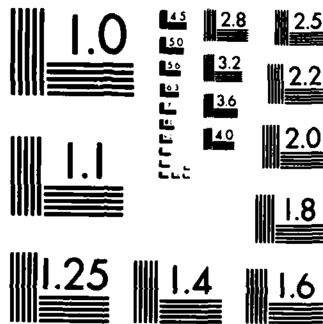
AD-A152 105 A D TO C INTERPRETER(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING K J SHOMPER DEC 84 AFIT/GCS/ENG/84D-27 1/2

UNCLASSIFIED F/G 9/2 NL

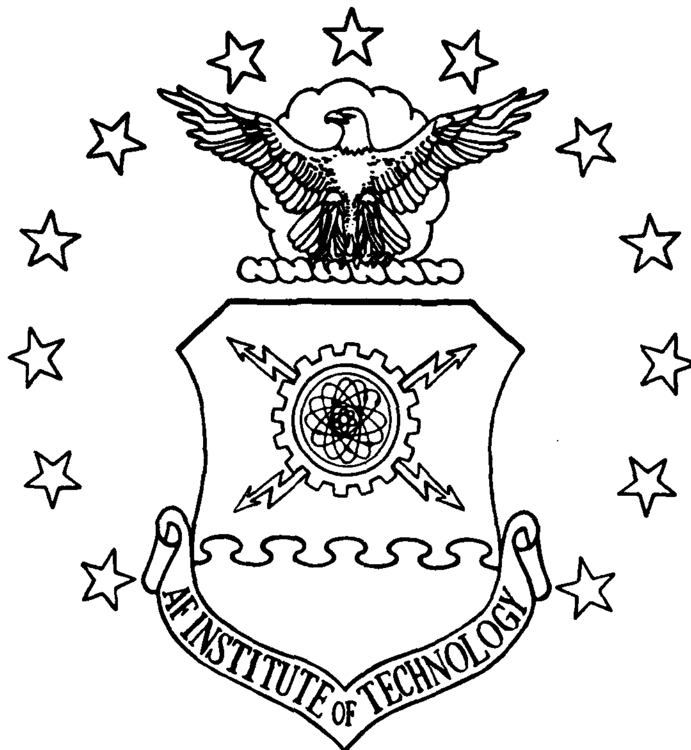


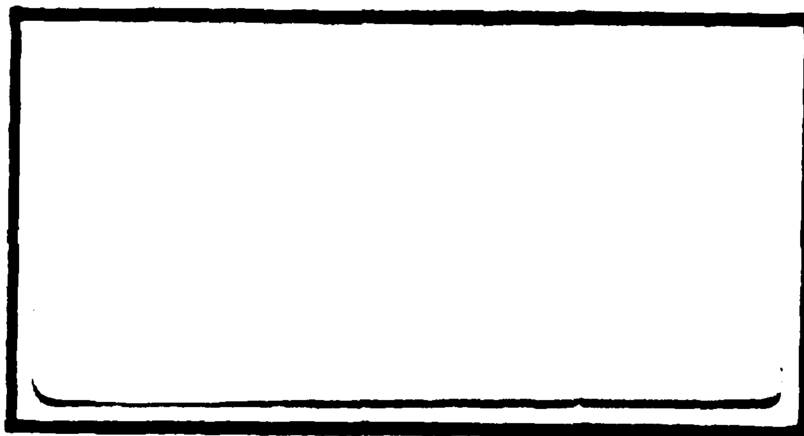
MICROCOPY RESOLUTION TEST CHART

DARDS-1963-A

REPRODUCED AT GOVERNMENT EXPENSE

AD-A152 105





DTIC FILE COPY

This document has been approved
f r public release and sale; its
... ti n is unlimited.

DTIC
ELECTE
APR 5 1985
S D
A

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

85 03 13 180

AFIT/GCS/ENG/84

A D TO C

INTERPRETER

THESIS

Kevin J. Shomper
Second Lieutenant, USAF

AFIT/GCS/ENG/84D-27

DTIC
ELECTE
APR 5 1985
D
A

Approved for public release; distrubution unlimited

AFIT/GCS/ENG/84D-27

A D TO C
INTERPRETER

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of
Master of Science in Electrical Engineering

Kevin J. Shomper, B.A.
Second Lieutenant, USAF

A1

December 1984

Approved for public release; distribution unlimited

Preface

The purpose of this report was to implement the D language, and thereby gain some insight as to its efficiency, and to a lesser degree D's usefulness. This analysis extended in part to the D machine, from which the language was drawn.

Having fallen short of a full implementation, this thesis still serves to provide a static analysis of the language design, along with further definition. Discussed is D's purpose, its history, and its current state. Each section is built upon the previous one; therefore, it is suggested those who read it start at the beginning (like a good book). With some minor modifications to D's weaknesses, D has the potential to become a widely used language.

This thesis would not have been possible were it not for the following people to whom I wish to express my gratitude. First to Dr. Harold Carter. Except for his patience, with my procrasinating ways, and guidence, I'd have long ago gone down a road of no return. Secondly to Dr. Panna Nagarsenker, whose constant curiosity provided the impetuous to keep me working, while she supplied helpful hints. Thirdly, I wish to acknowledge the support of my wife Connie, without whose encouragement this thesis would have been infinitly greater. Finally my highest thanks goes to my Lord and Savior Jesus Christ whose divine hand placed me at AFIT, and whose grace and mercy enabled me to accomplish the tasks set before me.

Kevin J. Shomper

Table of Contents

Page

Preface ii

List of Figures v

List of Tables vi

Abstract vii

I. Introduction I-1

The Problem I-1
Background on the Problem I-3
Approach I-4
Literature Review I-5

II. Requirements Definition and Justification II-1

Why C II-2
System Constraints II-3
User Interface II-5
Unix Compatability and Program I/O II-6
Other Requirements II-7

III. The Language III-1

File Order III-1
Naming III-2
Comments III-3
Classes III-4
Operators III-5
Punctuation III-6
Expressions III-9
Statements III-11
Blocks III-11
The Header III-12
Block Linkage III-12
Lines III-12
Files III-13
Linkage III-14
State Files III-16
Storage Files III-16
Actor Files III-18
Bundle Files III-20
Input and Output III-20
Open, Creat, Close, and Unlink III-21
An Example III-22

Page

IV. The Design IV-1

The Lexical Analyzer IV-5
The Interpreter IV-7
Interpret IV-8
Linkage IV-9
The Bundle File Body IV-10
The State File Body IV-11
Scalar Definition IV-12
Structure Definition IV-12
Object Declaration IV-13
Logical Name IV-13
Object Declaration Revisited IV-13
Declarator IV-14
Expression IV-15
Storage IV-16
Actor IV-17
Function Declaration IV-18
Operator Declaration IV-19
Actor's Linkage IV-19
Block IV-19
Return and New IV-21
Error IV-21
Other Functions IV-22
Conclusion IV-25

V. Results and Analysis V-1

An Analysis of D V-1
Weaknesses V-1
Strengths V-4
Changes to D V-5
Function vs. Specification V-7

VI. Conclusion VI-1

Following on VI-1
Do It In Ada VI-1
The Copy Problem Solved VI-2
Generics in C VI-2

Appendix A: D Syntax Diagrams A-1

Appendix B: Keywords and Operators B-1

Appendix C: Design Structure Charts C-1

Appendix D: Source Code and Sample Runs D-1

Bibliography BIB-1

Vita V-1

List of Figures

Figure		Page
1.	Structure Chart Notation	IV-4
2.	Structure Chart Control Sequencing	IV-4
3.	Global Design	IV-5
4.	The Addition of Get_token	IV-5
5.	The Structure of Get_token	IV-7
6.	The Program control Structure	IV-26

List of Tables

Table Page

1. Description of Punctuation Symbols III-8

Abstract

In a continuing effort to define and analyze the D language a partial interpreter has been built. This interpreter stresses both syntactic and semantic error reporting in order to function as a learning aid to the inexperianced D programmer. The language definition is completed, with the exception of the accept function and the ? block control. All error checking has been accomplished except expression type checking.

D contains weaknesses; most notably in its class construction. Although, this is a perceived problem and not a functional one. It is reccommended that the reader be familiar with both Ada an C in order to fully grasp the ideas. Follow on work should complete the code generation, but in Ada not in C. Only then will the full potential of D be realized.

