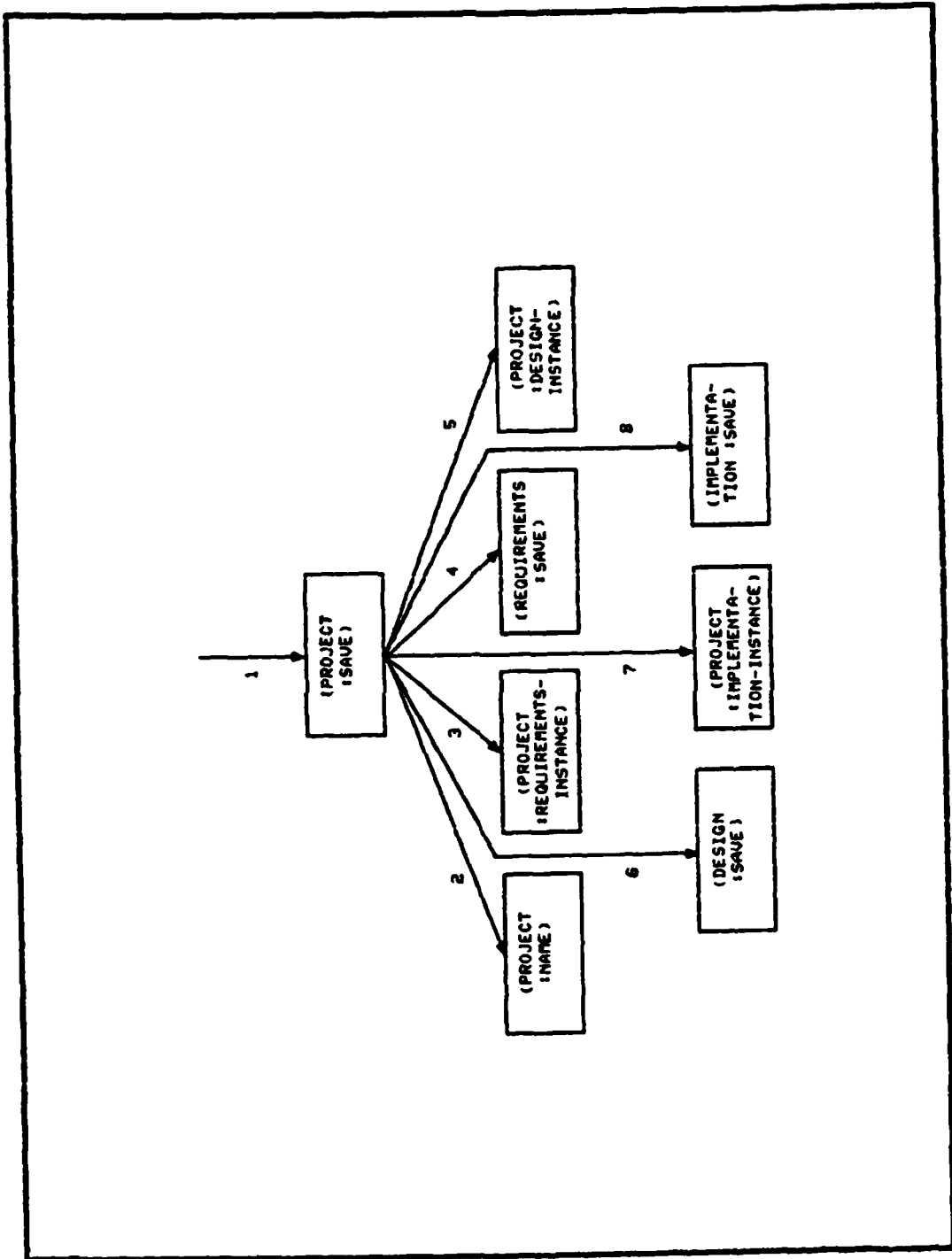




(project :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables	data		
3	(none)			
4	list of instance variables	data		
5	list of invalid instance variables	data		
6	(none)			

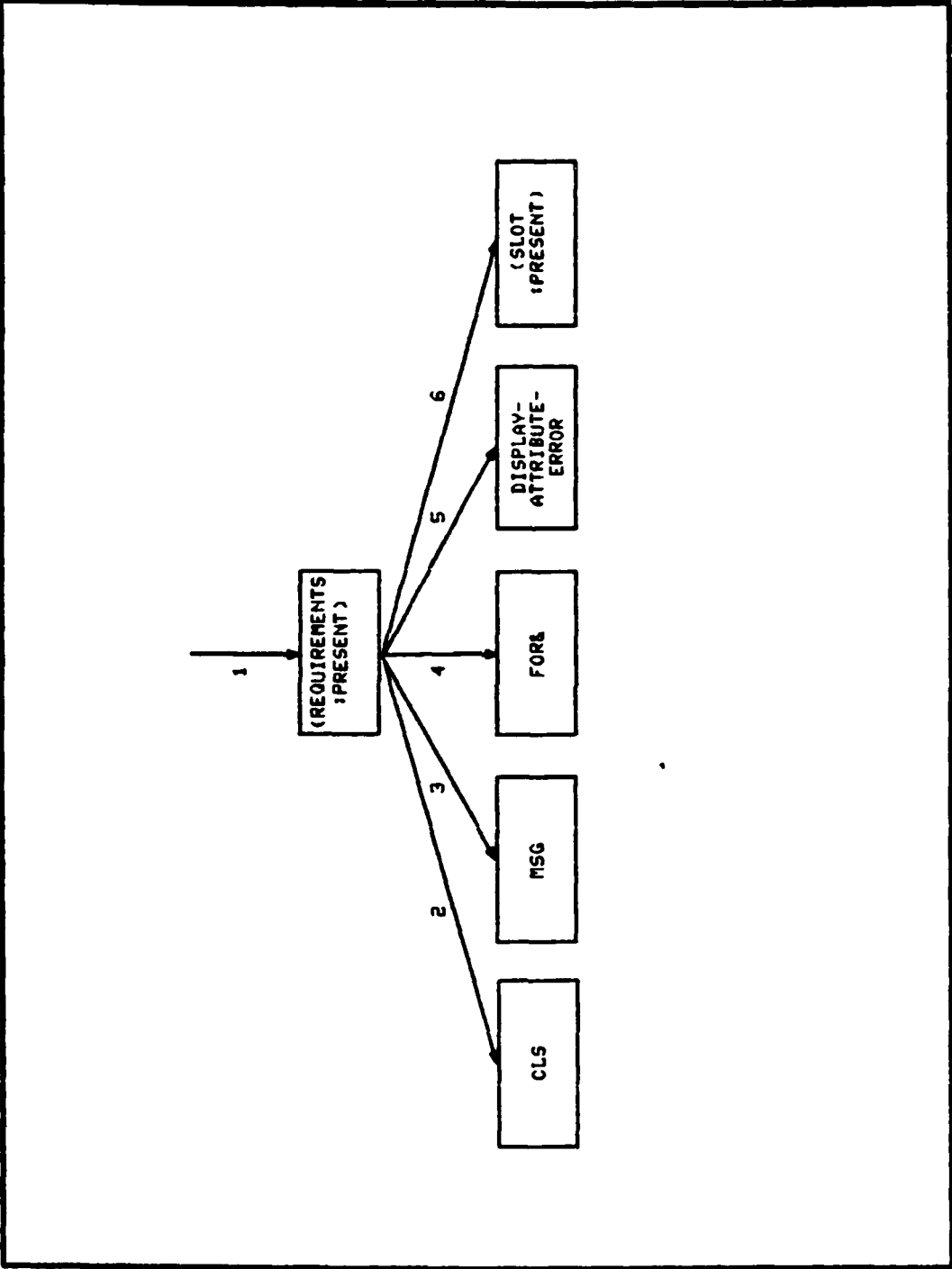




(project :save) Interfaces

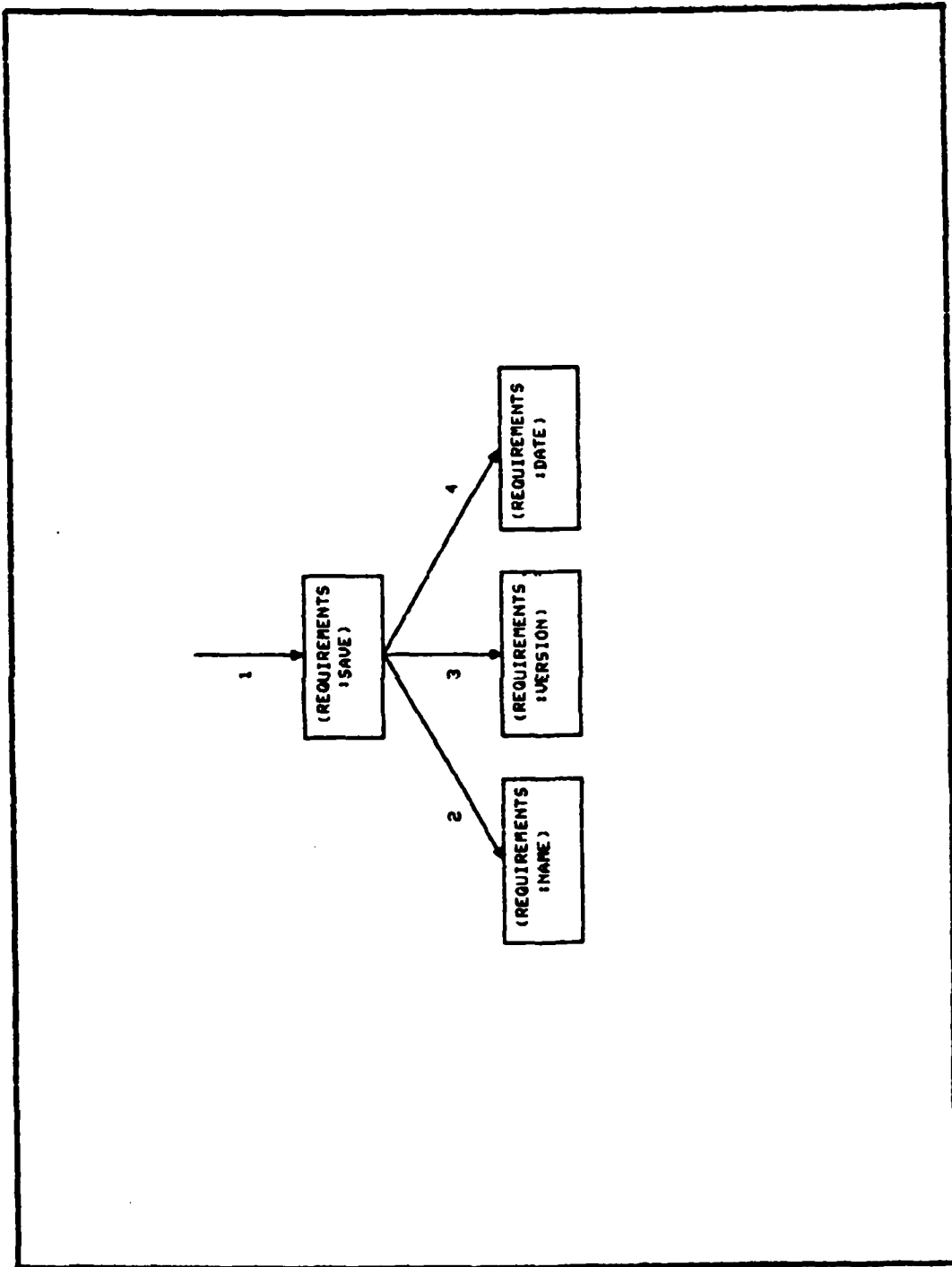
#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	(none)		project name	data
3	(none)		pointer to requirements instance	data
4	pointer to project instance output stream	data data		
5	(none)		pointer to design instance	data
6	pointer to project instance output stream	data data		
7	(none)		pointer to implementation instance	data
8	pointer to project instance output stream	data data		

REQUIREMENTS Methods



(requirements :present) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	(none)			
2	list of instance variables		data	
3	(none)			
4	list of instance variables		data	
5	list of invalid instance variables		data	
6	(none)			



(requirements :save) Interfaces

#	Passed Parameters	Type	Returned Parameters	Type
1	project instance pointer stream	data data		
2	(none)		name	data
3	(none)		version	data
4	(none)		date	data

Appendix C
Grammar Definition

Appendix C
Table of Contents

	Page
Introduction	C- 3
Grammar Definition (BNF)	C- 4
Correspondence Between Instance Variables and Attribute Values	C-10

Introduction

This appendix defines the grammar of the Data Dictionary System (DDS). The first part of this definition specifies the productions of the grammar using a Backus-Naur Form (BNF) notation. The second part defines the correspondence between the instance variables of the various object types and literal values contained in the <attribute> production of the grammar.

Grammar Definition

Notation

::= is defined to be
[] select one
() optional
<> class symbol
\$† comment
¶ or
* user input

<grammar> ::=
 (please) <aux-grammar>

<activity> ::= *
 \$a user-defined member of the object class ACTIVITY†

<activity-alias> ::= *
 \$a user-defined member of the object class
 ACTIVITY-ALIAS†

<activity-alias-instance> ::=
 activity alias <activity-alias>

<activity-instance> ::=
 activity <activity>

<add> ::=
 [add ¶ append ¶ concatenate ¶ include ¶ insert]

<add-command> ::=
 [<add> <literal> to (the) <attribute> [of ¶ by]
 <instance> ¶ <add> <literal> to <instance> <attribute>
 ¶ the <attribute> of <instance> is <literal> ¶
 <instance> <attribute> is <literal> ¶ [
 <module-instance> ¶ <process-instance>] [calls ¶
 invokes] <literal> ¶ [<module-instance> ¶
 <process-instance>] [is called by ¶ is invoked by ¶
 is a submodule of] <literal>]

```

<attribute> ::=
  [ activities ¶ activity number ¶ activity numbers where
    used ¶ activity # ¶ alias ¶ aliases ¶ attributes ¶
    called modules ¶ called processes ¶ calling modules ¶
    calling processes ¶ common data changed ¶ common data
    used ¶ common parameters changed ¶ common parameters
    used ¶ common variables changed ¶ common variables
    used ¶ component data elements ¶ component parameters
    ¶ component variables ¶ components ¶ composition ¶
    control inputs ¶ controls ¶ data dictionary type ¶
    data elements ¶ data type ¶ date ¶ dd type ¶
    description ¶ design ¶ design data ¶ design
    information ¶ destination ¶ destinations ¶ driver ¶
    drivers ¶ entry date ¶ entry version ¶ files input ¶
    files output ¶ files read ¶ files written ¶ input
    files ¶ global data changed ¶ global data used ¶
    global parameters changed ¶ global parameters used ¶
    global variables changed ¶ global variables used ¶
    globals changed ¶ globals used ¶ hardware input ¶
    hardware output ¶ highest-level activity ¶
    implementation ¶ implementation data ¶ implementation
    information ¶ input data ¶ input files ¶ input flags ¶
    inputs ¶ invoked modules ¶ invoked processes ¶
    invoking modules ¶ invoking processes ¶ instance
    variables ¶ legal range ¶ legal values ¶ list of
    activities ¶ list of aliases ¶ list of called modules
    ¶ list of called processes ¶ list of calling modules ¶
    list of calling processes ¶ list of component data
    elements ¶ list of component parameters ¶ list of
    component variables ¶ list of components ¶ list of
    controls ¶ list of data elements ¶ list of input data
    ¶ list of input flags ¶ list of inputs ¶ list of
    invoked modules ¶ list of invoked processes ¶ list of
    invoking modules ¶ list of invoking processes ¶ list
    of legal values ¶ list of mechanisms ¶ list of modules
    ¶ list of modules called ¶ list of modules invoked ¶
    list of output data ¶ list of output flags ¶ list of
    outputs ¶ list of parameter aliases ¶ list of
    parameters ¶ list of process aliases ¶ list of
    processes ¶ list of processes called ¶ list of
    processes invoked ¶ list of related items ¶ list of
    related parameters ¶ list of related requirements ¶
    list of submodules ¶ list of subprocesses ¶ list of
    values ¶ list of variables ¶ main activity ¶ main
    module ¶ main modules ¶ main process ¶ main processes
    ¶ main program ¶ main programs ¶ max legal value ¶ max
    value ¶ maximum legal value ¶ maximum value ¶
    mechanisms ¶ min legal value ¶ min value ¶ minimum
    legal value ¶ minimum value ¶ module number ¶ module #
  ]