I. Introduction

Perceiving current architectures to be inadequate to the task of modern programming problems (5:1), Captain Jennings designed an architecture and a language. The purpose of this thesis is to take the designed language, D, and attempt to implement it. This implementation will take the form of an interpreter which transforms a program coded in D to one coded in C. The presentation format will be similar to the steps one would follow in software engineering: 1) present/define the problem, 2) set requirements, 3) design the system, 4) code, 5) verify through testing. In addition, a separate section analyzing the language will be included to aid the reader in understanding design decisions, as well as provide a guide to programming in the D language. This last use is added as it is hoped D will be a fully implemented language upon the completion of this thesis. Lastly a section presenting the differences between the designed version and the implemented version will be included.

The Problem

In its most rudimentary form the problem addressed is there does not currently exist a machine upon which the D language is implemented. From a software viewpoint one has basically two practical methods at his/her disposal to achieve the task. The most efficient method, apart from

keeping an experienced staff of assembly programmers on hand, is compiling. Compiling is a process of breaking apart a high-order language (HOL) program, also referred to as the inpu or source program, into lexical units called tokens. From these tokens a machine internal structure is formed, such as a tree or a table. This structure is then reduced, if possible, and an assembly program is generated from it. The specific assembly language used is the target language. Further optimization may be done at this point by means of a peephole optimizer: a routine in the compiler which examines a small section of the assembly code, usually three to five instructions at a time, and deletes, changes, or adds instructions without altering the program's meaning. The assembly instructions are then mapped one for one into machine code which comprises the machine executable, semantic interpretation of the HOL program.

..cept in rare circumstances is the machine code produced from the compiler and assembler, the latter being the software which conducts the mapping, as efficient as a good assembly programmer could produce; although with the speed of today's hardware, efficiency at this level is nearly never an issue of concern.

The second method of implementing a HOL through software would be by interpretation. This is similar to the compiling process described above in that an equivalent program in another language is generated by disassembling

the source program. Interpretation, sometimes called translation, differs by attempting to match the grammars of the source an target languages by function. For example the declaration of variable 'x' in source language A is transformed into the declaration of variable 'x' in target language B, without using an intermediate structure. Both the source and target languages have the same level of complexity, i.e. each are HOL's or each are assembly.

Considering the language D, the target HOL will then be compiled producing the necessary machine code. This implies that a compiler must eventually be available for the final target HOL.

Though less efficient interpreters are often used when a compiler is unavailable for the source language. Such is the case with D. Having been born less than one year ago a complier has not yet been constructed for it. Since compiler construction involves a considerably greater effort then the building of an interpreter, the latter route was chosen as the means to implement D at this time.

Backround on the Problem

The background on compilers, parsing methods, and interpreters is extensive and current. The main focus of the literature search will be on efficient methods of parsing, tree representations of data, and error recovery and reporting.

The backround of D on the other hand is scant. The

sole source of documentation on the language may be found in the thesis by Capt. Jennings, completed while he was at the Air Force Institute of Technology. Though information on D specifically is lacking, a partial understanding of D can be achieved through an understanding of it predecessors, namely C and Ada, of which documentation is prevelent.

Approach

The following sequence of events will be utilized to obtain a solution to the problem. First both a literature search concerning parsing and error detection/recovery, and a detailed analysis of D will proceed concurrently.

Error detection and reporting play an important role, because so little written material is available to the D user; therefore a desired quality of the interpreter shall be to produce accurate, and easily understood error diagnostics.

The analysis phase will cover the syntactic structure, from which the guidelines for the parsers construction will stem, and the heart of the interpreter: D's semantics.

Design methods will follow ideas introduced with the advent of software engineering e.g. Structured Analysis Design Technique (SADT) diagrams, structure charts, and data dictionaries. And shall proceed top-down as these methods demonstrate.

Coding of the interpreter has been determined to be in the programming language C. The pros and cons of this

decision are discussed in the requirements section.

Both during coding and when the program is running testing will be conducted in order to verify the program's correctness. The testing will be limited to correctness and robustness (how well does it handle the situations it finds itself in). There will not be a statement on its efficiency; though grossly inefficient code will be considered unacceptable. Validation will be confined to a path-testing methodology using sample D programs.

Literature Review

The research and information used in this thesis may be grouped into three sections: 1) the language, 2) previous interpretive work, and 3) methods of software engineering.

While one can understand how to program using D by reading Capt. Jennings thesis alone, a greater insight to the language design may be achieved if the reader has a certain familiarity with the programming languages C and Ada. D atempts to take the most useful of Ada's new ideas, and implement them, while retaining a compact language similar to C (5:ii). Jennings thesis is necessarly incomplete in its design. This is not a criticism of the work, rather than a warning to readers that the language will undergo further definition, and minor evolution during implementation. These changes may cause contridictions to arise with earlier writings. Therefore Jennings thesis serves as a good introduction to D, and a valuble source of design

answers, but should not be used as a programming guide.

The second area of research involves parsing algorithims, specifically those with error detection and recovery methods. The method chosen, while admittedly not the most efficient, implements the language in an easily understood fashion similar to Wirth's PL/0 parser (10:311). Other methods surveyed, but rejected were a phrase-level recovery scheme (ref 4), a context free parsing algorithim (ref 3), and error methods used with the Helsinki Language Processor (ref 8). The justification for their rejection will be included with the implementation section.

Finally the design process was shaped by L.J. Peters book Software Design : Methodologies and Techniques; as he gave the best description of a structure charts uses. Once more the reason for using structure charts will be discussed in the following section.

Material was both prolific and current concerning parsing and design methods, though often the authors attemped to place too much information on too few pages. In addition it would have been helpful to have compared Capt Jennings thesis with other written work on D, unfortunately this was not possible.

II. Requirements Definition and Justification

Every software effort which performs a function of some use other than introducing a student to programming should include a phase known as requirements analysis. In this phase the system is carefully studied, and from this inspection a concise, clear, and consistent statement is generated as to what the system will do (7:12), or intends to do. This statement (or statements) becomes the yardstick by which the system is measured, upon completion, to guage whether the stated goals have been met. Requirements definition can also serve a purpose with an unfinished work by providing guidence towards the intended goal. Therefore it is necessary to reduce ambiguity in requirements definition to a minimum while allowing flexibility for unanticipated and/or unavoidable problems.

The D to C interpreter proposed is no exception to the need for requirements definition. Broadly speaking the interpreter, hereafter also referred to as the system, has only one requirement; it must input, in some manner, a program coded in the D language, determine the meaning of the program structures, and produce, as output an equivalent program in C source code.

Within this global view many other requirements are evident. The method used to present them here will be to define the requirement; then follow it by an explanation or justification as to why this requirement was chosen.

Why C

As stated earlier a D program will be converted into C source code. Why was C chosen, as the target language, over more common languages like Pascal or FORTRAN, or Ada?

First, C was seen as superior to the Pascal and Fortran, because its architecture allows it the same or better flexibility and power than the other languages possess e.g. pointer arithmetic, better string handling routines, and lower level I/O providing increased functionality. Jennings (the author of D) also described D as the successor to C, indicating a greater similarity between them (5:ii&117).

Second, while Ada is capable of implementing a greater subset of D, possibly all, than is C, it cannot be used, because AFIT does not have an Ada compiler; therefore programs generated by the system would not be subject to dynamic testing. Furthermore the quick implementation of D, at AFIT, which was a driving force behind this thesis, would be lost.

In addition, preliminary work on a D/C comparison has been completed by Capt. Jennings. Henceforth C emerges as the natural language, currently, to target the translation of D.

Another requirement involving C is that the interpreter will be written in this language. This follows from the decision made above as it aids the system engineer by allow-

ing him/her to focus on two rather than three languages. Certain languages may also be quickly dispatched. FORTRAN's lack of structure, poor typing, and crude string handling make it a ludicrous choice for implementing an interpreter. Ada, on the other hand, would be an excellent choice, except for the lack of a compiler. It is therefore eliminated due to insufficient support rather than a deficiency in the language.

Pascal and C are then left as the only serious contenders. Both are more than sufficient for the purpose at hand, but C has a greater variety of I/O and string handling functions as part of its standard I/O package. In addition, C's structure lends itself to the use of the UNIX makefile facility, which is a time saving asset to programming. While it is possible to model Pascal's arrays in order to handle strings exactly like C, and alter the program structure so it will perform as well under makefile, one is still using C in an ideological sense. Furthermore Pascal would still be lacking in I/O, most importantly the error handling.

Finally C is the language of choice for current architectures (5:102). Hence C emerges as the best language available for this project.

System Constraints

A Vax 11/780 running the Unix operating system was chosen for this project. The Unix environment provides the

greatest degree of portability for C programs, since most C users are on a Unix system (6:159), and Unix has become prevelent in the last few years, especially in school systems (9:19) where this interpreter should spend its life. Unix also provides a direct connect capability between the operating system and the program through system calls should this be necessary (e.g. file creation and deletion).

A Vax computer was chosen as AFIT's most convienient and suitable system currently supporting Unix.

The interpreter will stand alone as a program. This does not imply that it will be one large file. On the contrary, it will be split up for use with makefile, but it will not call or incorporate other programs which have been previously developed. The reason underlying this is transportability. Reliance on other programs hinders transportability by forcing additional constraints on the system. If the system engineer creates dependencies on other programs beyond the scope of his/her control, there is a greater risk that another location desiring the system will not have the needed additional programs, or that those programs cannot be transported to that installation. Portability will be further enhanced by requiring all code sections which are non-standard, e.g. terminal control, operating system interfaces, etc., to be confined to separate routines.

This does allow the use of non-standard programs in designing, implementing, or testing the system.

The final constraint concerns C's inability to implement all of D, but the greatest possible subset will be implemented. Furthermore a section indicating the defficiencies will be included in this thesis.

User Interface

The system would accomplish its formost goal if valid D programs yielded equivalent C code, while invalid input was indicated by any abnormal program termination; however such a system would receive little use. Not only would this case be diametrically opposed to user friendliness, but as a learning aid it would be atrocious. A proper implementation given valid input should indicate its manipulation of external files as a minimum, and more importantly should report error conditions in a manner which will help the user correct them.

Most programmers need little experience with compilers before they realize the error messages produced are far from adequate (2:246). For example the C compiler supplies such as "syntax error", and "illegal character 134 (octal)"; or as in one Pascal compiler, "error in type of standard procedure parameter". Therefore this system will report any detected syntax errors along with the parsers view of where the programmer went wrong. This in no way guarentees all errors will be found; though a program containing at least one syntax error will generate at least one error message as soon after detection as possible. Nor will it insure that

valid code will not be cited as containing errors if it follows code with errors, as the recovery phase may cause further program difficulty.

It is recognized that this is a vague definition of error recovery, but quality in this area cannot be pinpointed. Still it is hoped that the quality will be sufficient to provide yet another learning tool for the beginning D programmer.

In addition detection of fatal and non-fatal semantic errors, e.g. variable type checking will be incorperated into this system. Its purpose is to further assist the programmer in catching subtle mistakes. Once again errors in previous sections of the program may cause a cascading effect throughout the program resulting in a greater amount of errors reported than the program actually contains.

The error warning itself will at a minimum contain a descriptor defining whether the error is fatal or non-fatal, the line number of the file it occurred on, and the reason for the error.

Unix Compatability and Program I/O

Since this system is being developed to work in the Unix environment, its invokation and file classification will follow the standard Unix style. For example, like the C compiler, this program will be ivoked as "dc <file name>". If options are added to control the program's control or loquacity, they will be input with the familiar

"dc -<option> <file name>"

format (9:230). A usage message will be incorperated with the system informing the user of the proper command syntax.

In addition, the C code file will contain a '.c' extension. If the input file has a '.d' extension (for D code), the 'd' will be changed to 'c', otherwise a '.c' will be added to the input name. A possible option would be for the program to warn its user if overwriting of an existing .c file would take place, and allow the user to route the output elsewhere.

Other Requirements

The question as whether to carry comments from the input program to the output program or not is an opinionated one. Arguments can be made for either side; therefore for preference reasons only it was decided to discard comments.

III. The Language

As stated earlier D, having been influenced by C, shows many glimpses of C's rich use of operators as well as its apparent lack of form. This free format being exhibited by permitting files to be mere collections of functions with or without a main routine for an entry point. D follows this structure, and indeed adds to this freedom by permitting unconstrained organization of the languages four file types.

File Order

However to comply with the assumption that one pass of the interpreter is sufficient to obtain the program's meaning an order will be imposed on the file types. An ordering will also permit greater semantic error checking. The following constraints will be added to the D language: 1) all scalars and structures must be defined in a state file before use in any other file, 2) all variables, operators, and functions must be declared in a storage file before being referenced in an actor file.

Although D is clearly a child, or at least in the C family of languages, one should not be misled into thinking the differences are insignificant. On the contrary, to the casual observer D appears totally unrelated.

This section describing the language will begin at the bottom with the primitive language symbols and will continue upwards to the file types, discussing their operation and

interaction. Henceforth when the word file is used it refers to one of the four D file types, state, storage, actor, or bundle. If a Unix system file is discussed it will be so named, or be clear from its context. All information contained in this section is derived from Capt. Jennings thesis (reference 5) except for constraints imposed on D due to the lesser capabilities of C. Four the reader familiar with syntax diagrams these are provided in Appendix A. Note that they show a top-down structure!

Naming

D naturally has four disjoint name spaces; an additional name space will be created to prevent name collisions. A further constraint of eight significant characters need be placed on sets which allow names greater than eight characters. This is due to the limitations of C as it is implemented on a VAX 11/780 system. Should this program be run on a system other then this one the eight character limitation may not apply.

The name spaces are broken into the following sets. Keywords, which are English words used as a form of punctuation to enhance the program's readability. A list of the keywords is provided in Appendix B. The second set operator names are any combination of one to three operator symbols. The functions and precedence of the operators is also listed in Appendix B. Variable, structure, and function names make up the third name space. These are strings of symbols whose

maximum length is determined by the maximum size of the terminal's input line; although names of this length are impractical. The string must begin with an alphabetic character, upper or lower case, and be followed by zero or more alphameric characters or an underscore. The fourth name space consists of literals, perhaps better known when refered to as constants. Literals follow the same rules as variable or function names except the first character must be a digit or an underscore. Since their value's may not change within an actor, they must be initialized upon declaration. Lastly, the fifth space, names in the set starting with three underscores, or ___, followed by alphameric characters is generated by the interpreter to circumvent name collisions due to the smaller C naming set. The actual mapping of C reserved words and operator names into the fifth set is described later. Literal names may begin with three underscores but the user is cautioned against this practice as it can only lead to trouble.

Comments

The language supports a facility whereby the user may comment his/her program as good coding style demands. The '~~' punctuation indicates the following stream of symbols is a comment, and is to be disreguarded by the interpreter, or in the future the compiler. The comment is ended by a line terminator symbol; although it may be extended over multiple lines with the line continuation character. If

extended the following line is all comment; in other words any program construct placed on the line will be disreguarded.

Classes

Two classes or types are predefined in D. The first is a range of integers; which is not actually defined by the language, but is included in the file standard definitions as int or integer. The size of the range is dependent upon which system the the language is implemented on. Standard_ definitions must be shared with all files accessing integers unless the user wishes to define his/her own set. Furthermore the interpreter will be designed so that if there exists a file called stndrd_def in the users workspace it will be accessed; otherwise a generic standard definition package, to be described later, will be used. The default for all files without a link to a standard_definitions file will be the generic one.

The second class, predefined for expressions only, is the quoted literal. These are either single characters surrounded by reverse apostrophies, e.g. `CHARACTER`, or character(s) surrounded by double quotes, e.g. "one or more characters". Non-printable control and formatting characters are not permitted in quoted literals.

Note that in order to define a variable to be of the quoted literal type one need define the class in a state file first using the class abstraction operator. An example

of this is provided at the end of this section where a demonstration and explanation of the language is given.

Operators

Three operators are predefined in the D language. The remainder are the responsibility of the user, unless a standard definition package is used. Those which need not be defined are the assignment operator, integer addition, and integer subtraction. From these three must all the other operators, both arithmatic and logical, familiar to users of other languages, be defined.

The assignment operator, denoted '=', takes the value of an expression on its left and assigns that value to the logical name on its right if the classes match. If the classes do not match a class conversion will take place if possible, and the assignment will be made; otherwise an error will be the result of the assignment. The class conversions possible are integer to character, floating or fixed point to integer, floating point to fixed point, and vice versa. The rules for the conversions are the ASCII character set, truncation, and rounding respectively. All other class conversions must be user defined.