```

¶ modules ¶ name ¶ node ¶ node number ¶ node # ¶
 number ¶ output data ¶ output files ¶ output flags ¶
 outputs ¶ parameter aliases ¶ parameters ¶ parent
 activity ¶ parent data element ¶ parent parameter ¶
 parent variable ¶ part of ¶ passed from ¶ passed to ¶
 process aliases ¶ process number ¶ process # ¶
 processes ¶ processes called ¶ processes invoked ¶
 project ¶ project name ¶ range ¶ related items ¶
 related parameters ¶ related requirement ¶ related
 requirements ¶ related requirement number ¶ related
 requirements numbers ¶ requirement number ¶
 requirement # ¶ requirements ¶ requirements data ¶
 requirements information ¶ SADT activity numbers where
 used ¶ slots ¶ source ¶ sources ¶ storage type ¶
 submodules ¶ subprocesses ¶ synonym ¶ synonyms ¶ title
 ¶ top-level activity ¶ top-level program ¶ top-level
 programs ¶ type ¶ values ¶ variables ¶ version ¶ #]

```

<attribute-list> ::=
  <attribute> ( [ and <attribute> ¶ <attribute-list> ] )

<aux-grammar> ::=
  [ <command> ¶ <meta-command> ]

<cls> ::=
  [ cls ¶ clear screen ¶ clear the screen ]

<command> ::=
  [ <add-command> ¶ <create-command> ¶ <delete-command> ¶
    <destroy-command> ¶ <initialize-command> ¶
    <modify-command> ¶ <present-command> ]

<create> ::=
  [ create ¶ make ]

<create-command> ::=
  [ <create> ( a ) ( new ) <object-class> ( called )
    <literal> ¶ <literal> is ( a ) ( new ) <object-class>
  ]

<data-element> ::= *
  $a user-defined member of the object class
  DATA-ELEMENT†

<data-element-alias> ::= *
  $a user-defined member of the object class
  DATA-ELEMENT-ALIAS†
  
```

```

<data-element-alias-instance> ::=
    data element alias <data-element-alias>

<data-element-instance> ::=
    data element <data-element>

<delete> ::=
    [ delete ¶ purge ¶ remove ]

<delete-command> ::=
    [ <delete> <literal> ( from ) <attribute> of <instance>
      ¶ <delete> <literal> ( from ) <instance> <attribute> ]

<design> ::= *
    $a user-defined member of the object class DESIGN†

<design-instance> ::=
    design <design>

<destroy> ::=
    destroy

<destroy-command> ::=
    <destroy> <instance>

<exit> ::=
    [ all pow ¶ exit ¶ quit ]

<help> ::=
    [ h ¶ help ¶ ? ]

<implementation> ::= *
    $a user-defined member of the object class
    IMPLEMENTATION†

<implementation-instance> ::=
    implementation <implementation>

<initialize> ::=
    [ init ¶ initialize ¶ reinit ¶ reinitialize ¶ reset ]

<initialize-command> ::=
    <initialize> <instance>

```

```

<instance> ::=
  [ <activity-instance> ¶ <activity-alias-instance> ¶
    <data-element-instance> ¶
    <data-element-alias-instance> ¶ <design-instance> ¶
    <implementation-instance> ¶ <module-instance> ¶
    <parameter-instance> ¶ <parameter-alias-instance> ¶
    <process-instance> ¶ <process-alias-instance> ¶
    <project-instance> ¶ <requirements-instance> ¶
    <variable-instance> ]

<literal> ::= *
  $any string of characters delimited by blank$

<meta-command> ::=
  [ <cls> ¶ <exit> ¶ <help> ¶ <save> ]

<modify> ::=
  [ alter ¶ change ¶ mod ¶ modify ¶ replace ]

<modify-command> ::=
  [ <modify> <attribute> of <instance> to <literal> ¶
    <modify> <instance> <attribute> to <literal> ]

<module> ::= *
  $a user-defined member of the object class MODULE$

<module-instance> ::=
  module <module>

<parameter> ::= *
  $a user-defined member of the object class PARAMETER$

<parameter-alias> ::= *
  $a user-defined member of the object class
  PARAMETER-ALIAS$

<parameter-alias-instance> ::=
  parameter alias <parameter-alias>

<parameter-instance> ::=
  parameter <parameter>

<present> ::=
  [ display ¶ present ¶ print ¶ show ( me ) ¶ tell ( me )
    ¶ type ¶ what [ is ¶ are ] ¶ who is ]

```



```

<present-command> ::=
  [ <present> the <attribute-list> [ of ¶ by ] <instance>
    ¶ <present> <instance> ( <attribute-list> ) ]

<process> ::= *
  $a user-defined member of the object class PROCESST

<process-alias> ::= *
  $a user-defined member of the object class
  PROCESS-ALIAS†

<process-alias-instance> ::=
  process alias <process-alias>

<process-instance> ::=
  process <process>

<project> ::= *
  $a user-defined member of the object class PROJECT†

<project-instance> ::=
  [ project <project> ¶ *project* ]

<requirements> ::= *
  $a user-defined member of the object class
  REQUIREMENTS†

<requirements-instance> ::=
  requirements <requirements>

<save> ::=
  [ keep ¶ save ¶ store ]

<variable> ::= *
  $a user-defined member of the object class VARIABLE†

<variable-instance> ::=
  variable <variable>

```

Correspondence Between Instance Variables
and Attribute Values

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
activities	activities list of activities	
aliases	alias aliases list of aliases	Y
calling-processes	calling modules calling processes invoking modules invoking processes list of calling modules list of calling processes list of invoking modules list of invoking processes	Y Y
composition	component data elements component parameters component variables components composition list of component data elements list of componen parameters list of component variables list of components	
controls	control inputs controls list of controls	
data-elements	data elements list of data elements	

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
data-type	data type	
date	date	Y
	entry date	Y
dd-type	data dictionary type dd type	Y
description	description	
design	design	Y
	design data	
	design information	
files-read	files input files read input files	
files-written	files output files written output files	
global-data-changed	common data changed	
	common parameters changed	
	common variables changed	
	global data changed	Y
	global parameters changed	
	global variables changed	
	globals changed	Y
global-data-used	common data used	
	common parameters used	
	common variables used	
	global data used	Y
	global parameters used	
	global variables used	
	globals used	Y

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
hardware-input	hardware input hardware read	
hardware-output	hardware output hardware write	
implementation	implementation implementation data implementation information	Y
input-data	input data list of input data	Y
input-flags	input flags list of input flags	Y
inputs	inputs list of inputs	
main-program	driver drivers main module main modules main process main processes main program main programs top-level program top-level programs	Y Y Y Y
max-value	max legal value max value maximum legal value maximum value	
mechanisms	list of mechanisms mechanisms	

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
min-value	min legal value min value minimum legal value minimum value	
modules	list of modules modules	
name	name title	Y
number	activity number activity # node node number node # number #	
output-data	list of output data output data	Y
output-flags	list of output flags output flags	Y
outputs	list of outputs outputs	
parameter-aliases	list of parameter aliases parameter aliases	Y
parameters	list of parameters parameters	Y
parent-activity	parent activity	

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
part-of	parent data element	
	parent parameter	
	parent variable	
	part of	
passed-from	passed from	Y
	source	Y
	sources	Y
passed-to	destination	Y
	destinations	Y
	passed to	Y
process-aliases	list of process aliases	
	process aliases	Y
processes	list of processes	
	processes	Y
processes-called	called modules	
	called processes	Y
	invoked modules	
	invoked processes	Y
	list of called modules	
	list of called processes	
	list of invoked modules	
	list of invoked processes	
	list of modules called	
	list of modules invoked	
	list of processes called	
	list of processes invoked	
	list of submodules	
	list of subprocesses	
	modules called	
modules invoked		
processes called	Y	
processes invoked	Y	
submodules		
subprocesses	Y	

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
project	project	Y
	project name	Y
range	legal range	
	range	
related-items	list of related items	
	list of related parameters	
	list of related requirements	
	related items	
	related parameters	
	related requirements	
	related requirements number	
	related requirements numbers	
	related requirements #	
	related SADT activity	
	related SADT data items	
	requirement number	
	requirement #	
requirements	requirements	Y
	requirements data	
	requirements information	
slots	attributes	
	instance variables	
	slots	
storage-type	storage type	
synonym	synonym	Y
	synonyms	Y
top-level-activity	highest-level activity	
	main activity	
	top-level activity	

<u>Instance Variable</u>	<u>Attribute Value</u>	<u>Implemented?</u>
type	type	Y
values	legal values list of legal values list of values values	
variables	list of variables variables	
version	entry version version	Y Y
where-used	activity numbers where used SADT activity numbers where used where used	

Appendix D
Implemented Grammar

Appendix D
Table of Contents

	Page
Introduction	D- 3
Implemented Grammar Listing	D- 4

Introduction

This appendix contains a listing of the Data Dictionary System's (DDS) implemented grammar. It is presented in the same Lisp-readable format as the actual grammar disk file.

```

(DECLARE (SPECIAL *META-SYMBOLS*))
(SETQ *META-SYMBOLS* NIL)

(DECLARE (SPECIAL <GRAMMAR>))
(CREATE-PRODUCTION <GRAMMAR>)
(SETQ
  <GRAMMAR> '(<GRAMMAR> ((PLEASE ((<AUX-GRAMMAR> (($ NIL
NIL)) NIL)) NIL)
              (<AUX-GRAMMAR> (($ NIL NIL)) NIL))
              (SEND *NEW-EVENT* :EXECUTE)))

(DECLARE (SPECIAL <AUX-GRAMMAR>))
(CREATE-PRODUCTION <AUX-GRAMMAR>)
(SETQ <AUX-GRAMMAR>
      '(<AUX-GRAMMAR>
        ((<COMMAND> ((* NIL (SEND *NEW-EVENT* :SET-ACTION-TYPE
NIL))) NIL)
         (<META-COMMAND>
          ((* NIL (SEND *NEW-EVENT* :SET-ACTION-TYPE 'META)))
          NIL))
        NIL))

(DECLARE (SPECIAL <COMMAND>))
(CREATE-PRODUCTION <COMMAND>)
(SETQ <COMMAND> '(<COMMAND> ((<PRESENT-COMMAND> ((* NIL
NIL)) NIL)
                          (<INITIALIZE-COMMAND> ((* NIL NIL)
NIL))
                          NIL))

(DECLARE (SPECIAL <ADD-COMMAND>))
(CREATE-PRODUCTION <ADD-COMMAND>)
(SETQ
  <ADD-COMMAND> '(<ADD-COMMAND> NIL (SEND *NEW-EVENT*
:SET-ACTION ':ADD)))

(DECLARE (SPECIAL <CREATE-COMMAND>))
(CREATE-PRODUCTION <CREATE-COMMAND>)
(SETQ <CREATE-COMMAND>
      '(<CREATE-COMMAND> NIL (SEND *NEW-EVENT* :SET-ACTION
':CREATE)))

(DECLARE (SPECIAL <DELETE-COMMAND>))
(CREATE-PRODUCTION <DELETE-COMMAND>)
(SETQ <DELETE-COMMAND>
      '(<DELETE-COMMAND> NIL (SEND *NEW-EVENT* :SET-ACTION
':DELETE)))

```

```

(DECLARE (SPECIAL <DESTROY-COMMAND>))
(CREATE-PRODUCTION <DESTROY-COMMAND>)
(SETQ <DESTROY-COMMAND>
  '(<DESTROY-COMMAND> NIL (SEND *NEW-EVENT* :SET-ACTION
':DESTROY)))

```

```

(DECLARE (SPECIAL <INITIALIZE-COMMAND>))
(CREATE-PRODUCTION <INITIALIZE-COMMAND>)
(SETQ <INITIALIZE-COMMAND>
  '(<INITIALIZE-COMMAND>
  ((<INITIALIZE> ((<INSTANCE> ((* NIL NIL)) NIL)) NIL))
  (SEND *NEW-EVENT* :SET-ACTION ':REINITIALIZE)))

```

```

(DECLARE (SPECIAL <MODIFY-COMMAND>))
(CREATE-PRODUCTION <MODIFY-COMMAND>)
(SETQ <MODIFY-COMMAND>
  '(<MODIFY-COMMAND> NIL (SEND *NEW-EVENT* :SET-ACTION
':MODIFY)))

```

```

(DECLARE (SPECIAL <PRESENT-COMMAND>))
(CREATE-PRODUCTION <PRESENT-COMMAND>)
(SETQ
  <PRESENT-COMMAND>
  '(<PRESENT-COMMAND>
    ((<PRESENT>
      ((<INSTANCE> ((* NIL NIL) (<ATTRIBUTE-LIST> ((* NIL
NIL)) NIL)) NIL)
      (THE ((<ATTRIBUTE-LIST> ((BY ((<INSTANCE> NIL NIL))
NIL)
          (OF ((<INSTANCE> ((* NIL NIL)) NIL))
NIL))
          NIL))
      NIL))
    (SEND *NEW-EVENT* :SET-ACTION ':PRESENT)))

```

```

(DECLARE (SPECIAL <ADD>))
(CREATE-PRODUCTION <ADD>)
(SETQ
  <ADD> '(<ADD> ((ADD ((* NIL NIL)) NIL) (APPEND ((* NIL
NIL)) NIL)
  (CONCATENATE ((* NIL NIL)) NIL)
  (INCLUDE ((* NIL NIL)) NIL) (INSERT ((* NIL NIL))
NIL))
  NIL))