Integer addition and subtraction are denoted '+' and '-' respectively. And although the programmer could denote multiplication by '/' and division by '*' it would be wise to stay with preconceived ideas concerning the matching of operators and their functions.

Examples of operator definitions, done so in actor files, will be demonstrated at the end of this section.

Punctuation

Before continuing further it should be noted that the heart of the D language is contained within its large punctuation set, of which keywords are a part. A description of this set, keywords excluded, will aid the reader in understanding the language to follow. Assimilation of the punctuation's meaning may be facilitated if the reader is familiar with three of the four D file types.

State files are where variables and structures are defined. For example:

```
rational :: { int : numerator
              /
              int : denominator
            }
```

would define a rational to be a structure composed of two integers separated by a '/'. Rational, now defined, becomes an object class like integer. Readers familiar with the programming languages C or Pascal may liken the state definition to C's typedef or Pascal's type declaration.

The Storage file creates variables of the types specified, or declares the input and output types of operators and functions. The former being like the var in Pascal or the variable declaration in C e.g. int name. The latter is similar to the type declaration preceding functions in C, and of it arguments following the function header. An

example declaring 'm' and 'n' to be rational variables, 'i' to be an integer variable and '*' to be an operator accepting rational arguments and yielding an integer result follows:

```
rational : m, n
integer : i, * rational
```

With the above declaration i = m * n is a valid expression.

Actor files define operators and functions and are the only files to contain program statements, after declaring their input and output variables.

Table 1 describes the role punctuation plays in each of the three files.

Table I: Description of Punctuation Symbols (5:49)

symbol	S A D files	meaning
{}	S	delimits structure definitions
	A	delimits blocks
[]	S A D	denotes an array by following a <name>
	d d D	with an index range optionally specified by scalar value within square brackets
	e A e	with an index optionally specified by a scalar value within square brackets
()	e A e	controls expression precedence
)	S	suppress zeros to right
(	S	suppress zeros to left
,	S A D	line continuation character
:	d d D	<class> [or type] delimiter -- declaration
::	S A	<class> [or type] delimiter -- definition
~~	S A D	comment initiator
`	e A e	single symbol quoted literal delimiter
"	e A e	string quoted literal delimiter
'	e A e	possession indicator
#	d d D	creates pointer to declared class
	e A e	obtains object referenced by pointer
@	e A e	address abstraction on following <name>
$	d d D	class abstraction on following <name>
	A	delimits guard expressions
?	A	selects block behavior
<blank>	A	goto next line of block
;	A	goto next open line of block
\	A	goto beginning of block and reexecute

KEY:
- S - contained in state file
- A - contained in actor file
- D - contained if storage (Declaration) file
- e - only appears in expressions [S or D file]
- d - only appears in declarations [S or A file]

Expressions

Expressions in D take on many forms these being a literal or quoted literal, a logical name optionally prefaced by an '@', two expressions bound by an operator or an expression with a prefix or suffix operator. Others are an expression in parentheses, a function name optionally followed by an expression, the reserved word "null" or an accept function. All expressions are reducable, given the values of their respective subexpressions, to a single value of a specific class subject to the rules of operator precedence and binding.

Punctuation preceding a logical name specifically '@', '#', ''', and parentheses bind the tightest, or have the highest precedence. Next in the binding hiarchy are function names, than operators. The operator precedence is determined by the right most operator in the name, and is as follows:

!, ^	highest	operators on the same level have the same precedence
*, /		
~, %		
+, -		
&, |		
<, >		
=	lowest	

For example the operators +=^, **+, =/|, and !!= are in order from highest to lowest.

If an '@' precedes a logical name the value returned is the address of that logical name. In other words the value

returned is equivelent to that which a pointer pointing to that logical name would have. A '#' is the inverse of an '@'. If a '#' precedes a pointer the value returned is the value of the object pointed to. One could say logical_name = #@logical_name. The expression pointer = @#pointer is not valid though. Multiple #'s may precede pointer in order to follow a chain of pointers; although too many will return a "null" value. The number of valid #'s the user may string together is dependent upon the system.

Array names if not followed by the square brackets constitute pointers to the array, specifically the first element in the array.

Components of a structure can be accessed individually through the use of the ''' punctuation. For example given the definition for a rational number described earlier one could assign k, if it were an integer, to the numerator of the rational number m through the expression
k = m'numerator.

A pointer should have the value "null" when it is not pointing to anything. The user should take care when assigning a pointer to null. For if the object which that pointer referenced is not referenced by another pointer it's lost.

Finally multiple assignments in an expression are allowed by D; i.e. var1 = var2 = var3 = expression is syntacticly correct.

Statements

D is a very compact language having only four statement types. The expression already discussed is the first of them. Second is the return. This statement is a semantic twin of the C return statement. It terminates the current actor and returns to the calling function. The statement allows an optional expression to be evaluated and its value become the formal return variable. A function if defined without an output variable will generate an error if it contains a return with an expression.

The third statement is the "new" function. Taking a pointer as an argument it dynamically creates a storage location of the class #pointer. Alternately said, given the class of the object which the pointer is allowed to point to another object of that class is created. The pointer then receives the created objects location as its value, and any object previously referenced by pointer is lost. That is, unless another pointer had also referenced the object.

The fourth statement type, a block or compound statement, is of an importance to deserve a section of its own.

Blocks

Syntactically a block is a header followed by linkage and one or more lines inside curly braces. Although it is a statement, statements may only appear inside a block, with the exception of the expression. Blocks appear only inside actor files to define the actor. Primed with this the

subconstructs in the block will be identified.

The Header. The block header consists of a label and an optional selector. The label follows the same naming conventions for function names. It provides part of the jump capability as any statement within that actor may jump to the beginning or end of a labled block using the continuation.

The selector is an expression; if present the value of the expression becomes the value a guard must have to be declared open. If the selector is not present any positive guard value is declared open. While all negitive guard values are closed.

Block Linkage. A discussion on linkage is presented later, but it should be mentioned renaming may not occur inside of a block.

Lines. Lines, in D, are charged with the responsibility of program control. Consisting of a statement, optionally prefaced by a guard, and followed by a continuation symbol, they emulate the three basic control structures, conditional, sequential, and iterative.

Conditional control is implemented through the guard. Syntactically an expression followed by a '$', the guard is evaluated; if it matches the value in the first surrounding block's selector (or the default values, non-zero numbers, if no selector is present) the statement following the guard is executed. Otherwise control is passed to the next line

for guard evaluation.

Sequential control may be forced and iterative control is handled by continuations. They consist of an optional label followed by one of four continuation characters. The optional label indicates the actions of the continuation shall be applied to the named block. If a label is not present the current block, or first enclosing block is used.

The continuation characters are: 1) <blank>, denoting forced execution of the immeadiatly following statement reguardless of whether its guard is open, closed, or non-existent, 2) a ';', indicating normal control flow, as in other languages, to the next open statement, 3) a '\' which directs the program to reenter the block; including reevaluation if the selector and guard values if any exist, and 4) the '..' symbol terminates the block.

Files

Files may now be discussed in detail, along with their respective components and fuctions. A D program is divided into four file types, two to define objects, one to declare them, and one to organize the files. Each file need not repose in a separate Unix system file. On the contrary, D files should be combined in Unix files to create functional programs. The keywords *state*, *storage*, *actor*, and *bundle* determine the type of file which follows. Each file is then given a name, which may be used to reference the file by another. Referencing may only take place between files in

the same Unix file, or mutiple Unix files bound together through the use of the Unix #include facility. A line terminator separates the file declaration from its linkage, following which is the file body. It is the body where the file types are unique. Every file is then terminated by a period appearing on its own line, with nothing but white space preceding it.

Linkage. Linkage is the means whereby modules communicate with each other. It provides three functions to the user, sharing, copying, and renaming.

A file may gain access to another by including the desired file's name following the share keyword. This action permits the naming file to reference any object, whether variable, function, or operator, in the named file. The link created is one way, unless a corresponding share statement exists in the named file; although this would rarely be necessary. Communication between files may occur when two or more files share a common modifiable area, i.e. a storage file. The definition files, state and actor, are usually shared.