```

```
(DECLARE (SPECIAL <CREATE>))
(CREATE-PRODUCTION <CREATE>)
(SETQ
  <CREATE>
  '(<CREATE> ((CREATE ((* NIL NIL)) NIL) (MAKE ((* NIL NIL))
NIL)) NIL))
```

```
(DECLARE (SPECIAL <DELETE>))
(CREATE-PRODUCTION <DELETE>)
(SETQ
  <DELETE> '(<DELETE> ((DELETE ((* NIL NIL)) NIL) (PURGE ((*
NIL NIL)) NIL)
            (REMOVE ((* NIL NIL)) NIL))
  NIL))
```

```
(DECLARE (SPECIAL <DESTROY>))
(CREATE-PRODUCTION <DESTROY>)
(SETQ <DESTROY> '(<DESTROY> ((DESTROY ((* NIL NIL)) NIL))
NIL))
```

```
(DECLARE (SPECIAL <INITIALIZE>))
(CREATE-PRODUCTION <INITIALIZE>)
(SETQ <INITIALIZE>
  '(<INITIALIZE>
    ((REINIT ((* NIL NIL)) NIL)
     (REINITIALIZE ((* NIL NIL)) NIL) (RESET ((* NIL NIL))
NIL)
    (INIT ((* NIL NIL)) NIL) (INITIALIZE ((* NIL NIL))
NIL))
  NIL))
```

```
(DECLARE (SPECIAL <MODIFY>))
(CREATE-PRODUCTION <MODIFY>)
(SETQ
  <MODIFY> '(<MODIFY> ((ALTER ((* NIL NIL)) NIL) (CHANGE ((*
NIL NIL)) NIL)
            (MOD ((* NIL NIL)) NIL) (MODIFY ((* NIL
NIL)) NIL)
            (REPLACE ((* NIL NIL)) NIL))
  NIL))
```

```

(DECLARE (SPECIAL <PRESENT>))
(CREATE-PRODUCTION <PRESENT>)
(SETQ
  <PRESENT>
  '(<PRESENT> ((WHO ((IS ((* NIL NIL)) NIL)) NIL)
              (WHAT ((IS ((* NIL NIL)) NIL) (ARE ((* NIL NIL))
NIL)) NIL)
              (TYPE ((* NIL NIL)) NIL)
              (TELL ((ME ((* NIL NIL)) NIL) (* NIL NIL)) NIL)
              (SHOW ((ME ((* NIL NIL)) NIL) (* NIL NIL)) NIL)
              (PRINT ((* NIL NIL)) NIL) (PRESENT ((* NIL NIL))
NIL)
              (DISPLAY ((* NIL NIL)) NIL))
  NIL))

```

```

(DECLARE (SPECIAL <META-COMMAND>))
(CREATE-PRODUCTION <META-COMMAND>)
(SETQ
  <META-COMMAND>
  '(<META-COMMAND> (<SAVE> ((* NIL NIL)) NIL) (<HELP> ((*
NIL NIL)) NIL)
              (<EXIT> ((* NIL NIL)) NIL) (<CLS> ((* NIL
NIL)) NIL))
  NIL))

```

```

(DECLARE (SPECIAL <CLS>))
(CREATE-PRODUCTION <CLS>)
(SETQ
  <CLS>
  '(<CLS>
    ((CLS ((* NIL NIL)) NIL)
     (CLEAR
      ((THE ((SCREEN ((* NIL NIL)) NIL)) NIL) (SCREEN ((* NIL
NIL)) NIL))
      NIL))
    (CLS)))

```

```

(DECLARE (SPECIAL <EXIT>))
(CREATE-PRODUCTION <EXIT>)
(SETQ <EXIT> '(<EXIT> ((EXIT ((* NIL NIL)) NIL) (QUIT ((*
NIL NIL)) NIL)
              (ALL ((POW ((* NIL NIL)) NIL)) NIL))
              (SETQ *EXIT* T)))

```

```

(DECLARE (SPECIAL <HELP>))
(CREATE-PRODUCTION <HELP>)
(SETQ
  <HELP>
  '(<HELP>
    ((HELP ((* NIL NIL)) NIL) (H ((* NIL NIL)) NIL) (? ((*
NIL NIL)) NIL))
    (HELP)))

```

```

(DECLARE (SPECIAL <INSTANCE>))
(CREATE-PRODUCTION <INSTANCE>)
(SETQ <INSTANCE>
  '(<INSTANCE> ((<VARIABLE-INSTANCE> ((* NIL NIL)) NIL)
    (<REQUIREMENTS-INSTANCE> ((* NIL NIL)) NIL)
    (<PROJECT-INSTANCE> ((* NIL NIL)) NIL)
    (<PROCESS-ALIAS-INSTANCE> ((* NIL NIL)) NIL)
    (<PROCESS-INSTANCE> ((* NIL NIL)) NIL)
    (<PARAMETER-ALIAS-INSTANCE> ((* NIL NIL)) NIL)
    (<PARAMETER-INSTANCE> ((* NIL NIL)) NIL)
    (<MODULE-INSTANCE> ((* NIL NIL)) NIL)
    (<IMPLEMENTATION-INSTANCE> ((* NIL NIL)) NIL)
    (<DESIGN-INSTANCE> ((* NIL NIL)) NIL)
    (<DATA-ELEMENT-ALIAS-INSTANCE> ((* NIL NIL))
NIL)
    (<DATA-ELEMENT-INSTANCE> ((* NIL NIL)) NIL)
    (<ACTIVITY-ALIAS-INSTANCE> ((* NIL NIL)) NIL)
    (<ACTIVITY-INSTANCE> ((* NIL NIL)) NIL))
NIL))

```

```

(DECLARE (SPECIAL <ACTIVITY-INSTANCE>))
(CREATE-PRODUCTION <ACTIVITY-INSTANCE>)
(SETQ <ACTIVITY-INSTANCE>
  '(<ACTIVITY-INSTANCE>
    ((ACTIVITY ((<ACTIVITY> ((* NIL NIL)) NIL)) NIL))
NIL))

```

```

(DECLARE (SPECIAL <ACTIVITY-ALIAS-INSTANCE>))
(CREATE-PRODUCTION <ACTIVITY-ALIAS-INSTANCE>)
(SETQ
  <ACTIVITY-ALIAS-INSTANCE>
  '(<ACTIVITY-ALIAS-INSTANCE>
    ((ACTIVITY ((ALIAS ((<ACTIVITY-ALIAS> ((* NIL NIL))
NIL)) NIL)) NIL))
    NIL))

```



```

(DECLARE (SPECIAL <DATA-ELEMENT-INSTANCE>))
(CREATE-PRODUCTION <DATA-ELEMENT-INSTANCE>)
(SETQ <DATA-ELEMENT-INSTANCE>
      '(<DATA-ELEMENT-INSTANCE>
        ((DATA ((ELEMENT ((<DATA-ELEMENT> ((* NIL NIL)) NIL))
        NIL)) NIL))
        NIL))

```

```

(DECLARE (SPECIAL <DATA-ELEMENT-ALIAS-INSTANCE>))
(CREATE-PRODUCTION <DATA-ELEMENT-ALIAS-INSTANCE>)
(SETQ
  <DATA-ELEMENT-ALIAS-INSTANCE>
  '(<DATA-ELEMENT-ALIAS-INSTANCE>
    ((DATA
      (ELEMENT
        ((ALIAS ((<DATA-ELEMENT-ALIAS> ((* NIL NIL)) NIL))
        NIL)) NIL))
    NIL))
  NIL))

```

```

(DECLARE (SPECIAL <DESIGN-INSTANCE>))
(CREATE-PRODUCTION <DESIGN-INSTANCE>)
(SETQ
  <DESIGN-INSTANCE>
  '(<DESIGN-INSTANCE> ((DESIGN ((<DESIGN> ((* NIL NIL))
  NIL)) NIL)) NIL))

```

```

(DECLARE (SPECIAL <IMPLEMENTATION-INSTANCE>))
(CREATE-PRODUCTION <IMPLEMENTATION-INSTANCE>)
(SETQ
  <IMPLEMENTATION-INSTANCE>
  '(<IMPLEMENTATION-INSTANCE>
    ((IMPLEMENTATION ((<IMPLEMENTATION> ((* NIL NIL)) NIL))
    NIL)) NIL))

```

```

(DECLARE (SPECIAL <MODULE-INSTANCE>))
(CREATE-PRODUCTION <MODULE-INSTANCE>)
(SETQ
  <MODULE-INSTANCE>
  '(<MODULE-INSTANCE> ((MODULE ((<MODULE> ((* NIL NIL))
  NIL)) NIL)) NIL))

```

```

(DECLARE (SPECIAL <PARAMETER-INSTANCE>))
(CREATE-PRODUCTION <PARAMETER-INSTANCE>)
(SETQ <PARAMETER-INSTANCE>
      '(<PARAMETER-INSTANCE>
        ((PARAMETER ((<PARAMETER> ((* NIL NIL)) NIL)) NIL))
      NIL))

```

```

(DECLARE (SPECIAL <PARAMETER-ALIAS-INSTANCE>))
(CREATE-PRODUCTION <PARAMETER-ALIAS-INSTANCE>)
(SETQ <PARAMETER-ALIAS-INSTANCE>
  '(<PARAMETER-ALIAS-INSTANCE>
    ((PARAMETER
      ((ALIAS ((<PARAMETER-ALIAS> ((* NIL NIL)) NIL))
        NIL)) NIL))
    NIL))

(DECLARE (SPECIAL <PROCESS-INSTANCE>))
(CREATE-PRODUCTION <PROCESS-INSTANCE>)
(SETQ <PROCESS-INSTANCE>
  '(<PROCESS-INSTANCE>
    ((PROCESS ((<PROCESS> ((* NIL NIL)) NIL)) NIL)) NIL))

(DECLARE (SPECIAL <PROCESS-ALIAS-INSTANCE>))
(CREATE-PRODUCTION <PROCESS-ALIAS-INSTANCE>)
(SETQ
  <PROCESS-ALIAS-INSTANCE>
  '(<PROCESS-ALIAS-INSTANCE>
    ((PROCESS ((ALIAS ((<PROCESS-ALIAS> ((* NIL NIL)) NIL))
      NIL)) NIL))
    NIL))

(DECLARE (SPECIAL <PROJECT-INSTANCE>))
(CREATE-PRODUCTION <PROJECT-INSTANCE>)
(SETQ
  <PROJECT-INSTANCE>
  '(<PROJECT-INSTANCE>
    ((*PROJECT* ((* NIL (SEND *NEW-EVENT* :SET-OBJECT
*PROJECT*)) NIL)
      (PROJECT ((<PROJECT> ((* NIL NIL)) NIL)) NIL))
    NIL))

(DECLARE (SPECIAL <REQUIREMENTS-INSTANCE>))
(CREATE-PRODUCTION <REQUIREMENTS-INSTANCE>)
(SETQ <REQUIREMENTS-INSTANCE>
  '(<REQUIREMENTS-INSTANCE>
    ((REQUIREMENTS ((<REQUIREMENTS> ((* NIL NIL)) NIL))
      NIL)) NIL))

(DECLARE (SPECIAL <VARIABLE-INSTANCE>))
(CREATE-PRODUCTION <VARIABLE-INSTANCE>)
(SETQ <VARIABLE-INSTANCE>
  '(<VARIABLE-INSTANCE>
    ((VARIABLE ((<VARIABLE> ((* NIL NIL)) NIL)) NIL))
  NIL))

```

```

(DECLARE (SPECIAL <ACTIVITY>))
(CREATE-FUNCTION <ACTIVITY>)
(SETQ <ACTIVITY> '(<ACTIVITY> (LAMBDA (X) NIL)))

(DECLARE (SPECIAL <ACTIVITY-ALIAS>))
(CREATE-FUNCTION <ACTIVITY-ALIAS>)
(SETQ <ACTIVITY-ALIAS> '(<ACTIVITY-ALIAS> (LAMBDA (X) NIL)))

(DECLARE (SPECIAL <DATA-ELEMENT>))
(CREATE-FUNCTION <DATA-ELEMENT>)
(SETQ <DATA-ELEMENT> '(<DATA-ELEMENT> (LAMBDA (X) NIL)))

(DECLARE (SPECIAL <DATA-ELEMENT-ALIAS>))
(CREATE-FUNCTION <DATA-ELEMENT-ALIAS>)
(SETQ <DATA-ELEMENT-ALIAS> '(<DATA-ELEMENT-ALIAS> (LAMBDA
(X) NIL)))

(DECLARE (SPECIAL <DESIGN>))
(CREATE-FUNCTION <DESIGN>)
(SETQ <DESIGN>
'(<DESIGN>
(LAMBDA (X)
(LET ((INSTANCE (SEND *PROJECT* :IS-DESIGN X)))
(COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
INSTANCE)))))))

(DECLARE (SPECIAL <IMPLEMENTATION>))
(CREATE-FUNCTION <IMPLEMENTATION>)
(SETQ <IMPLEMENTATION>
'(<IMPLEMENTATION>
(LAMBDA (X)
(LET ((INSTANCE (SEND *PROJECT* :IS-IMPLEMENTATION
X)))
(COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
INSTANCE)))))))

(DECLARE (SPECIAL <MODULE>))
(CREATE-FUNCTION <MODULE>)
(SETQ <MODULE> '(<MODULE> (LAMBDA (X) NIL)))

```

```

(DECLARE (SPECIAL <PARAMETER>))
(CREATE-FUNCTION <PARAMETER>)
(SETQ
  <PARAMETER>
  '(<PARAMETER>
    (LAMBDA (X)
      (LET ((INSTANCE
              (SEND (SEND *PROJECT* :DESIGN-INSTANCE)
                    :IS-PARAMETER X)))
          (COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
                                INSTANCE)))))))

(DECLARE (SPECIAL <PARAMETER-ALIAS>))
(CREATE-FUNCTION <PARAMETER-ALIAS>)
(SETQ
  <PARAMETER-ALIAS>
  '(<PARAMETER-ALIAS>
    (LAMBDA (X)
      (LET ((INSTANCE
              (SEND
                (SEND *PROJECT* :DESIGN-INSTANCE)
                :IS-PARAMETER-ALIAS X)))
          (COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
                                INSTANCE)))))))

(DECLARE (SPECIAL <PROCESS>))
(CREATE-FUNCTION <PROCESS>)
(SETQ <PROCESS>
  '(<PROCESS>
    (LAMBDA (X)
      (LET ((INSTANCE
              (SEND (SEND *PROJECT* :DESIGN-INSTANCE)
                    :IS-PROCESS X)))
          (COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
                                INSTANCE)))))))

(DECLARE (SPECIAL <PROCESS-ALIAS>))
(CREATE-FUNCTION <PROCESS-ALIAS>)
(SETQ
  <PROCESS-ALIAS>
  '(<PROCESS-ALIAS>
    (LAMBDA (X)
      (LET
        ((INSTANCE
          (SEND (SEND *PROJECT* :DESIGN-INSTANCE)
                :IS-PROCESS-ALIAS X)))
          (COND (INSTANCE (SEND *NEW-EVENT* :SET-OBJECT
                                INSTANCE)))))))

```

```

(DECLARE (SPECIAL <PROJECT>))
(CREATE-FUNCTION <PROJECT>)
(SETQ <PROJECT>
  '(<PROJECT>
    (LAMBDA (X) (COND ((EQUAL X (SEND *PROJECT* :NAME))
                      (SEND *NEW-EVENT* :SET-OBJECT
                            *PROJECT*))))))

(DECLARE (SPECIAL <VARIABLE>))
(CREATE-FUNCTION <VARIABLE>)
(SETQ <VARIABLE> '(<VARIABLE> (LAMBDA (X) NIL)))

(DECLARE (SPECIAL <ATTRIBUTE-LIST>))
(CREATE-PRODUCTION <ATTRIBUTE-LIST>)
(SETQ <ATTRIBUTE-LIST>
  '(<ATTRIBUTE-LIST>
    ((<ATTRIBUTE> ((AND ((<ATTRIBUTE> (* NIL NIL)) NIL))
                    (<ATTRIBUTE-LIST> (* NIL NIL)) NIL) (* NIL
                    NIL))
    NIL))
  NIL))
  NIL))
  NIL))

```

```

(DECLARE (SPECIAL <ATTRIBUTE>))
(CREATE-PRODUCTION <ATTRIBUTE>)
(SETQ
  <ATTRIBUTE>
  '(<ATTRIBUTE>
    ((GLOBAL
      (DATA
        (CHANGED
          (* NIL
            (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'GLOBAL-DATA-CHANGED)))
          NIL)
        (USED
          (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'GLOBAL-DATA-USED)))
          NIL))
      NIL))
    (DESTINATIONS
      (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-TO))) NIL)
    (SOURCES
      (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-FROM))) NIL)
    (SYNONYMS (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'SYNONYM))) NIL)
    (DESTINATION
      (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-TO))) NIL)
    (SOURCE (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-FROM))) NIL)
    (PASSED
      (TO (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-TO))) NIL)
      (FROM
        (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PASSED-FROM))) NIL))
    (OUTPUT
      (FLAGS
        (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'OUTPUT-FLAGS))) NIL)
      (DATA
        (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'OUTPUT-DATA))) NIL))
    NIL)

```

```

(INPUT
  ((FLAGS
    (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' INPUT-FLAGS))) NIL)
  (DATA ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' INPUT-DATA))) NIL))
  NIL)
  (ALIASES ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' ALIASES))) NIL)
  (GLOBALS
    (USED ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' GLOBAL-DATA-USED)))
      NIL)
    (CHANGED
      (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' GLOBAL-DATA-CHANGED)))
      NIL))
      NIL)
      (INVOKING
        ((PROCESSES
          (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' CALLING-PROCESSES)))
          NIL))
          NIL)
          (ENTRY
            ((DATE ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' DATE))) NIL)
            (VERSION ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' VERSION))) NIL))
            NIL)
            (SYNONYM ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' SYNONYM))) NIL)
            (REQUIREMENTS
              (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' REQUIREMENTS))) NIL)
            (SUBPROCESSES
              (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' PROCESSES-CALLED))) NIL)
            (INVOKED
              ((PROCESSES
                (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
' PROCESSES-CALLED)))
                NIL))
              NIL)
              NIL)

```

```

(CALLED
  ((PROCESSES
    ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PROCESSES-CALLED)))
    NIL))
  NIL)
  (PROCESSES
    ((INVOKED
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PROCESSES-CALLED)))
      NIL)
    (CALLED
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PROCESSES-CALLED)))
      NIL)
      (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE 'PROCESSES)))
    NIL)
  (PROCESS
    ((ALIASES
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PROCESS-ALIASES)))
      NIL))
    NIL)
  (PARAMETERS
    ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PARAMETERS))) NIL)
  (PARAMETER
    ((ALIASES
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PARAMETER-ALIASES)))
      NIL))
    NIL)
  (MAIN
    ((PROGRAM
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'MAIN-PROGRAM))) NIL)
    (PROGRAMS
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'MAIN-PROGRAM))) NIL)
    (PROCESS
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'MAIN-PROGRAM))) NIL)
    (PROCESSES
      ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'MAIN-PROGRAM))) NIL))
    NIL)
  (IMPLEMENTATION
    ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'IMPLEMENTATION))) NIL)

```



```

      (DESIGN ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'DESIGN))) NIL)
      (DD ((TYPE ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'DD-TYPE)))) NIL))
      NIL)
      (CALLING
        ((PROCESSES
          ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'CALLING-PROCESSES))))
        NIL))
      NIL)
      (VERSION ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'VERSION))) NIL)
      (TYPE ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE 'TYPE)))
NIL)
      (PROJECT
        ((NAME ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE
'PROJECT)))) NIL)
        (* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE 'PROJECT)))
      NIL)
      (NAME ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE 'NAME)))
NIL)
      (DATE ((* NIL (SEND *NEW-EVENT* :ADD-ATTRIBUTE 'DATE)))
NIL))
      NIL))

```

```

(DECLARE (SPECIAL <LITERAL>))
(CREATE-FUNCTION <LITERAL>)
(SETQ <LITERAL> '(<LITERAL> (LAMBDA (X) NIL)))

```

```

(DECLARE (SPECIAL <SAVE>))
(CREATE-PRODUCTION <SAVE>)
(SETQ <SAVE> '(<SAVE> ((STORE ((* NIL NIL)) NIL) (KEEP ((*
NIL NIL)) NIL)
                (SAVE ((* NIL NIL)) NIL))
        (SEND *PROJECT* :SAVE)))

```

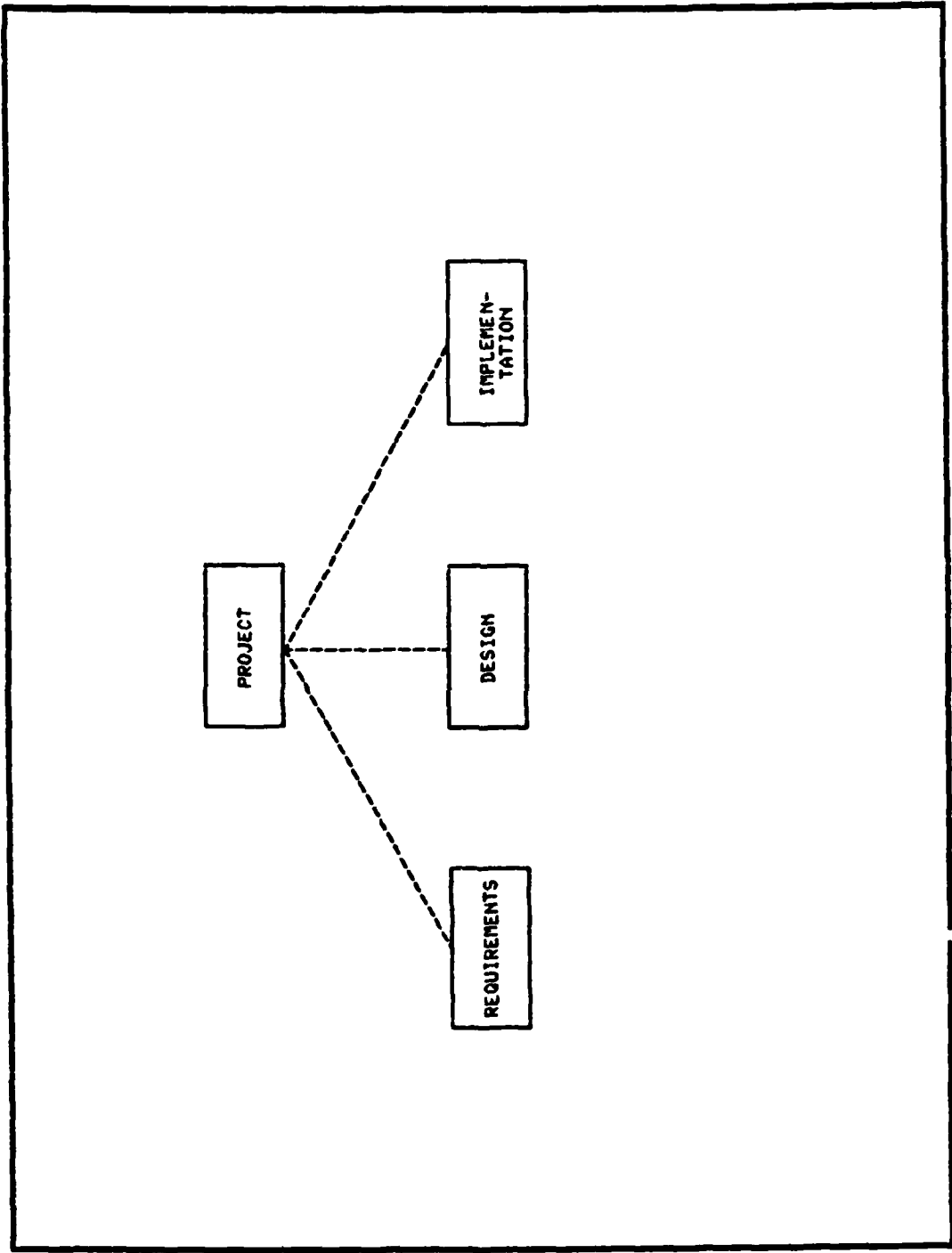
Appendix E
DDT Object Definitions

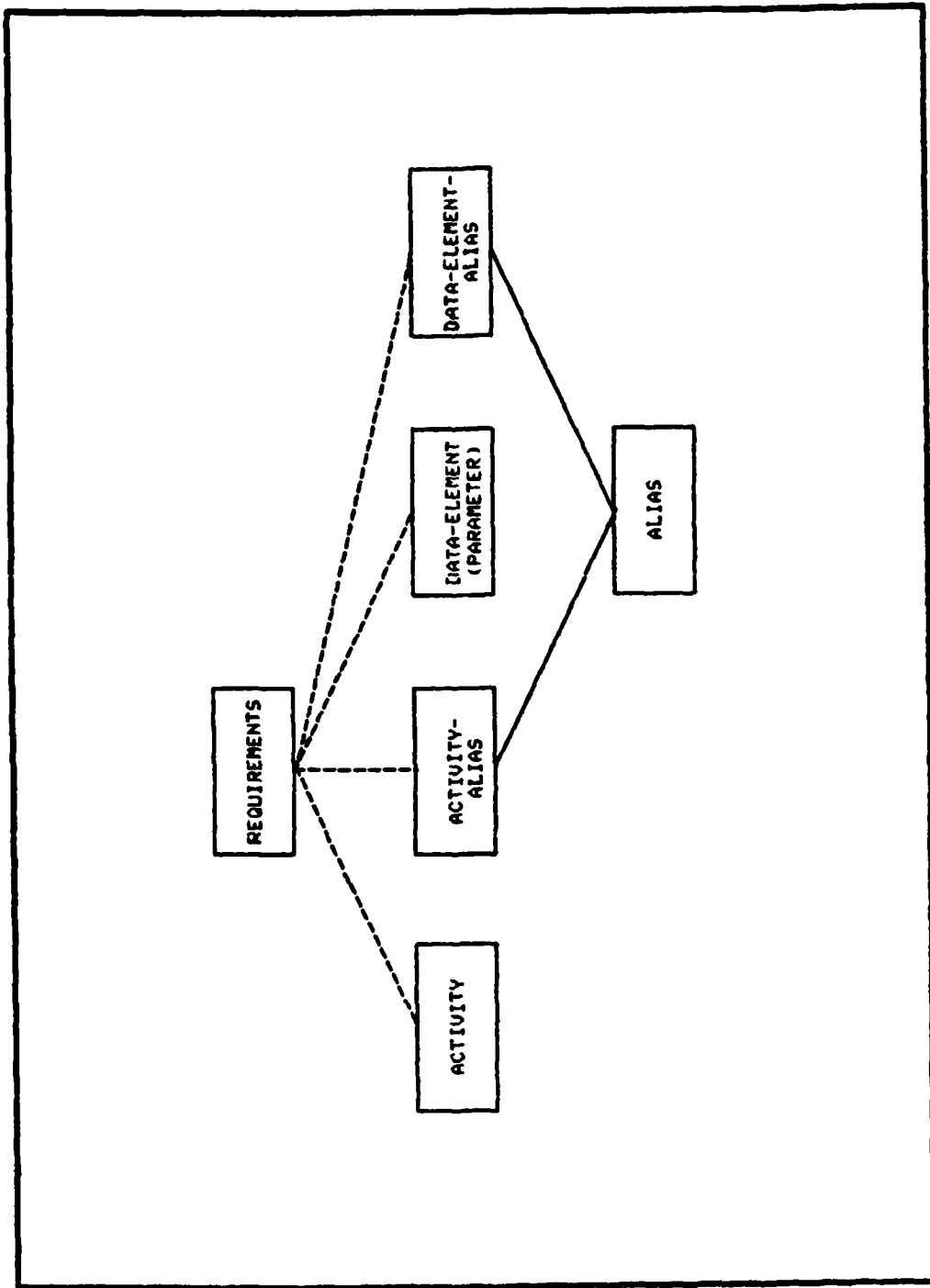
Appendix E
Table of Contents

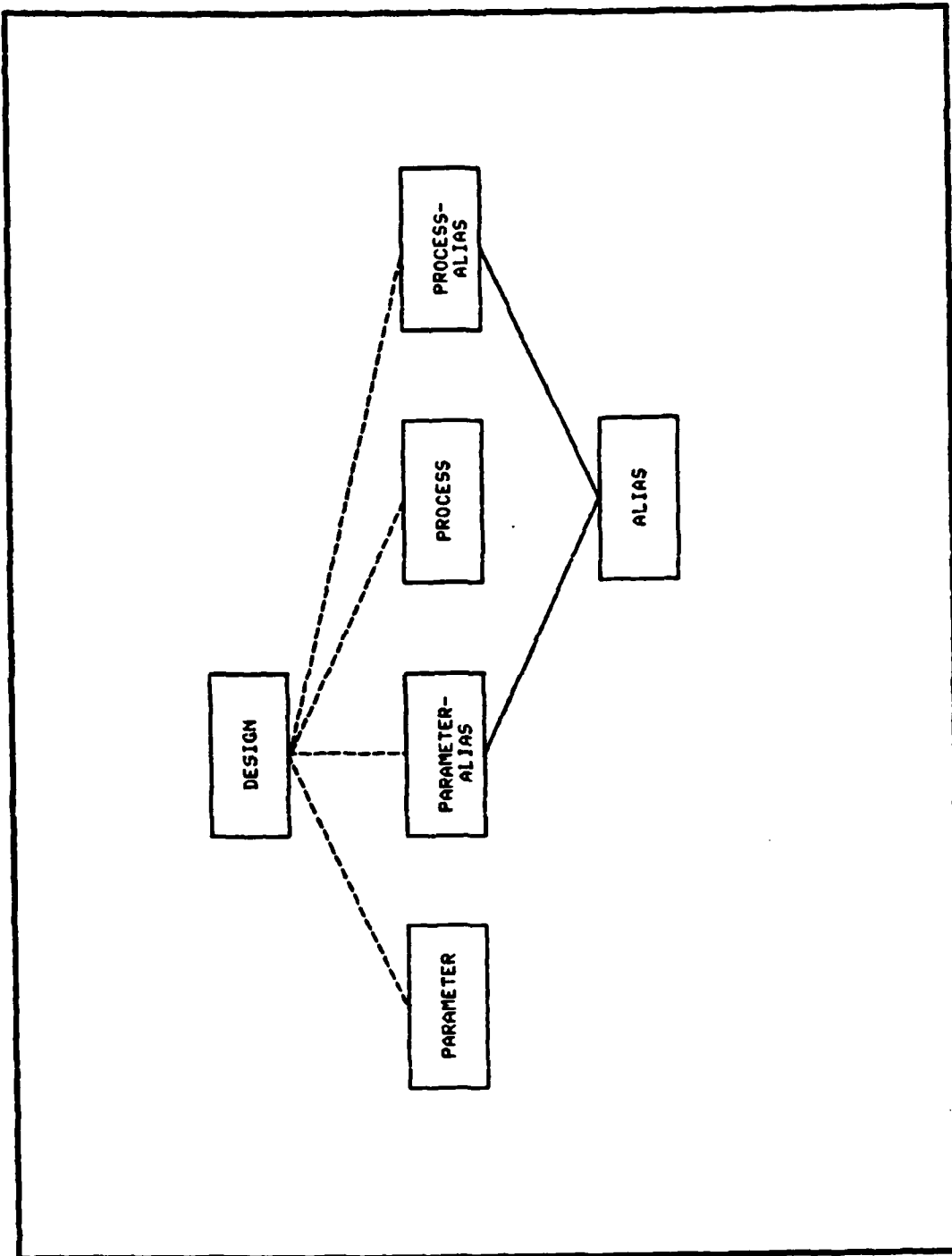
	Page
Introduction	E- 3
DDT Object Hierarchy Charts	E- 4
DDT Object Definitions	E- 8