The copy function, used in the same format as share, also provides access, but to an identical copy rather than the original file. There is no limit to the number of times a file may be copied; therefore if five files each copy another, five individual copies are produced. Storage files are often copied to prevent other files from accessing and

possibly destroying data, or they may be shared to provide interfile interaction. Copying a definition file is equivalent to sharing it, since modification on these files, during execution, is not possible. Bundle files may be shared or copied. The result is the respective function over all files named within the bundle file. For example, if file x shares bundle file y, and y contains files a, b and c, then it is as if x had shared a, b, and c, calling each one by name.

Renaming provides to user with the ability to change an objects name to a more desirable one. The idea was born as a way to facilitate program readability when linking utility routines, which often have meaningless names. Assume, for example, a storage file with a structure for personnel called "person", and a scratch variable "i". Let "person" contain three data areas, data1, data2, and data3, and further assume this storage file is being linked to an actor which performs an employee payroll function, which already has a variable "i" declared. To add clarity and to avoid a naming ambiguity the user may rename as follows:

```
rename i  j
rename person'data1  employee'name
rename person'data2  employee'position
rename person'data3  employee'address
```

The question concerning which "i" would be used if the user forgot, or did'nt care to, rename "i" will be left until the design of the interpreter. As mentioned earlier, renaming

may not occur in block linkage.

State Files. The state file, as mentioned earlier, is similar in it function as the C typedef. After linkage the user defines scalars as a range of integers, or structures from known templates. Once defined the scalar or structure names become known classes from which other objects may be defined or declared. Scalars are defined by providing a name, the double colon punctuation, and a pair of integers, the latter being greater, separated by a double period. The scalar's range is inclusive of the integers given. A shorthand notation is available on occasion by supplying a single integer. The range then lies between the value specified and zero inclusive.

Structures consist of a structure name, the double colon, and curly braces surrounding the components. Each component may be a class colon declaration, discussed under storage files, or a mixture of printable symbols (psymbol in the syntax diagrams, Appendix A) and periods, or the parentheses, meaning supression of zeros. If a variable is declared as a structure containing only one component, the component may be referenced by the variable name only. An example, of a structure, has been given under the subsection Punctuation and the paragraph beginning with State, or the reader may wish to reference the language example given at the end of this section.

Storage Files. Used to declare or instantiate from

classes defined in state files the storage file body, that section which lies between linkage and the terminating period, consists of at least one of the following set of symbols. Each set occupies its own separate line. The set is a defined class, a single colon, and one or more declarations, which are object names, separated by commas, with their optional initialization. The format of the set is similar to the Pascal var statement. Differences exist in that the class precedes the colon. The class abstraction operator, discussed in detail under actor files, may be used to provide the class. The other break in similarity is Pascal's inability to perform variable initialization with instantiation.

Optionals symbols may procede and succeed the object name, except function and operator names, to create new types. Names proceded by a '#' are declared to be pointers to the named class, while names followed by '[]' with an optional integer between the brackets are declared to be arrays of that class. The integer determines the size of the array. Should one not be present the array is an indefinite size. The pointer symbol, '#', binds tighter then the array brackets; therefore an object declared as so: class : #object[5] is an array of five pointers, which may point to objects of type class. If an array is initialized all its members are assigned the initializing value.

Functions and operators need also be declared. The

format is the same as for objects, except the input argument class follows the function or operator name. The function's or operators's class is defined by the class of its return value. If a function has no input or output values the keyword "null" is inserted as the class for the missing arguments.

Actor Files. Having now the ability to define objects, and declare objects, functions, and operators the actor file provides the ability to define the latter two and manipulate the former. Encased between the actor file's linkage and terminating period lies the only area where blocks exist, the actor file body. This region is split into one or more definitions, each of which are called actors. There is no upper bound on the number of actors which may reside in a actor file, so long as each actor is properly declared in a storage file which is linked to the actor file. All variables referenced by the actors must also be predefined, and declared. Linking may be postponed until entry into the block.

Further decomposition of the actor reveals two sections, the operator or function declaration, and the block. Having previously discussed the block, attention will now be directed on the declaration section.

The operator declaration consists of the operator name, a double colon, followed by the keyword "in". Operators must have one or two arguments and the "in" declares the

input argument class follows. A single colon separates the class from the declaration, just as in the storage file. The difference here lies in that a maximum of two declarations are allowed. The comma between them appearing where the operator would in an expression. Therefore if the operator being defined is a unary prefix operator the first declaration is left blank, and the second declaration is preceded by a comma. In a like manner if a unary suffix operator is defined only the first declaration is used; the comma and the second declaration are left blank. All binary operators must be infix.

Following the declaration of the input arguments is the like action for the operator's single output argument. Initiated by the keyword "out" it declares an object in the same manner as the unary suffix operator input, a class, single colon, and declaration. Both the input and output declarations are completed by a line terminator symbol.

The second type of actor declaration, that for functions, is similar to that described above, but since function arguments are optional differences arise. Again the function name and a double colon start the decaration off, and may be followed by "in" if the function receives a value. If a value is received the declaration is that used by the unary suffix operator input. This may appear contrary since function names procede their argument, but as a maximum of one is possible the second declaration is never

needed. The functions output or return declaration is identical to the operator output, except when ommitted due to no return value. Therefore a function without arguments, either way, is declared simply as: function_name :: line terminator.

After the operator or function arguments have been declared renaming is permitted preceding the defining block. A line terminator symbol follows the block to apply the restriction the file terminator, or period, must lie on a line by itself.

Bundle Files. The bundle file is unique in its function, differing from the other three files in that it organizes files rather than manipulate or define objects. Therefore, though the syntactic structure of its linkage is identical to the other file's, the meaning has changed. The share and copy functions provide their respective actions globally for all files organized by the bundle file. The rename function also performs its assigned task throughout all files named within the bundle's body.

The body of a bundle file is simply a list of file names, each appearing on their own line. These names, organize the files into logical packages. Every file (bundle files excluded) must be declared within its perceeding bundle file body, or an error is generated.

Input and Output

D like many of primitive high-level languages was con-

structed with low-level input and output (I/O) capabilities. D's I/O is simply the assignment of a value to a location in memory, or the receiving of a value from a memory location. The specific memory address is actually a port for some device.

Open, Creat, Close, and Unlink. File access, on the other hand, has not been defined; therefore, it will be necessary to interface with the Unix operating system. To access other than the standard files, is accomplished with the open, close, creat, and unlink functions. Open and creat, which need two arguments each, will take a structure as there argument. The structure's components are a file name, or an array of characters terminated by a literal zero bit pattern, and an integer. This structure is also defined in the language example as file_io_structure. The integer in the open function declares access as read only, 0, write only, 1, or read and write, 2. In the creat function the integer declares a protection mode for the Unix system. Protection is a nine bit number indicating read, write, and execute permissions for the file owner, his/her group, and all others respectively (6:162). If the bit is set it indicates permission is granted; a common protection is 755 octal or 493 decimal. For example a bit pattern of 111100001 gives the file owner full permission to read, write, or execute the file. Members of the owners group may read it only, and all others may execute it only. open and

creat return as a value an integer in the range of 3 to 15-25, to use as a file descriptor for subsequent reading or writing to that file.

The inverse function of open and creat are close and unlink. They require only the file name, as a string of characters terminated by a literal zero, as their input argument. Programs may have a maximum of 15-25 files, the number varies with systems, opened at any particular time; the close function allows the file descriptor to be reused, while leaving the file intact. Unlink, on the other hand, removes the file from the system. The four file operation functions all return as values either a non-zero integer denoting error free execution, or a zero if an error occurred. These functions will retain their Unix names and be accessable as open(file_io_structure), creat(file_io_structure), close(file_name), and unlink(file_name). A more detailed discussion on the interface to the Unix system is available to the reader in Appendix C. Though dealing with the C language it will further expand on what has been presented here, and may provide some useful higher level I/O functions.

An Example

The following subsection is a sample D program. The method in which it is presented will be the same as that used in reference 1. The numbers and colons to the left of the page are not part of the language, but serve to identify

lines for explanation.

The object of the program is to read a given number of rational numbers, and sort them using a binary tree. The assumption is made that the function get_number exists, but is not declared, and is globally shared with all the file present; it could be in another Unix file or the same one. The remainder of the code presented resides in a single Unix file.