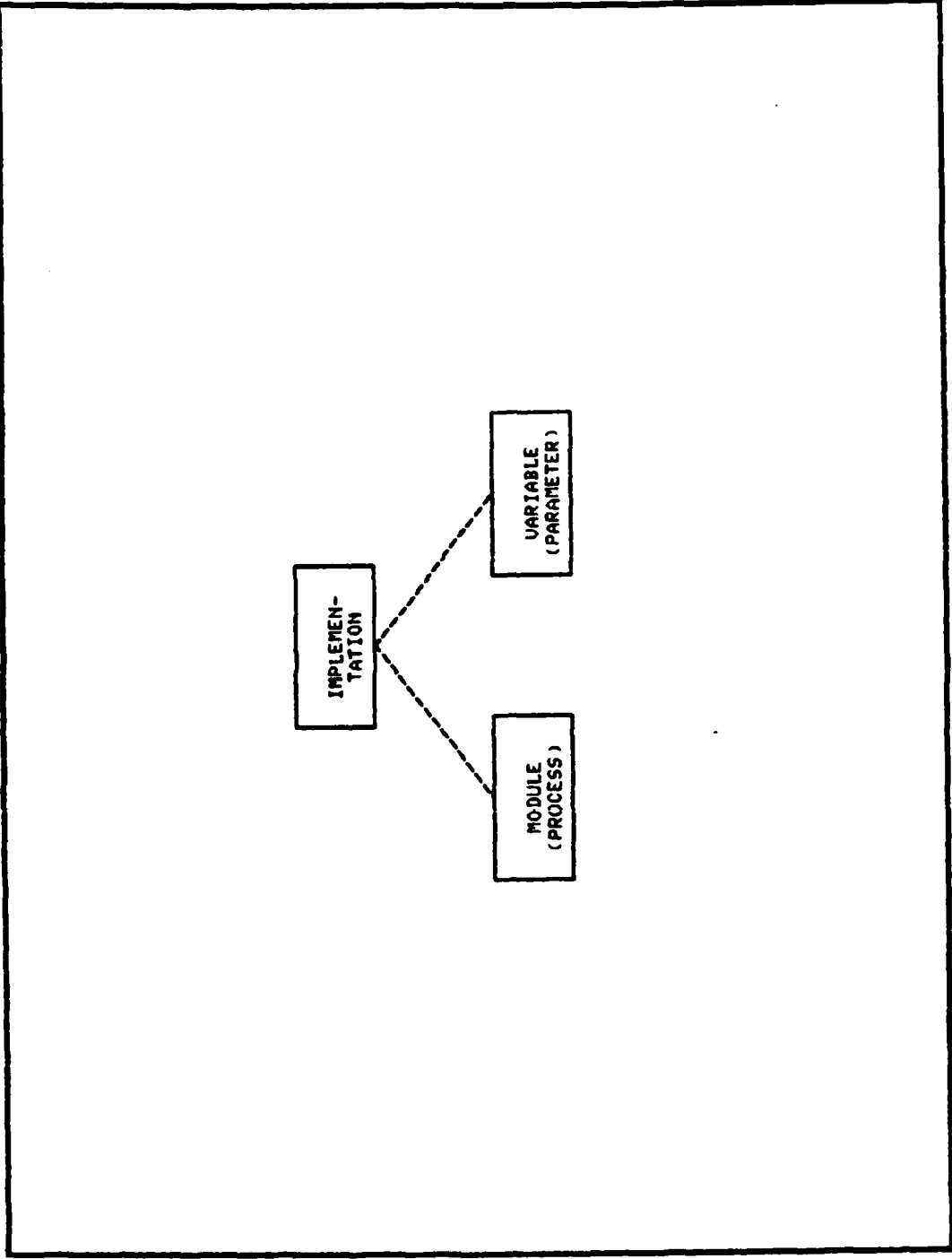
Introduction

This appendix defines the object-oriented structure of the Data Dictionary Tool (DDT). The first part of this definition shows the hierarchy of DDT's object classes. The second part defines the structure of each object class in terms of its included subclasses, its instance variables, and its associated access methods.









Object Class: ACTIVITY

Subclasses

HEADER
ENTRY

Instance Variables

NUMBER	Node number of activity
DESCRIPTION	Textual description of activity
INPUTS	Assoc-list of input parameter names and numbers
OUTPUTS	Assoc-list of output parameter names and numbers
CONTROLS	Assoc-list of control parameter names and numbers
MECHANISMS	Assoc-list of mechanism parameter and numbers
ALIASES	List of activity alias names
PARENT-ACTIVITY	Name of parent activity
RELATED-ITEMS	Paragraph number of textual requirements statement

Object Class: ACTIVITY-ALIAS

Subclasses

ALIAS

Instance Variables

WHERE-USED

List of SADT activity numbers

Object Class: ALIAS

Subclasses

HEADER

ENTRY

Instance Variables

DD-TYPE

Data dictionary type
("ACTIVITY", "DATA ELEMENT",
"PARAMETER", "PROCESS")

SYNONYM

Name of a defined module or
parameter

Methods

dd-type

Returns the value of DD-TYPE

set-dd-type (VAL)

Sets the value of DD-TYPE to VAL

synonym

Returns the value of SYNONYM

set-synonym (VAL)

Sets the value of SYNONYM to VAL

Object Class: ASSOC

Subclasses

none

Instance Variable

VALUE

Methods

initialize

Initializes all of ASSOC's
instance variables

present

Displays an ASSOC instance

value

Returns the value of VALUE

set-value (VAL)

Sets VALUE to VAL

Object Class: ATOM

Subclasses

none

Instance Variables

VALUE

Methods

initialize

Initializes all of ATOM's
instance variables

present

Displays an ATOM instance

value

Returns the value of VALUE

set-value (VAL)

Sets VALUE to VAL

Object Class: DATA-ELEMENT

Subclasses
PARAMETER

Instance Variables
none

Object Class: DATA-ELEMENT-ALIAS

Subclasses

ALIAS

Instance Variables

WHERE-USED

List of SADT activity numbers

Object class: DATE

Subclasses

none

Instance Variables

VALUE

Methods

initialize

Initializes all of DATE's
instance variables

present

Displays a DATE instance

value

Returns the value of VALUE

set-value (VAL)

Sets VALUE to VAL

day

Returns the value of the DAY
subfield of VALUE

month

Returns the value of the MONTH
subfield of VALUE

year

Returns the value of the YEAR
subfield of VALUE

Object Class: DESIGN

Subclasses

HEADER
ENTRY

Instance Variables

MAIN-PROGRAM
PROCESSES

PROCESS-ALIASES

PARAMETERS

PARAMETER-ALIASES

Name of main program

Assoc-list of module names and
pointers to module instances

Assoc-list of process alias names
and pointers to alias instances

Assoc-list of parameter names and
pointers to parameter instances

Assoc-list of parameter alias
names and pointers to alias
instances

Methods

create
initialize

Creates a DESIGN instance
Initializes all of DESIGN's
instance variables

present (ATTRIBUTE-
LIST)

Displays the instance variables
listed in ATTRIBUTE-LIST of a
DESIGN instance; if
ATTRIBUTE-LIST is empty, displays
all instance variables

reinitialize (DUMMY)

Initializes all of DESIGN's
instance variables except NAME,
TYPE, and PROJECT

save (PROJECT,
STREAM)

Saves a DESIGN instance and all
of its sub-instances to STREAM

initialize-date

Initializes DATE

initialize-main-
program

Initializes MAIN-PROGRAM

main-program

Returns the value of MAIN-PROGRAM

set-main-program
(VAL)

Sets the value of MAIN-PROGRAM to
VAL

initialize-name

Initializes NAME

create-parameter-alias	Creates a PARAMETER-ALIAS instance and adds it to the list of PARAMETER-ALIASES
initialize-parameter-aliases parameter-aliases	Initializes PARAMETER-ALIASES Returns the value of PARAMETER-ALIASES
parameter-alias-instance-list	Returns a list of pointers to the PARAMETER-ALIASES of a DESIGN
set-parameter-aliases (VAL)	Sets the value of PARAMETER-ALIASES to VAL
is-parameter-alias (VAL)	Returns a pointer to the appropriate instance if VAL is a PARAMETER-ALIAS, otherwise returns NIL
create-parameter	Creates a PARAMETER instance and adds it to the list of PARAMETERS
initialize-parameters parameters	Initializes PARAMETERS Returns the value of PARAMETERS
parameter-instance-list	Returns a list of pointers to the PARAMETERS of a DESIGN
set-parameters (VAL)	Sets the value of PARAMETERS to VAL
is-parameter (VAL)	Returns a pointer to the appropriate instance if VAL is a PARAMETER, otherwise returns NIL

create-process-alias	Creates a PROCESS-ALIAS instance and adds it to the list of PROCESS-ALIASES
initialize-process-aliases process-aliases	Initializes PROCESS-ALIASES Returns the value of PROCESS-ALIASES
process-alias-instance-list	Returns a list of pointers to the PROCESS ALIASes of a DESIGN
set-process-aliases (VAL)	Sets the value of PROCESS-ALIASES to VAL
is-process-alias (VAL)	Returns a pointer to the appropriate instance if VAL is a PROCESS-ALIAS, otherwise returns NIL
create-process	Creates a PROCESS instance and adds it to the list of PROCESSES
initialize-processes processes	Initializes PROCESSES Returns the value of PROCESSES
process-instance-list	Returns a list of pointers to the PROCESSES of a DESIGN
set-processes (VAL)	Sets the value of PROCESSES to VAL
is-process (VAL)	Returns a pointer to the appropriate instance if VAL is a PROCESS, otherwise returns NIL
initialize-project	Initializes PROJECT
initializ -type	Initializes TYPE
initialize-version	Initializes VERSION

Object Class: ENTRY

Subclasses

none

Instance Variables

VERSION

Version number of entry

DATE

Date of entry

Methods

date

Returns the value of DATE

version

Returns the value of VERSION

increment-version

Increments the value of VERSION
by 1

Object Class: HEADER

Subclasses

none

Instance Variables

NAME

Name of instance

TYPE

Type of instance

("ACTIVITY", "ALIAS", "DATA
ELEMENT", "DESIGN",
"IMPLEMENTATION", "MODULE",
"PARAMETER", "PROCESS",
"REQUIREMENTS", "VARIABLE")

PROJECT

Name of project

Methods

name

Returns the value of NAME

set-name (VAL)

Sets the value of NAME to VAL

project

Returns the value of PROJECT

type

Returns the value of TYPE

Object Class: IMPLEMENTATION

Subclasses

HEADER
ENTRY

Instance Variables

MAIN-PROGRAM
MODULES

Name of main program

Assoc-list of module names and
pointers to module instances

VARIABLES

Assoc-list of variable names and
pointers to variable instances

Methods

create

Creates an IMPLEMENTATION
instance

initialize

Initializes all of
IMPLEMENTATION's instance
variables

present (ATTRIBUTE-
LIST)

Displays the instance variables
listed in ATTRIBUTE-LIST of an
IMPLEMENTATION instance; if
ATTRIBUTE-LIST is empty, displays
all instance variables

reinitialize (DUMMY)

Initializes all of
IMPLEMENTATION's instance
variables except NAME, TYPE, and
PROJECT

save (PROJECT,
STREAM)

Saves an IMPLEMENTATION instance
and all of its sub-instances to
STREAM

Object Class: LIST

Subclasses

none

Instance Variables

VALUE

Methods

initialize

Initializes all of LIST's
instance variables

present

Displays a LIST instance

value

Returns the value of VALUE

set-value (VAL)

Sets VALUE to VAL

Object Class: MODULE

Subclasses
PROCESS

Instance Variables
none

Object Class: PARAMETER

Subclasses

HEADER
ENTRY

Instance Variables

<u>DESCRIPTION</u>	Textual description of parameter/variable/data element
DATA-TYPE	Data type of parameter/variable/data element
MIN-VALUE	Minimum value of parameter/variable/data element (if applicable)
MAX-VALUE	Maximum value of parameter/variable/data element (if applicable)
RANGE	Range of parameter/variable/data element (if applicable)
VALUES	List of legal values of parameter/variable/data element (if applicable)
STORAGE-TYPE	Storage type of parameter/variable/data element ("FILE", "GLOBAL", "HARDWARE", "I/O", or "PASSED")
PART-OF	Name of parent parameter/variable/data element (if applicable)
COMPOSITION	List of sub-parameter/variable/data element names (if applicable)
ALIASES	List of parameter/variable/data element alias names (if any)
PASSED-FROM	List of process/module/activity names passed from
PASSED-TO	List of process/module/activity names passed to
RELATED-ITEMS	List of related SADT-data-item/parameter/require- ment names/numbers

Methods

create	Creates a PARAMETER instance
initialize	Initializes all of PARAMETER's instance variables
present (ATTRIBUTE-LIST)	Displays the instance variables listed in ATTRIBUTE-LIST of a PARAMETER instance; if ATTRIBUTE-LIST is empty, displays all instance variables
reinitialize (DUMMY)	Initializes all of PARAMETER's instance variables except NAME, TYPE, and PROJECT
save (DESIGN, STREAM)	Saves a PARAMETER instance to STREAM
initialize-aliases	Initializes ALIASES
aliases	Returns the value of ALIASES
set-aliases (VAL)	Sets the value of ALIASES to VAL
initialize-composition	Initializes COMPOSITION
composition	Returns the value of COMPOSITION
set-composition (VAL)	Sets the value of COMPOSITION to VAL
initialize-data-type	Initializes DATA-TYPE
data-type	Returns the value of DATA-TYPE
set-data-type (VAL)	Sets the value of DATA-TYPE to VAL
initialize-date	Initializes DATE
initialize-description	Initializes DESCRIPTION
description	Returns the value of DESCRIPTION
set-description (VAL)	Sets the value of DESCRIPTION to VAL
initialize-max-value	Initializes MAX-VALUE
max-value	Returns the value of MAX-VALUE
set-max-value (VAL)	Sets the value of MAX-VALUE to VAL
initialize-min-value	Initializes MIN-VALUE
min-value	Returns the value of MIN-VALUE
set-min-value (VAL)	Sets the value of MIN-VALUE to VAL

initialize-name	Initializes NAME
initialize-part-of part-of set-part-of (VAL)	Initializes PART-OF Returns the value of PART-OF Sets the value of PART-OF to VAL
initialize-passed- from passed-from set-passed-from (VAL)	Initializes PASSED-FROM Returns the value of PASSED-FROM Sets the value of PASSED-FROM to VAL
initialize-passed-to passed-to set-passed-to (VAL)	Initializes PASSED-TO Returns the value of PASSED-To Sets the value of PASSED-TO to VAL
initialize-project	Initializes PROJECT
initialize-range range set-range (VAL)	Initializes RANGE Returns the value of RANGE Sets the value of RANGE to VAL
initialize-related- items related-items set-related-items (VAL)	Initializes RELATED-ITEMS Returns the value of RELATED-ITEMS Sets the value of RELATED-ITEMS to VAL
initialize-storage- type storage-type set-storage-type (VAL)	Initializes STORAGE-TYPE Returns the value of STORAGE-TYPE Sets the value of STORAGE-TYPE to VAL
initialize-type	Initializes TYPE
initialize-values values set-values (VAL)	Initializes VALUES Returns the value of VALUES Sets the value of VALUES to VAL
initialize-version	Initializes VERSION