*** code segment ***

```
1 :  bundle rational_sort
2 :  structure_definitions
3 :  io_struct_def
4 :  variables
5 :  mul_var
6 :  build_tree
7 :  .
```

*** explanation ***

```
1 :  the program is headed by a bundle file named rational_
     sort.
2 :  The file structure_definitions is declared to be in
     rational_sort.
3 :  The file io_struct_def is declared to be in rational_
     sort.
4 :  Ditto for variables.
5 :  Ditto for mul_var.
6 :  Ditto for build_tree.
7 :  The bundle file is terminated.
```

*** code segment ***

```
1 :  state structure_definitions
2 :  rational :: { int : numerator
3 :                /
4 :                int : denominator
5 :              }
6 :  complex :: { rational : real_part, imaginary_part
7 :             }
8 :  treenode :: { rational : number
```

```
9 :                   treenode : #left, #right
10:              }
11:  list_info :: { treenode : #node
12:                 int      : n
13:               }
14:  .

15:  state io_struct_def
16:  share structure_definitions
17:  io_structure :: { int  : file_descriptor
18:                    char : buffer[512]
19:                    int  : n
20:                  }
21:  file_io_structure :: { char : file_name[20]
22:                         int  : mode
23:                       }
24:  .
```

*** explanation ***

```
1 :  the state file structure_definitions is initiated.
2 :  the structure rational is defined to be an integer
     numerator,
3 :  followed by a '/',
4 :  followed by an integer named denominator.
5 :  the definition of the structure rational is terminated.
6 :  the structure complex is defined, and has the integer
     members real_part, and imaginary part.
7 :  the definition of the structure complex is terminated.
8 :  treenode is composed of a rational number named number,
9 :  followed by two pointers, left and right, which may
     point to an object of the class treenode.
10:  now defined treenode, like rational and char, becomes a
     class
11:  list_info is a structure with components node, a poin-
     ter to a treenode structure,
12:  and n, an integer.
13:  the list_info definition is terminated.
14:  the state file is terminated.
```

*** code segment ***

```
1 :  storage variables
2 :  share structure_definitions
3 :  rational  : number, get_number null
4 :  treenode  : #root = null
5 :  list_info : info
6 :  int       : > int, > rational, * int, == int
7 :  null      : main_routine int,
8 :              put_in_list list_info
9 :  .
```

```
10:  storage mul_var
11:  int : sign = 0
12:  .
```

*** explanation ***

1 : the storage file variables is initiated.
2 : linkage is made to the classes in structure_definitions.
3 : the variable number is declared to be of class rational, and the function get_number is declared to return a rational result and receive no inputs.
4 : root is declared to be a pointer to the class treenode, and is given an initial value of null.
5 : info is declared to be of type list_info.
6 : the operator '>' is declared to return and integer and accept integer arguments, there is also an operator, '>', which returns an integer, but it accepts rational arguments. The operators '*' and '==' both accept and return integer arguments
7 : main_routine and put_in_list are declared as functions which do not return values, but do require arguments of the classes
8 : int and list_info respectively. Note that although the comma following main_routine is a name delimiter, it can function in the dual role of a line continuation character also.
9 : the storage file variables is terminated.
10: the storage file mul_var is initiated.
11: an integer variable, sign, is declared, and initiated to zero.
12: the storage file mul_var is terminated.

*** code segment ***

```
1 :  actor build_tree
2 :  share structure_definitions,
3 :        variables
```

*** explanation ***

1 : the actor file build_tree is initiated.
2 : linkage is connected to structure_definitions,
3 : and variables. Remember it is assumed that get_number is defined and shared

*** code segment ***

```
4 :  main_routine :: in integer : count
5 :                 { count > 0 $ number = getnumber
6 :                           0 $ info'node = root
7 :                           0 $ info'n = number
8 :                           0 $ put_in_list info \
```

```
9 :                 }
```

*** explanation ***

4 : an actor function is defined under the name main_ routine; it receives as input an integer, and gives it the local name count.

5 : the defining block is opened by '{', the expression following is a guard because it is followed by a '$'. If the expression, once evaluated, is greater then zero the statement after the '$' is executed. Control passes to line 10 in order to evaluate the expression. Following the statement is the line continuation character <blank> indicating the next statement will be executed even though the guard is closed. If 'count > 0' is evaluated non-positive the block will be terminated since all other statements in the block are closed.

6 : the node component of the variable info is assigned the same value as root. Execution of this statement may only take place if the preceding statement is executed.

7 : the n component of info is assigned the same value as number.

8 : the function put_in_list is called with info as its argument.

9 : the defining block of main_routine is terminated.

*** code segment ***

```
10:  > :: in  int : intl, int2
11:        out int : true
12:    { intl - int2 $ return 1
13:                    return 0
14:    }
```

*** explanation ***

10: the operator '>' is to be defined; its input arguments

11: are integers, and its output argument is an integer.

12: the defining block is opened an the expression intl - int2 is evaluated. If the result is positive the operator will execute the following statement and return a value of one. The return also terminates the action by giving control to the calling actor.

13: if the expression in 12 is negative or false a zero is the return value.

14: the defining block is closed.

*** code segment ***

```
15:  put_in_list :: in list_info : info
16:     rename info'*node pnode
```

```
17:      rename info'node  node
18:      rename info'n     n
19:      null
20:      { node $ { new node;
21:                 pnode'number = n;
22:                 pnode'left = pnode'right = null
23:               } ..
24:               { n > pnode'number $ { node = pnode'right
25:                                        put_in_list info
26:                                      } ..
27:                                      node,
28:                                        = pnode'left
29:                                      put_in_list info
30:               }
31:    }
```

*** explanation ***

15: the function put_in_list is identified as receiving an argument of type info_list, named info.
16: the local variable info is has one of its components renamed so info'#node can be referenced by pnode
17: the local variable info is renamed so its components are accessable as node
18: and n.
19: the expression 'null' precedes the defining block therefore for a guard to be declared as open it must evaluate to null.
20: the defining block is opened and the value of node is determined; if it is null, then execution continues with the opening of the unnamed block following the '$', and an object of the type node points to is dynamically created. node receives the location of the created object as its value. Otherwise control passes to line 23.
21: the component number of node is assigned the n's value.
22: the left and right pointers of node are assigned null.
23: the current block is terminated and as a statement its line continuation character directs the program to terminate the current block, which now is the defining block.
24: an unlabled, unguarded block is opened and the following expression is a guard. If evaluated non-positive control passes to line 27, otherwise another block is opened, and node is assigned the right child of #node.
25: the function put_in_list is recursively called with info as its argument.
26: the unlabled block is terminated, and the current block is terminated, by the '..', causing control to pass to line 31.
27: node is assigned the left child of #node.
28: the comma was used to continue the line starting on

line 27.
29: the function put_in_list is recursively called with info as its argument.
30: the current block is terminated.
31: the defining block is terminated.

*** code segment ***

```
31:   > :: in  rational : rat1, rat2
32:          out int      : true
33:      rename rat1'numerator    n1
34:      rename rat1'denominator  d1
         rename rat2'numerator    n2
         rename rat2'denominator  d2
35:      { n1*d2-n2*d1 $ return 1
36:                      return 0
37:      }
```

*** explanation ***

31: the definition of '>' for rationals is initiated.
32: it returns an integer result.
33: the component rat1'numerator is renamed to n1
34: the other components are renamed in a like manner
35: the defining block is opened, and the expression n1*d2-n2*d1 is evaluated; if positive the actor terminates by returning a value of one.
36: Otherwise it terminates by returning a value of zero.
37: the defining block ends.

*** code segment ***

```
38:   * :: in  int : int1, int2
39:          out int : product = 0
40:      { copy mul_var
41:        { int1 == 0 $ ..
42:          int2 == 0 $ ..
43:                      product = product + int1;
44:          int2 > 0  $ int2 = int2 - 1 \
45:                      int2 = int2 + 1;
46:                      sign = 1 \
47:        }
48:        sign $ product = 0 - product
49:      }
```

*** explanation ***

38: the integer multiplier operator's definition is started.
39: it returns an integer which is initialized to zero.
40: the defining block is opened and linkage is extended to the storage file mul_var. The variable sign still has

its initialized value because the copy function preserves it. The share function will do the same, but only if the variable has not been previously modified.

41: an unlabled block is opened and intl is checked to see if it equals zero. This is accomplished by passing control to the '==' operator. If intl is zero the current block terminates.

42: int2 is checked in the same manner, also causing block termination if zero.

43: product is incremented, or decremented by intl.

44: if int2 is greater than zero it is decremented, and the current block is reexecuted.

45: otherwise int2 is incremented,

46: sign is assigned the value of one, and the current block is reentered.

47: the current block is terminated.

48: if sign is positive the product's sign is changed.

49: the defining block is terminated.

*** code segment ***

```
50:  == :: in  int : intl, int2
51:         out int : true
52:     { intl - int2 $ return 0
53:       int2 - intl $ return 0
54:                     return 1
55:     }
```

*** explanation ***

50: the '==' operator is initiated, and declared to require integer arguments.

51: an integer result is returned.

52: the defining block is opened and intl - int2 is evaluated; if the result is positive the actor terminates by returning zero.

53: otherwise int2 - intl is evaluated. Again if positive the actor terminates by returning zero.

54: otherwise a one is returned, terminating the actor.

55: the defining block is terminated.

This program does not attempt to demonstrate the entire D language, but it is included as means whereby the reader may gain a greater understanding of D.

IV The Design

Design is a human activity that has eluded definition for some time (7:23), partly because in every problem decisions are made which are based on the engineer's subjective feelings. Software design is further complicated by personal opinion as to what the best means for the problem solution is. Design possibilities may be as varied as the solution space permits, forcing evaluation criteria to be politically motivated. Software design quality can no longer then be termed as right or wrong, but must be thought of as a continuum from good through fair, poor, and bad (7:23).

Though design cannot be formalized into a step by step process yeilding a singular result; the engineer is not excused from using tried design techniques to ensure the final product occupies the highest level of the quality continuum. There exist a plethera of methods to act as a basis for the solution. These may be classified into four basic catagories: 1) top-down structured design, 2) data-structure design, 3) Parnas decomposition criteria, and 4) object-oriented design (1:32). No one method can be singled out as the best for all circumstances. Each of these methods have their respective problem sets for which they were defined. For example, the data-structure design technique imparts a fine detail to program objects, observes their interrelationships, and then forms the program structure from these relationships (7:152). Such a technique

works well for COBOL applications where operations take a back seat to data elements. Top-down design, on the other hand, resides at the other end of the spectrum placing the emphasis on operations, increasing the likelihood of a functional solution, at the expense of the data structures.

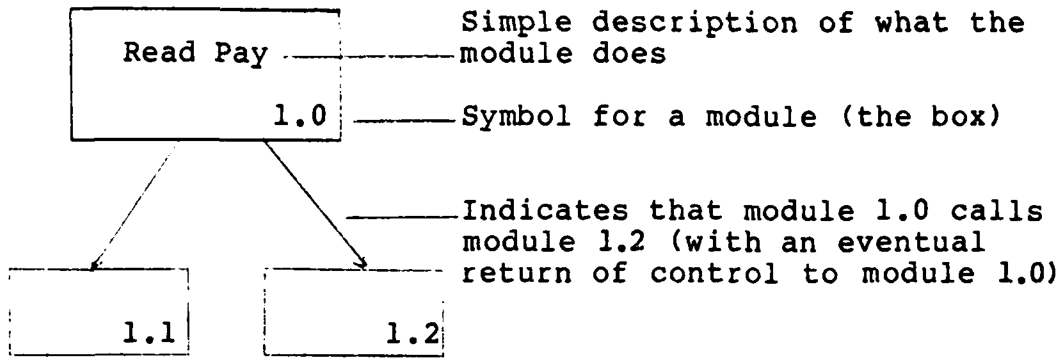
While both the Parnas method and object-oriented design achieve a healthy balance of actors (operations), and objects. These two methods appeared flawed in that they don't lead to a clear, logical orientation of the program structure. A better design method, and the one to be used for this system, will be a combination of the top-down and object-oriented designs. This method will yield a greater degree of flexibility for data element construction, while decomposing the system functionally; thereby allowing it to be modled after the D language structure.

During design one attempts to model the problem into some software representable abstraction, which will permit an algorithm(s) (the implementation) to manipulate the input into meaningful, human interpretable form, or that which another program will understand. This modeling seeks to match the structure of the problem into logical units. These units have two types; the passive types are objects (or nouns); they are acted upon by the second type called operators (or verbs). The nouns are instantiated as typed constructs according to the implementation language's rules. If these rules are weak, as in FORTRAN, further abstraction

by the engineer is demanded. Meanwhile the verbs become modules, or subprograms of the set language.

Until now the process described is the initial steps taken in object-oriented design. Top-down design is introduced by functionally decomposing the modules into successivly less complicated operations. This breaking apart of modules continues recursively until every resulting module is either simple enough to code, or its further decomposition is implementation dependent. The object-oriented analysis of nouns and verbs is repeated at each level of decomposition.

The method used to pictorially show the structure of the design will be structure charts. This method was chosen for its readability, simplicity, and ease of documentation.



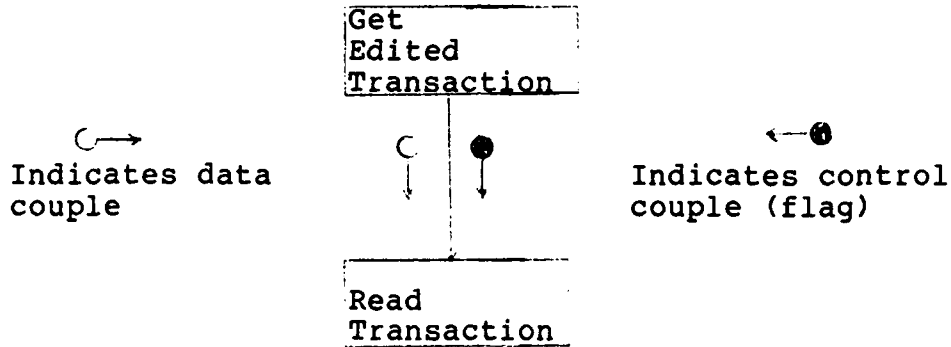


Figure 1: Structure Chart Notation (5:61)

In addition, the following will be used to show control sequencing:

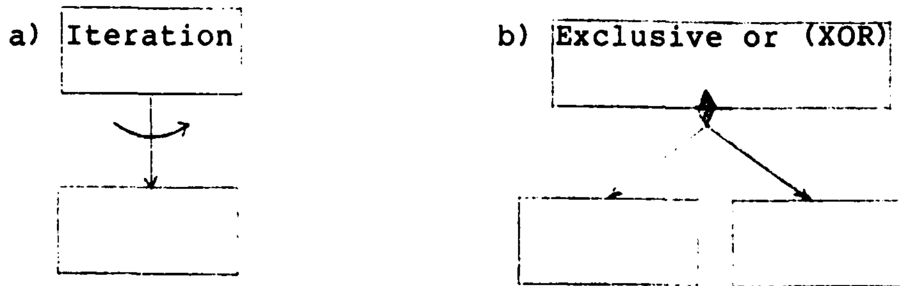


Figure 2: Structure Chart Control Sequencing

The initial design problem is to interpret a D program. Since the input form is already predefined, and the output must be a C program file, for the C compiler, the the global

data input and output have been set. The system at its highest level then has the following structure.

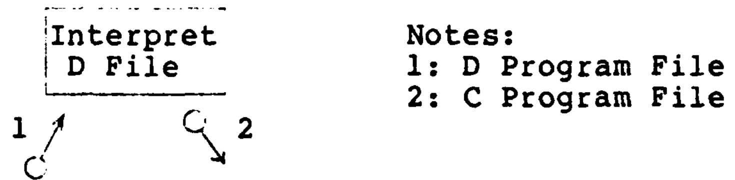


Figure 3: Global Design

To implement this design with code would be ludicrous; hence a further dividing of "Interpret D File" is needed. This decomposition, which will generate more data elements and procedures, must address the question: along what lines will the decomposition proceed? Experience provides guidelines for a distinction between the lexical analysis and the actual language interpretation.

The Lexical Analyzer

The lexical analyzer returns units, called tokens, along with their additional characteristics. From this description a module called get_token is perceived which will return an object of type token. Tokens will then be provided to the interpreter, forming the structure in figure 5.