Object Class: PARAMETER-ALIAS

Subclasses
ALIAS

Instance Variables
PASSED-FROM

List of process names parameter
is passed from

PASSED-TO

List of process names parameter
is passed to

Methods
create

Creates a PARAMETER-ALIAS
instance

initialize

Initializes all of
PARAMETER-ALIAS' instance
variables

present (ATTRIBUTE-
LIST)

Displays the instance variables
listed in ATTRIBUTE-LIST of a
PARAMETER-ALIAS instance; if
ATTRIBUTE-LIST is empty, displays
all instance variables

reinitialize (DUMMY)

Initializes all of
PARAMETER-ALIAS' instance
variables except NAME, TYPE,
PROJECT, and DD-TYPE
Saves a PARAMETER-ALIAS instance
to STREAM

save (DESIGN, STREAM)

initialize-date

Initializes DATE

initialize-dd-type

Initializes DD-TYPE

initialize-name

Initializes NAME

initialize-passed-
from

Initializes PASSED-FROM

passed-from

Returns the value of PASSED-FROM

set-passed-from (VAL)

Sets the value of PASSED-FROM to
VAL

initialize-passed-to
passed-to

Initializes PASSED-TO

set-passed-to (VAL)

Returns the value of PASSED-TO

Sets the values of PASSED-TO to
VAL

initialize-project

Initializes PROJECT

initialize-synonym	Initializes SYNONYM
initialize-type	Initializes TYPE
initialize-version	Initializes VERSION

Object Class: PROCESS

Subclasses

HEADER
ENTRY

Instance Variables

NUMBER	Number of process/module
DESCRIPTION	Textual description of process/module
INPUT-DATA	List of input data parameter/variable names
INPUT-FLAG	List of input flag parameter/variables names
OUTPUT-DATA	List of output data parameter/variable names
OUTPUT-FLAGS	List of output flag parameter/variables names
GLOBAL-DATA-USED	List of globals used
GLOBAL-DATA-CHANGED	List of globals changed
FILES-READ	List of filenames of files read
FILES-WRITTEN	List of filenames of files written
HARDWARE-INPUT	
HARDWARE-OUTPUT	
ALIASES	List of process/module alias names
CALLING-PROCESSES	List of calling process/module names
PROCESSES-CALLED	List of processes/modules called names
RELATED-ITEMS	List of related SADT-activity-names/process-names

Methods

create	Creates a PROCESS instance
initialize	Initializes all of PROCESS' instance variables
present (ATTRIBUTE-LIST)	Displays the instance variables listed in ATTRIBUTE-LIST of a PROCESS instance; if ATTRIBUTE-LIST is empty, displays all instance variables
reinitialize (DUMMY)	Initializes all of PROCESS' instance variables except NAME, TYPE, and PROJECT
save (DESIGN, STREAM)	Saves a PROCESS instance to STREAM
initialize-aliases	Initializes ALIASES
aliases	Returns the value of ALIASES
set-aliases (VAL)	Sets the value of ALIASES to VAL
initialize-calling-processes	Initializes CALLING-PROCESSES
calling-processes	Returns the value of CALLING-PROCESSES
set-calling-processes (VAL)	Sets the value of CALLING-PROCESSES to VAL
add-calling-processes (LIST)	For each process name in LIST: Adds to CALLING-PROCESSES; If necessary, Creates PROCESS instance; Adds NAME to PROCESSES-CALLED slot
delete-calling-processes (LIST)	For each process name in LIST: Deletes from CALLING-PROCESSES; Deletes NAME from PROCESSES-CALLED slot
initialize-date	Initializes DATE
initialize-description	Initializes DESCRIPTION
description	Returns the value of DESCRIPTION
set-description (VAL)	Sets the value of DESCRIPTION to VAL

initialize-files-read	Initializes FILES-READ
files-read	Returns the value of FILES-READ
set-files-read (VAL)	Sets the value of FILES-READ to VAL
initialize-files-written	Initializes FILES-WRITTEN
files-written	Returns the value of FILES-WRITTEN
set-files-written (VAL)	Sets the value of FILES-WRITTEN to VAL
initialize-globals-changed	Initializes GLOBAL-DATA-CHANGED
globals-changed	Returns the value of GLOBAL-DATA-CHANGED
set-globals-changed (VAL)	Sets the value of GLOBAL-DATA-CHANGED to VAL
initialize-globals-used	Initializes GLOBAL-DATA-USED
globals-used	Returns the value of GLOBAL-DATA-USED
set-globals-used (VAL)	Sets the value of GLOBAL-DATA-USED to VAL
initialize-hardware-input	Initializes HARDWARE-INPUT
hardware-input	Returns the value of HARDWARE-INPUT
set-hardware-input (VAL)	Sets the value of HARDWARE-INPUT to VAL
initialize-hardware-output	Initializes HARDWARE-OUTPUT
hardware-output	Returns the value of HARDWARE-OUTPUT
set-hardware-output (VAL)	Sets the value of HARDWARE-OUTPUT to VAL
initialize-input-data	Initializes INPUT-DATA
input-data	Returns the value of INPUT-DATA
set-input-data (VAL)	Sets the value of INPUT-DATA to VAL

initialize-input-flags	Initializes INPUT-FLAGS
input-flags	Returns the value of INPUT-FLAGS
set-input-flags (VAL)	Sets the value of INPUT-FLAGS to VAL
initialize-name	Initializes NAME
initialize-number	Initializes NUMBER
number	Returns the value of NUMBER
set-number (VAL)	Sets the value of NUMBER to VAL
initialize-output-	
data	Initializes OUTPUT-DATA
output-data	Returns the value of OUTPUT-DATA
set-output-data (VAL)	Sets the value of OUTPUT-DATA to VAL
initialize-output-	
flags	Initializes OUTPUT-FLAGS
output-flags	Returns the value of OUTPUT-FLAGS
set-output-flags (VAL)	Sets the value of OUTPUT-FLAGS to VAL
initialize-processes-	
called	Initializes PROCESSES-CALLED
processes-called	Returns the value of PROCESSES-CALLED
set-processes-called	
(VAL)	Sets the value of PROCESSES-CALLED to VAL
add-processes-called	
(LIST)	For each process name in LIST: Adds to PROCESSES-CALLED; If necessary, Creates PROCESS instance; Adds NAME to CALLING-PROCESSES slot
delete-processes-	
called (LIST)	For each process name in LIST: Deletes from PROCESSES-CALLED; Deletes NAME from CALLING-PROCESSES slot
initialize-project	Initializes PROJECT

initialize-related-
items
related-items

Initializes RELATED-ITEMS
Returns the value of
RELATED-ITEMS

set-related-items
(VAL)

Sets the value of RELATED-ITEMS
to VAL

initialize-type

Initializes TYPE

initialize-version

Initializes VERSION

Object Class: PROCESS-ALIAS

Subclasses
ALIAS

Instance Variables
none

Methods

create	Creates a PROCESS-ALIAS instance
initialize	Initializes all of PROCESS-ALIAS' instance variables
present (ATTRIBUTE-LIST)	Displays the instance variables listed in ATTRIBUTE-LIST of a PROCESS-ALIAS instance; if ATTRIBUTE-LIST is empty, displays all instance variables
reinitialize (DUMMY)	Initializes all of PROCESS-ALIAS' instance variables except NAME, TYPE, PROJECT, and DD-TYPE
save (DESIGN, STREAM)	Saves a PROCESS-ALIAS instance to STREAM
initialize-date	Initializes DATE
initialize-dd-type	Initializes DD-TYPE
initialize-name	Initializes NAME
initialize-project	Initializes PROJECT
initialize-synonym	Initializes SYNONYM
initialize-type	Initializes TYPE
initialize-version	Initializes VERSION

Object Class: PROJECT

Subclasses

ENTRY

Instance Variables

NAME

Name of project

REQUIREMENTS

Assoc-list of requirements names and pointers to requirements instance(s)

DESIGN

Assoc-list of design names and pointers to design instance(s)

IMPLEMENTATION

Assoc-list of implementation names and pointers to implementation instance(s)

Methods

create

Creates a PROJECT instance

initialize

Initializes all of PROJECT's instance variables

reinitialize (DUMMY)

Initializes all of PROJECT's instance variables except NAME

present (ATTRIBUTE-LIST)

Displays the instance variables listed in ATTRIBUTE-LIST of a PROJECT instance; if ATTRIBUTE-LIST is empty, displays all instance variables

save

Saves a PROJECT instance and all of its sub-instances to STREAM

create-design (NAME)

Creates a DESIGN instance whose name is NAME and adds it to DESIGN

initialize-design

Initializes DESIGN

design

Returns the value of DESIGN

design-instance

Returns a pointer to the DESIGN instance

set-design (VAL)

Sets the value of DESIGN to VAL

is-design (VAL)

Returns a pointer to the appropriate instance if VAL is a DESIGN, otherwise returns NIL

create-implementation (NAME)	Creates an IMPLEMENTATION instance whose name is NAME and adds it to IMPLEMENTATION
initialize- implementation implementation	Initializes IMPLEMENTATION Returns the value of IMPLEMENTATION
implementation- instance	Returns a pointer to the IMPLEMENTATION instance
set-implementation (VAL)	Sets the value of IMPLEMENTATION to VAL
is-implementation (VAL)	Returns a pointer to the appropriate instance if VAL is an IMPLEMENTATION, otherwise returns NIL
initialize-name name set-name (VAL)	Initializes NAME Returns the value of NAME Sets the value of NAME to VAL
create-requirements (NAME)	Creates a REQUIREMENTS instance whose name is NAME and adds it to REQUIREMENTS
initialize- requirements requirements requirements-instance	Initializes REQUIREMENTS Returns the value of REQUIREMENTS Returns a pointer to the REQUIREMENTS instance
set-requirements (VAL)	Sets the value of REQUIREMENTS to VAL
is-requirements (VAL)	Returns a pointer to the appropriate instance if VAL is an REQUIREMENTS, otherwise returns NIL

Object Class: REQUIREMENTS

Subclasses

HEADER

ENTRY

Instance Variables

TOP-LEVEL-ACTIVITY

Name of the highest level activity

ACTIVITIES

Assoc-list of activity names and pointers to activity instances

DATA-ELEMENTS

Assoc-list of data element names and pointers to data element instances

ALIASES

Assoc-list of alias names and pointers to alias instances

Methods

create

Creates a REQUIREMENTS instance
Initializes all of REQUIREMENTS' instance variables

initialize

present (ATTRIBUTE-LIST)

Displays the instance variables listed in ATTRIBUTE-LIST of a REQUIREMENTS instance; if ATTRIBUTE-LIST is empty, displays all instance variables

reinitialize (DUMMY)

Initializes all of REQUIREMENTS' instance variables except NAME, TYPE, and PROJECT

save (PROJECT, STREAM)

Saves a REQUIREMENTS instance and all of its sub-instances to STREAM

AD-A164 026

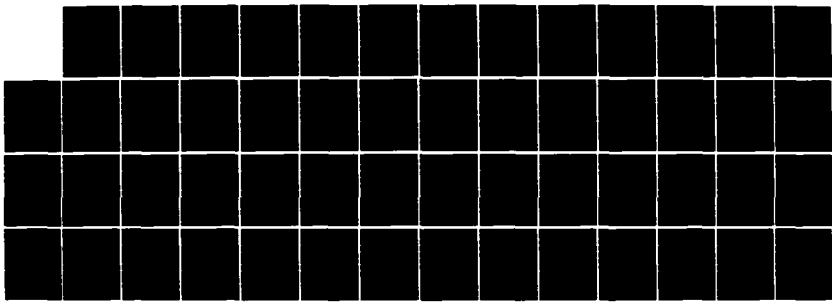
A NATURAL LANGUAGE PROCESSOR AND ITS APPLICATION TO A
DATA DICTIONARY SYSTEM(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. S A WOLFE
DEC 85 AFIT/GCS/ENG/85D-19

4/4

UNCLASSIFIED

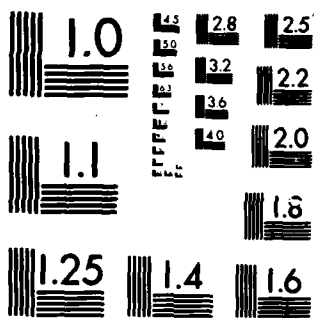
F/G 9/2

NL

A large grid of 13 columns and 5 rows of blacked-out cells, likely representing redacted data or a table structure.

END

FORM
1-75



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Object Class: SLOT

Subclasses

none

Instance Variables

VALUE	Pointer to low-level value instance
REQUIRED	Boolean - "YES" if entry required "NO" otherwise
SET	Boolean - "YES" if value has been set "NO" otherwise
PRESENT-LABEL	Label to be displayed for present operation

Methods

create	Creates a SLOT instance
initialize	Initializes all of SLOT's instance variables
present	Displays a SLOT instance
present-label	Returns the value of PRESENT-LABEL
set-present-label (VAL)	Sets the value of PRESENT-LABEL to VAL
required	Returns the value of REQUIRED
set-required (VAL)	Sets the value of REQUIRED to VAL
set	Returns the value of SET
set-set (VAL)	Sets the value of SET to VAL
value	Returns the value of VALUE
set-value (VAL)	Sets the value of VALUE to VAL

Object Class: TEXT

Subclasses

none

Instance Variables

VALUE

Methods

initialize

Initializes all of TEXT's
instance variables

present

Displays a TEXT instance

value

Returns the value of VALUE

set-value (VAL)

Sets the value of VALUE to VAL

Object Class: VARIABLE

Subclasses
PARAMETER

Instance Variables
none

Appendix F
Interface Object Definition

Appendix F
Table of Contents

	Page
Introduction	F- 3
Interface Object Definitions	F- 4

Introduction

This appendix defines the object-oriented structure of the Data Dictionary System's (DDS) natural language front-end to Data Dictionary Tool (DDT) Interface. The Interface's EVENT object class is specified in terms of its included subclasses, its instance variables, and its associated access methods.

Object Class: EVENT

Subclasses

none

Instance Variables

ACTION

Action to perform

ACTION-TYPE

Type of action - either NIL if an action upon a data dictionary object of META if not

OBJECT

Data dictionary object on which to perform the action specified by ACTION

ATTRIBUTES

A list of instance variables of OBJECT on which to perform the action specified by ACTION

Methods

execute

Interprets an EVENT instance

initialize

Initializes all of EVENT's instance variables

present

Displays all of EVENT's instance variables

initialize-action

Initializes ACTION

initialize-action-type

Initializes ACTION-TYPE

initialize-attributes

Initializes ATTRIBUTES

add-attribute (LIST)

For each attribute in LIST: Verifies not already a member of ATTRIBUTES; If not, adds to ATTRIBUTES

initialize-object

Initializes OBJECT

Appendix G

Test Plan

Appendix G
Table of Contents

	Page
Introduction	G- 3
System-Level Tests	G- 3
Grammar Constructor Tests	G- 4
Sentence Interpreter Tests	G- 6
Data Dictionary Access Tests	G- 7
Grammar Tests	G-10
Data Dictionary Content Tests	G-11
SDW Interface Tests	G-11
Conclusion	G-12

Introduction

This appendix specifies the test plan to be used in formally testing the DDS data dictionary system including the CoIn natural language processor. This plan consists of a series of functional tests which specify what is being tested, how the test should be performed (at a high level), and what the results of the test should be. This test plan is organized to follow the System Requirements section of the requirements chapter of this thesis, Chapter II.

System-Level Tests

1. This test is to determine if DDS is able to accept a valid input sentence from the user, interpret the sentence as a command, and retrieve and/or modify the requested information from the data dictionary. The test should be performed by entering several valid requests into the system. Any information requested to be displayed should be displayed. Any information requested to be modified should be modified.
2. This test is to determine if DDS is able to reject an invalid input sentence from the user. The test should be performed by entering several invalid requests into the system. Each of the requests should cause an error message to be displayed.

Grammar Constructor Tests

1. This test is to determine if the grammar constructor is able to save a grammar from the working grammar file to a permanent grammar file. The test should be performed by entering a valid request to save a working grammar. The grammar should be saved to a permanent file.
2. This test is to determine if the grammar constructor is able to load a grammar from a permanent grammar file to the working grammar file. The test should be performed by entering a valid request to load a permanent grammar. The grammar should be loaded to the working grammar file.
3. This test is to determine if the grammar constructor is able to initialize the working grammar. The test should be performed by first loading the working grammar from a permanent grammar and then entering a valid request to initialize the working grammar. The working grammar should be initialized.
4. This test is to determine if the grammar constructor is able to create production rules. The test should be performed by entering a valid request to create a new production rule. The production rule should be created.
5. This test is to determine if the grammar constructor is able to display the existing production rules. The test should be performed by entering a valid request to display a

production rule. The requested production rule should be displayed.

6. This test is to determine if the grammar constructor is able to display a list of the existing production rules. The test should be performed by entering a valid request to display a list of the existing production rules. A list of the existing production rules should be displayed.

7. This test is to determine if the grammar constructor is able to modify an existing production rule. The test should be performed by entering a series of valid requests to modify production rules. At least one entry should be made for each defined modification command. The requested modifications should be made to the proper production rules.

8. This test is to determine if the grammar constructor is able to delete an existing production rule. The test should be performed by entering a valid request to delete a production rule. The specified production rule should be deleted.

9. This test is to determine if the grammar constructor is able to reject invalid construction requests. The test should be performed by entering a series of invalid requests. At least one entry should be made for each of the defined commands of the the constructor. Each of the requests should cause an error message to be displayed.

Sentence Interpreter Tests

1. This test is to determine if the sentence interpreter is able to accept input sentences that are valid within the defined grammar. The test should be performed by entering a series of valid sentences. At least one entry should be made which exercises each of the defined grammar productions. The interpreter should correctly parse each entry.

2. This test is to determine if the sentence interpreter is able to reject input sentences that are invalid within the defined grammar. The test should be performed by entering a series of invalid sentences. At least one sentence should be entered for each of the following cases: a sentence whose first word is not valid within the grammar, a sentence whose last word is not valid within the grammar, a sentence which contains a word that is not valid within the grammar and in which the word is neither the first nor the last word of the sentence, a sentence which is correct within the grammar except that it contains at least one extra word at the end, and a sentence that is correct within the grammar except that it ends before the defined end-of-sentence. The interpreter should reject each entry and display an error message. The error message should attempt to show where in the input sentence the error occurred. The error message should offer suggestions as to how to correct the problem.

3. This test is to determine if the sentence interpreter is able to syntactically translate the results of a valid sentence parse into the proper execution commands of the application tool. The test should be performed by entering a series of valid sentences should generate all (if this is reasonable) or several (if it is not reasonable to generate all) of the execution commands of the application tool. For each input sentence, the proper application commands should be generated.

Data Dictionary Access Tests

1. This test is to determine if the data dictionary access process is able to access all of the information stored in the data dictionary file. The test should be performed by entering a series of valid requests to access the data dictionary. As a minimum, an entry should be made which accesses an instance of each object type in the data dictionary. Also as a minimum, an entry should be made which causes each access type (add, delete, display, etc.) to be performed. For each entry, the proper access to the information stored in the data dictionary should be done.
2. This test is to determine if the data dictionary access process is able to maintain the "consistency" of the data in the data dictionary. The test should be performed by entering a series of valid requests which cause each of the

defined consistency routines to be invoked. For each entry, the data dictionary access process should maintain the consistency of the information stored in the data dictionary.

3. This test is to determine if the data dictionary access process is able to add new objects to the data dictionary. The test should be performed by entering a series of valid requests to add new object instances. As a minimum, an entry should be made which adds a new instance of each defined object type. The specified instances should be added to the data dictionary.

4. This test is to determine if the data dictionary access process is able to modify existing information in the data dictionary. The test should be performed by entering a series of valid requests to modify existing object instances. An entry should be made which exercises each defined modification function. For each entry, the requested modification should be made.

5. This test is to determine if the data dictionary access process is able to reinitialize existing information in the data dictionary. The test should be performed by entering a series of valid requests to reinitialize existing object instance. As a minimum, an entry should be made which reinitializes an instance of each defined object type. The

specified instances should be reinitialized to their respective initial states.

6. This test is to determine if the data dictionary access process is able to delete existing information in the data dictionary. The test should be performed by entering a series of valid requests to delete existing object instances. As a minimum, an entry should be made which deletes an instance of each defined object type. The specified instances should be deleted from the data dictionary.

7. This test is to determine if the data dictionary access process is able to reject modification requests which reference information not contained in the data dictionary. The test should be performed by entering a series of modification requests which reference information not contained in the data dictionary but which are otherwise valid. An entry should be made which exercises each defined modification function. For each request, the data dictionary access process should respond with an error message.

8. This test is to determine if the data dictionary access process is able to display information stored in the data dictionary. The test should be performed by entering a series of valid presentation requests. As a minimum, an entry should be made which displays an instance of each

define object type. The specified instances should be displayed.

9. This test is to determine if the data dictionary access process is able to reject presentation requests which reference information not contained in the data dictionary. The test should be performed by entering a series of presentation requests which reference information not contained in the data dictionary but which are otherwise valid. For each request, the data dictionary access process should respond with an error message.

10. This test is to determine if the data dictionary access process is independent of the natural language processor. This test should be performed by inspecting the source code listings of the data dictionary to verify that it does not reference any of the modules or data local to the natural language processor. No references to the modules or data of the natural language processor should be found.

Grammar Tests

1. This test is to determine if the defined grammar of DDS is functionally complete. This test should be performed by entering a series of valid requests which invoke each of the operations provided by the data dictionary access process. Requests should be made which access instance of each of the

defined object types. The specified operation should be performed on the specified object instance.

2. This test is to determine if the defined grammar of DDS is sufficiently flexible. No test is specified to validate this requirement. The entries of users of varying expertise should be monitored to determine flexibility shortcomings.

Data Dictionary Content Tests

1. This test is to determine if the data dictionary contents of DDS is consistent with the requirements specified in Development Documentation Guidelines and Standards (AFIT/ENG, 1984). This test should be performed by inspecting the source code listings to verify that they are consistent with this document. No inconsistencies should be found.

SDW Interface Tests

1. This test is to determine if the data dictionary system is interfaced into the Software Development Workbench (SDW). This test should be performed by executing the SDW and entering the proper codes to invoke DDS. The SDW should execute DDS.

2. This test is to determine if the natural language processor is interfaced into the SDW. This test should be performed by executing the SDW and entering the proper codes

to invoke the natural language processor. The SDW should execute the natural language processor.

Conclusion

This test plan has specified a suite of tests that validate the requirements of DDS. As the requirements change, this document should be updated to remain consistent with them.

Appendix H
CoIn User Manual

CoIn User Manual

Version 1.0

Capt Stephen A. Wolfe, USAF

AFIT/GCS-85D

December 1985

Table of Contents

	Page
List of Figures	H- 4
I. Introduction	H- 5
II. Scope	H- 6
III. Using the Grammar Constructor	H- 7
Getting Started	H- 7
Creating a Grammar	H- 8
Modifying a Grammar	H- 8
Exiting	H-14
IV. Using the Sentence Interpreter	H-15
V. A Short Example	H-16
Grammar Productions	H-16
Constructor Commands	H-16
Sample Interpreter Inputs	H-17
VI. Conclusion	H-18
Bibliography	H-19

List of Figures

Figure		Page
1	Production Record Fields	H- 9
2	Function Record Fields	H-10

I. Introduction

CoIn is a natural language processing system. It is implemented on the AFIT Information Systems Laboratory VAX-11/780 computer. CoIn is a part of AFIT's Software Development Workbench (SDW). CoIn is written in NIL Lisp and runs within the NIL interpreter environment.

CoIn consists of two subsystems: a grammar constructor and a sentence interpreter. The grammar constructor allows you to define a grammar consisting of production records and function records (these are described in Chapter III). Using a grammar built with the constructor, the sentence interpreter is able to parse input sentences and execute Lisp code that you have built into the grammar.

II. Scope

This manual describes the use of the CoIn natural language processing system. It is assumed that the reader has a minimal working knowledge of the VMS operating system (see DEC, 1984, for more information), the SDW (Hadfield, 1982), and the NIL interpreter (Burke, et al, 1984).

III. Using the Grammar Constructor

This chapter describes how to use the grammar constructor subsystem of CoIn. This chapter is divided into four sections: Getting Started, which explains how to execute CoIn; Creating a Grammar, which explains how to create a new grammar and how to permanently store it when you're done; Modifying a Grammar, which explains how to create new production and function records and how to alter and delete existing production and function records; and Exiting which explains how to exit from CoIn to the SDW.

Getting Started

CoIn is integrated into the SDW. This makes it very easy to execute. First log into the SDW account, enter the SDW, and move to your project directory. Return to the top-level menu and enter 'ED' (the Text Editors functional group) at the prompt. The EDITOR (ED) MENU will be displayed. When prompted, enter 'NL' (the CoIn Natural Language Grammar Editor). NIL will be invoked, and CoIn will be loaded. When NIL is done loading, the message "*** CoIn Loaded ***" will be displayed. At that time, CoIn is ready for input.

Creating a Grammar

Most of the time you will probably not want to create a grammar from scratch. You'll probably just want to add to or modify an existing grammar. However, it is possible to create a new grammar. To do so, first invoke CoIn as described in the previous section, then type "(initialize-grammar)". Be careful! If you have loaded a grammar, modified it, and not saved it, this command will destroy all of your changes!

After initializing the grammar, you can then proceed to create your grammar by using the commands as described in the next section, Modifying a Grammar. When your grammar is complete (or anytime you want to save a partial definition), you can type "(save-grammar)" to store it on a disk file. The grammar will be saved in a file called GRAMMAR.LSP in the current NIL working directory (your default directory unless you have told NIL otherwise).

Modifying a Grammar

A grammar consists of a set of production records and function records. Production records are used to store the productions of a grammar and consist of three fields: NAME, SUB-PRODUCTION LIST, and CODE (see Figure 1). The NAME field contains the element which the current word of the input sentence must match for a parse to proceed. The

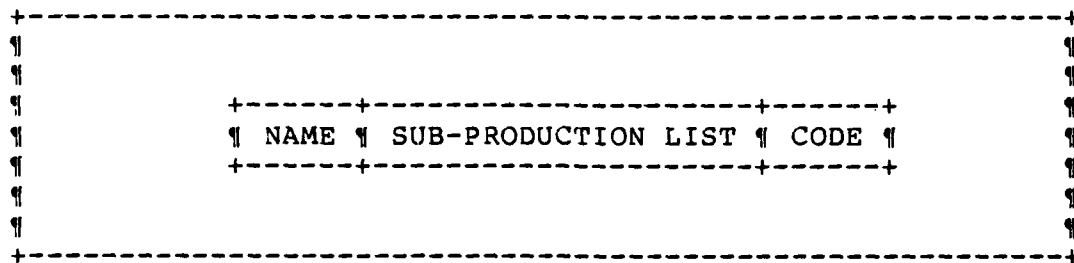


Figure 1. Production Record Fields

SUB-PRODUCTION LIST field contains a list of possible sub-productions which can be used to continue the parse should the current word parse successfully. The CODE field contains a piece of Lisp code which is executed if the production it is associated with is successfully traversed. A production must end with an end-of-sentence marker (\$) if it is at the top level of the grammar. Otherwise, it must end with an end-of-production marker (*).

Function records are used to store "functions". A function is equivalent to a Lisp predicate function. The function is applied to the current word of an input sentence and, if the word is part of the function's domain, non-NIL is returned. Otherwise, NIL is returned. A function record consists of two fields: NAME and CODE (see Figure 2). The NAME field contains the name of the function, and the CODE field contains the Lisp code to be applied.

The following subsections describe how to create a new production record, create a new function record, add a

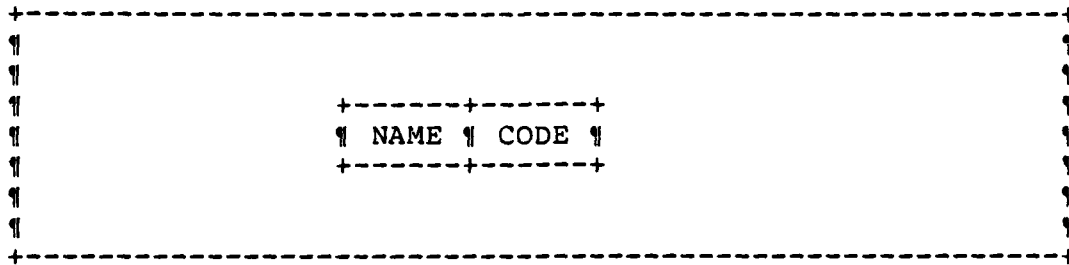


Figure 2. Function Record Fields

grammar production to an existing production record, modify the CODE field of an existing function record, destroy an existing production record, destroy an existing function record, and do other miscellaneous modifications.

Creating a Production Record. To create a new production record, type "(create-production PRODUCTION-NAME (LISP-CODE))" where PRODUCTION-NAME is replaced by the name of the production record you wish to create and LISP-CODE is replaced by the Lisp code that you want executed when the production record is successfully used. For instance, if you want to create a production record called <display-message-1> that displays the message "message 1" when it is traversed, then you should enter:

```
(create-production <display-message-1>
  (msg "message 1"))
```

If the production record <display-message-1> already exists, an error message is displayed and the existing production

record is left as is. The LISP-CODE field is optional.

Creating a Function Record. To create a new function record, type "(create-function FUNCTION-NAME (LISP-CODE))" where FUNCTION-NAME is replaced by the name of the function record you wish to create and LISP-CODE is replaced by the Lisp code that defines the function. For instance, if you want to create a function record called <number> that is defined using the Lisp "numberp" predicate, then you should enter:

```
(create-function <number>
  (lambda (x) (numberp x)))
```

The LISP-CODE field is optional but if left out, must subsequently be modified before using the grammar. If this is not done, the sentence interpreter will crash when it tries to use the function record. As is true for production records, an error message is displayed if you try to create a function record using a name that already exists.

Adding a Production. To add a grammar production to an existing production record, type "(add-production (PRODUCTION) (LISP-CODE) PRODUCTION-NAME)" where PRODUCTION is replaced by the grammar production to be added, LISP-CODE is replaced by the Lisp code you want executed when the production is successfully traversed, and PRODUCTION-NAME is

replaced by the name of the production record to which you want the production added. For example:

```
(add-production (what is *) () <display>)
```

adds the production "(what is *)" to the existing production record <display>. As is shown, the LISP-CODE parameter may be empty. However it is not optional. The parentheses must be included. If the <display> production record does not exist, an error message is displayed.

Note the use of the end-of-production marker (*). All productions that are not at the top level (that is, they are not part of your highest level production record) must end with an end-of-production marker. All productions at the top level must end with an end-of-sentence marker (\$).

Modifying a Function. To modify the CODE field of an existing function record, type "(modify-function (LISP-CODE) FUNCTION-NAME)" where LISP-CODE is replaced by the new Lisp code that defines the function and FUNCTION-NAME is replaced by the function to be modified. For example:

```
(modify-function (lambda (x) (null x)) <empty-list>)
```

modifies the CODE field of the function record <empty-list> to be the Lisp expression (lambda (x) (null x)). If the specified function record does not exist, an error message is displayed. The LISP-CODE may be empty, but, as is true

for the add-production operation, the parentheses must be included.

Destroying a Production Record. To delete an existing production record from a grammar, type "(destroy-production PRODUCTION-NAME)" where PRODUCTION-NAME is replaced by the name of the production record to destroy. For instance, to destroy the production record <display>, enter:

```
(destroy-production <display>)
```

As usual, if the specified production record does not exist, an error message is displayed.

Destroying a Function. To delete an existing function record from a grammar, type "(destroy-function FUNCTION-NAME)" where FUNCTION-NAME is replaced by the name of the FUNCTION to destroy. For example:

```
(destroy-function <number>)
```

deletes the <number> function record. Again, an error message is displayed if the function record does not exist.

Other Operations. In addition to the operations described above, a grammar can be modified by using a Lisp s-expression editor that is included in CoIn. The editor is a slightly modified version of a Lisp function editor described in Artificial Intelligence Programming by Charniak, Riesbeck, and McDermott (Charniak, et al, 1980).

To specify a production or function record to edit, type "(sedit STRUCTURE-NAME)" where STRUCTURE-NAME is replaced by the name of the production or function record that you wish to edit. Once you have entered this command, the editor is used exactly as described in Chapter 7 of Artificial Intelligence Programming.

Exiting

To exit from NIL and return to the SDW, type "(quit)". This will return you to the Text Editors functional group of the SDW.

IV. Using the Sentence Interpreter

This chapter describes how to use the sentence interpreter subsystem of CoIn. This chapter has only one section, Interpreting A Sentence, which explains how to enter a request to interpret a sentence. Invoking and exiting the system is the same as is described in Chapter 3.

Interpreting a Sentence

To invoke the interpreter to parse a sentence, type "(parse '(SENTENCE) TOP-LEVEL-PRODUCTION)" where SENTENCE is replaced by the sentence to be parsed and TOP-LEVEL-PRODUCTION is replaced by the name of your top-level production record. For instance:

```
(parse '(show me module M1) <grammar>)
```

tells the interpreter to parse the sentence "show me module M1" using the grammar specified by the top-level production record <grammar>. If your input sentence is not valid within the specified grammar, an error message is displayed.

V. A Short Example

This chapter provides a short, but one hopes useful, example of the use of CoIn. First a grammar is defined, then the constructor commands to implement it are shown. Finally, several sample interpreter commands are included. The defined grammar accepts any sentence consisting of a string of one or more occurrences of the word "a".

Grammar Productions

```
<grammar> -> <a>
<a>       -> a
<a>       -> a <a>
```

Constructor Commands

```
(initialize-grammar)
(create-production <grammar> (msg "Parse complete" N))
(add-production (<a> $) () <grammar>)
(create-production <a> ())
(add-production (a *) (msg "Using (a *)" N) <a>)
(add-production (a <a> *) (msg "Using (a <a>)" N) <a>)
```

Sample Interpreter Inputs

(parse '(a) <grammar>)

(parse '(a a) <grammar>)

(parse '(a a a a a) <grammar>)

VI. Conclusion

This manual has described how to use the CoIn natural language processor. For an extended example of the use of CoIn, see (Wolfe, 1985), which describes the implementation of a natural language human-computer interface for a data dictionary system.

Bibliography

Burke, Glenn S., et al. NIL Reference Manual. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, 1984.

Charniak, Eugene, et al. Artificial Intelligence Programming. Hillsdale NJ: Lawrence Erlbaum Associates, Publishers, 1980.

DEC. Introduction to VAX/VMS. Maynard MA: Digital Equipment Corporation, 1984.

Hadfield, 2Lt Steven M. An Interactive and Automated Software Development Environment. MS Thesis, AFIT/GCS/EE/82D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.

Wolfe, Capt Stephen A. A Natural Language Processor and its Application to a Data Dictionary System. MS Thesis, AFIT/GCS/ENG/85D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.

Appendix I
DDS User Manual

DDS User Manual

Version 1.0

Capt Stephen A. Wolfe, USAF

AFIT/GCS-85D

December 1985

Table of Contents

	Page
I. Introduction	I- 4
II. Scope	I- 5
III. Using DDS	I- 6
IV. Conclusion	I- 9
Bibliography	I-10

I. Introduction

DDS is a data dictionary system which employs a natural language human-computer interface. To interact with DDS, one enters English sentences which specify to DDS the actions it is to perform. DDS is implemented on the AFIT Information Systems Laboratory VAX-11/780 computer. It is a part of AFIT's Software Development Workbench (SDW). DDS is written in NIL Lisp and runs within the NIL interpreter environment.

II. Scope

This manual describes the use of the DDS data dictionary system. It is assumed that the reader has a minimal working knowledge of the VMS operating system (see DEC, 1984, for more information), the SDW (Hadfield, 1982), and the NIL interpreter (Burke, et al, 1984).

III. Using DDS

This chapter describes how to use DDS. This chapter is divided into 3 sections: Getting Started, which explains how to execute DDS; Using the System, which explains how to enter commands to the system; and Exiting which explains how to exit from DDS to the SDW.

Getting Started

DDS is integrated into the SDW. This makes it very easy to execute. First log into the SDW account, enter the SDW, and move to your project directory. Return to the top-level menu and enter 'DS' (the Design Tools functional group) at the prompt. The DESIGN TOOL (DS) MENU will be displayed. When prompted, enter 'DD' (the DDS data dictionary system). NIL will be invoked, and DDS will be loaded. When DDS is done loading, the message "*** Data Dictionary System (DDS) Loaded ***" will be displayed. To execute DDS, type "(dds)" (without the quotes). DDS will be invoked and will display the message "Data Dictionary System (DDS)" followed by a prompt "DDS->". At that time, DDS is ready for input.

Using the System

Currently the capabilities of DDS are very limited. The only data manipulation operations that are implemented are the initialization and presentation functions. Also implemented are the clear screen, help (again limited), and exit meta-functions. The best source for determining what the current system can and cannot do is Appendix D (Implemented Grammar) of A Natural Language Processor and its Application to a Data Dictionary System (Wolfe, 1985). This chapter presents examples of valid input sentences for the current grammar. Please note that the input sentences are enclosed within parentheses, and that they contain no punctuation. This format must be used, or NIL will complain unmercifully.

Clearing the Screen. The following examples show how to clear the terminal screen in DDS:

(please clear the screen)

(cls)

Displaying the Help Message. The following example shows how to display the help message in DDS:

(help)

Exiting from DDS. The following examples show how to exit from DDS:

(quit)

(exit)

Initializing a Data Dictionary Entry. Before an entry can be initialized, it must exist. Currently, the only way to create an entry is to modify your data base (the file DB.LSP) using a text editor. In the following examples, it is assumed that P1 is a defined process and PARAM is a defined parameter.

(initialize process P1)

(please init parameter PARAM)

Displaying a Data Dictionary Entry. Before an entry can be displayed, it must exist. See the discussion in the previous subsection concerning creation of an entry. In the following examples, it is again assumed that P1 is a defined process and PARAM is a defined parameter.

(please show me process P1)

(what are the processes called by process P1)

(display the name type and version of parameter PARAM)

Exiting

To exit from DDS and return to NIL, type "(quit)". To exit from NIL and return to the SDW, type "(quit)" again. This will return you to the Design Tools functional group of the SDW.

IV. Conclusion

This manual has described how to use the DDS data dictionary system. For a more complete description of DDS, see (Wolfe, 1985).

Bibliography

Burke, Glenn S., et al. NIL Reference Manual. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, 1984.

DEC. Introduction to VAX/VMS. Maynard MA: Digital Equipment Corporation, 1984.

Hadfield, 2Lt Steven M. An Interactive and Automated Software Development Environment. MS Thesis, AFIT/GCS/EE/82D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.

Wolfe, Capt Stephen A. A Natural Language Processor and its Application to a Data Dictionary System. MS Thesis, AFIT/GCS/ENG/85D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.

Bibliography

BIB-1

Bibliography

- AFIT/ENG. AFIT/ENG Development Documentation Guidelines and Standards, Draft #2. Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1984.
- Aho, Alfred V., et al. The Design and Analysis of Computer Algorithms. Reading MA: Addison-Wesley Publishing Company, 1974.
- Barr, Avron and Edward A. Feigenbaum. The Handbook of Artificial Intelligence, Volume I. Los Altos CA: William Kaufmann, Inc., 1981.
- . The Handbook of Artificial Intelligence, Volume II. Los Altos CA: William Kaufmann, Inc., 1982.
- Bobrow, Daniel G., et al. "GUS, A Frame-Driven Dialog System," Artificial Intelligence, 8: 155-173 (April 1977).
- Burke, Glenn S., et al. NIL Reference Manual. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, 1984.
- Charniak, Eugene, et al. Artificial Intelligence Programming. Hillsdale NJ: Lawrence Erlbaum Associates, Publishers, 1980.
- Davis, Richard M. Thesis Projects in Science and Engineering. New York: St. Martin's Press, 1980.
- DeMarco, Tom. Structured Analysis and System Specification. New York: Yourdon Press, 1979.
- Goldberg, Adele and David Robson. Smalltalk-80, The Language and Its Implementation. Reading MA: Addison-Wesley Publishing Company, 1983.
- Goldberg, Adele. Smalltalk-80, The Interactive Programming Environment. Reading MA: Addison-Wesley Publishing Company, 1984.
- Hadfield, 2Lt Steven M. and Gary B. Lamont. "The Software Development Workbench: An Integrated Software Development Environment," Proceedings of the Digital equipment Computer User Society. 171-177. 1983.

- Hadfield, 2Lt Steven M. An Interactive and Automated Software Development Environment. MS Thesis, AFIT/GCS/EE/82D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.
- Harris, L. R. "User Oriented Data Base Query with the ROBOT Natural Language Query System," International Journal of Man-Machine Studies, 9: 697-713 (November 1977).
- Hayes-Roth, et al. Building Expert Systems. Reading MA: Addison-Wesley Publishing Company, 1983.
- Hendrix, Gary G. "Human Engineering for Applied Natural Language Processing," 5th International Joint Conference on Artificial Intelligence - 1977, Proceedings of the Conference, Volume One. 183-191. Department of Computer Science, Carnegie-Mellon University, Pittsburg PA, 1977.
- Hendrix, Gary G., et al. "Developing a Natural Language Interface to Complex Data," ACM Transactions on Database Systems, 3: 105-147 (June 1978).
- Horowitz, Ellis and Sartaj Sahni. Fundamentals of Data Structures in Pascal. Rockville MD: Computer Science Press, 1984.
- Kowalski, Robert. "AI and Software Engineering," Datamation, 30: 92-102 (Nov 1, 1984).
- Mihaloew, Reed A. SYSFL, A Systems Flowcharting Routine Using Interactive Graphics. Aeronautical Systems Division Computer Center, Air Force Systems Command, Wright-Patterson AFB OH, undated.
- Myers, Glenford J. Reliable Software Through Composite Design. New York: Van Nostrand Reinhold Company, 1975.
- Peters, Lawrence J. Software Design: Methods and Techniques. New York: Yourdon Press, 1981.
- Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill Book Company, 1983.
- . "Natural-Language Interfaces," Computer, 17: 39-47 (September 1984).

- Schank, Roger C. and Christopher K. Riesbeck. Inside Computer Understanding. Hillsdale NJ: Lawrence Erlbaum Associates, Publishers, 1981.
- Steele, Guy L., Jr. Common LISP. Burlington MA: Digital Press, 1984.
- Thomas, Capt Charles W. An Automated/Interactive Software Engineering Tool to Generate Data Dictionaries. MS Thesis, AFIT/GCS/ENG/84D-29. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984.
- Waltz, David L. "An English Language Question Answering System for a Large Relational Database," Communications of the ACM, 21: 526-539 (July 1978).
- Winston, Patrick Henry. Artificial Intelligence. Reading MA: Addison-Wesley Publishing Company, 1984.
- Wirth, Niklaus. Algorithms + Data Structures = Programs. Englewood Cliffs NJ: Prentice-Hall, Inc., 1976.
- Wolfe, Capt Stephen A. George: A Tool for Building and Parsing Semantic Grammars. Unpublished report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1985a.
- . Course Project: A Software Design Tool. Unpublished report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1985b.
- Woods, W. A. "Transition Network Grammars for Natural Language Analysis," Communications of the ACM, 13: 591-606 (October 1970).

Vita

VIT-1

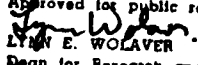
VITA

Captain Stephen A. Wolfe was born on 23 February 1956 in Portland, Oregon. He graduated from high school in Spring Valley, California, and attended San Diego State University from which he received the degree of Bachelor of Arts in Computer Science in May 1979. He graduated with high honors and with distinction in the major. After graduating, he received a commission in the United States Air Force through the Officer Training School program. His first assignment was at Space Division in Los Angeles, California, where he was a systems analyst and applications programmer. In April 1982, he was transferred to Washington, DC, where he became the lead software engineer for the procurement of the Ground Launched Cruise Missile Weapons Control System. He entered the School of Engineering, Air Force Institute of Technology, in May 1984.

Permanent address: 1221 Purdy Street
Spring Valley, CA 92077

AD-A164026

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1d. RESTRICTIVE MARKINGS	
2. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/85D-19		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/EN	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright Patterson AFB, OH 45433		7b. ADDRESS (City, State and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Ctr	8b. OFFICE SYMBOL (If applicable) RADDC/COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code) Griffiss AFB, NY		10. SOURCE OF FUNDING NOS.	
11. TITLE (Include Security Classification) See Box 19		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT NO.
12. PERSONAL AUTHOR(S) Stephen A. Wolfe, A.B., Capt, USAF			
13a. TYPE OF REPORT MS Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1985 December	15. PAGE COUNT 343
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB. GR.	
09	02		
		Natural Language Data Dictionary	
		Automated Tools Software Development	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)			
Title: A NATURAL LANGUAGE PROCESSOR AND ITS APPLICATION TO A DATA DICTIONARY SYSTEM			
Thesis Chairman: Dr. Gary B. Lamont			
<p style="text-align: right;">Approved for public release: IAW AFR 180-1.  LYNN E. WOLAVER 16 JAN 86 Dean for Research and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433</p>			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Gary B. Lamont		22b. TELEPHONE NUMBER (Include Area Code) 255-3450	22c. OFFICE SYMBOL AFIT/ENG

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

The development of a human-computer interface construction and interpretation tool capable of processing English-like of natural language user input is discussed. The utility of the tool is demonstrated by using it to create the natural language interface for a data dictionary system. The data dictionary's development is also documented and is used as the overall context for the presentation.

SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

386

DTIC