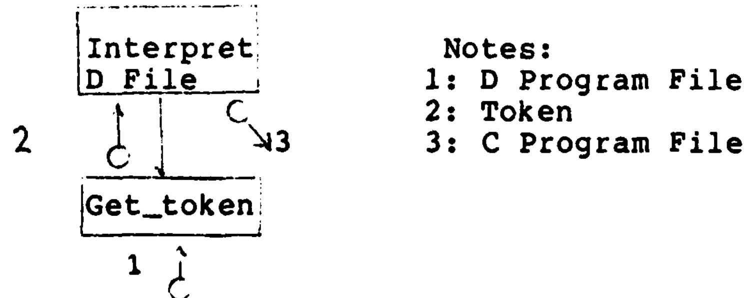


Figure 4: The Addition of Get_token

Now a token has the following properties: 1) its name - the literal string of characters which form it, 2) its type - the lexical class to which it belongs, and 3) its position - the line number it occupies in the program file. It is possible to implement a greater detail for the third characteristic, i.e. by supplying the token's position on said line, though this increased detail is rarely needed.

Get_token is still to complicated to code; therefore an orderderd fracturing of it becomes necessary. Since the module is dealing with tokens, the logical choice would be to split it along one of the token characteristics. A split depending on naming would be impossible, because the possible combinations of names would exceed the computer's memory limitations. In a like manner, keying on the token's line number provides little information, and varies widely. Hence the decomposition of Get_token should key on the classes of tokens.

D tokens will be catagorized into seven types: literal names, integers, identifier names, keywords, punctuation symbols, operator symbols, and comments. This disection disagrees with Jennings view of five token types: symbols, names, quoted literals, integers, and comments (5:46), yet no contridiction exists. The former list merely expands on naming, and symbols, while reguarding quoted literals as expressions (5:55) to be handled by the interpreter.

Figure 5 shows the final structure of Get_token. Note, an

implementation dependent module, get_character, has beed added to "read" the D program file.

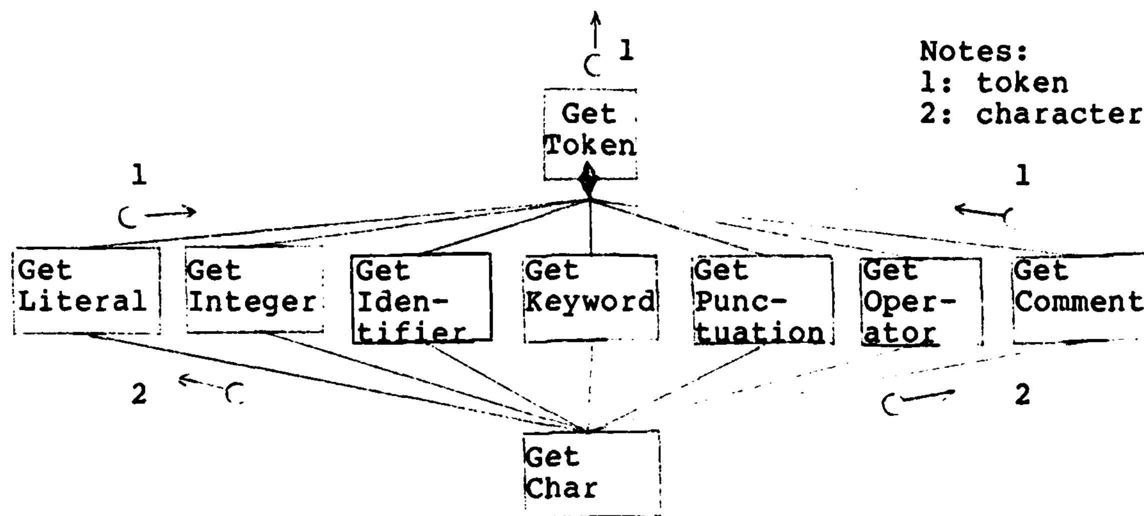


Figure 5: The Structure of Get_token

The Interpreter

The interpreter, like Get_token, must also be broken down into smaller units. Unlike Get_token, a break along lexical classes would only produce confusion. The interpreter deals with another well defined structure though, that is the D program input. The input's consistency lies in that it must correspond to established syntax rules. Though the interpreter must maintain flexibility for input containing minor aberations from D's structure, i.e. provide syntax error recovery, it basically receives D structured code.

In addition, the parsing of the program, the interpreter's first major function, should provide nearly all of the control sequencing, as the systems actions are contingent upon the D program. Hence the system structure should be

modeled after D's.

Wirth presents a method for moving directly from a deterministic syntax graph to a parser (10:291). While this technique is an integral part of the following design, it cannot be fully implemented, because D fails the second of two restrictions imposed on the graph. Specifically, all first symbols following a null path must be disjoint from the first symbols which are alternatives to the null path. Furthermore a formal root, or goal, was not part of Capt. Jennings language; although it is intuitively grasped that the D files should be grouped into conventional program files.

Interpret

The module Interpret will be the driver of the interpreter. Its function will be to initialize all externals necessary through function calls. These pre-interpret functions are Unix file set up, the writing of D's run time environment to the appropriate file, variable initialization, and structure initialization.

Furthermore, since the D files are not syntactically connected (rather there connection is logical through the linkage construction), it is the responsibility of Interpret to control their calling. The files, maintained as a list or tree, have certain properties. These properties will be grouped by the C structure, and are:

* name - the file name following the file type keyword

* type - bundle, state, storage, or actor
* code - another structure holding the C code generated by the file's body
* vdef - variables (classes) defined; a state file function
* vdec - variables declared (instantiated); a storage file function
* fodef - functions or operators defined; an actor file function
* fodec - functions or operators declared; a storage file function
* rtree - a tree containing all the renaming actions taken
* next (or left & right) - pointer(s) to the next record

The flow of control for Interpret will be to determine the file type by its keyword, perform linkage with the Linkage module, then call the appropriate file body module to interpret the file. This sequence will iterate until all files have been translated. The arguments passed to the file body modules are the D program file pointer (dptr), the C program file pointer (cptr), and the root of the file record structure. Linkage will receive the same minus cptr.

Linkage

The module Linkage proceeds the call to each file body. Linkage is partially generic in that it will process the linkage of all four files, as well as the renaming following function or operator declarations, and the linking of variables inside blocks. Though it needs to be told which of the above situations it is in, because the restart patterns, in the event of an error, are different for each case. Linkage will not generate code, as C contains no equivelent structure, but will manipulate the file records recording file

relationships. For example, a share operation from a state file to a storage file will cause a duplication of the state file's vdef list to be placed on the storage file's vdef list.

Linkage's decomposition then consists of an iteration over three modules Share, Copy, and Rename, each of which will receive the variables dptr, root, and file_type. The first two will call functions to duplicate and move the file record's list, while the latter will access a function to add name pairs to the rtree node. Rename must also access a module which will parse a logical name as one of the named pairs. Discussion on this module will be deferred until later.

The Bundle File Body

The bundle file body, module Bundle, is very simple. Its task is to read a list of file names from input, and place those names into the ordered structure which contains the file records. The algorithm involved is: while a name is read put it in the file structure end while. Throughout the interpreter much list activity takes place with different objects. The algorithms involved for placement and retrieval will be the same reguardless of what objects are being manipulated. Therefore, whenever a function is referenced to operate on a list it is assumed the reader is sufficiently versed in linked list and tree walking techniques to omit further detail.

The State File Body

The state file body is considerably more complicated. A state file defines two types of classes the scalar, and the structure. It will iterate over these two calls to functions until a file keyword or terminating period is found.

The structure differs from the scalar following the receipt of a token named '::. If a '{' is next the module Structure will be called, and if an integer or integer expression is detected Scalar is called, otherwise an error is generated. Scalar and Structure are passed dptr, so they may in turn pass it to Get_ token. Both also contain sufficient arguments discussed below to return all the necessary information gleaned from the parsed code. State then places that information into yet another data structure to be added to the state file's vdef list.

This new structure, called an sosnode (for state or structure node), has the following elements: the class' name, its members (another structure), a lower bound, an upper bound, and appropriate pointer(s) to the next sosnode.

Members will be a linked list of character arrays which hold the names of the structures members. Members, along with "code", a variable if like structure, will be passed to Structure, while lower bound and upper bound will be passed to Scalar. Both modules are also sent an error flag, which if set upon return indicates an error occured during the

definition. If error is not set the structure or scalar is added to the vdef list. Note, it is possible to distinguish between the two types after the definition. If the type is a scalar, then members equals null, otherwise lower and upper bound are both null.

Scalar Definition

Scalar has a simple sequential flow. It looks for two integers or integer expressions separated by '..'. Integer expressions are handled by the module Expression, whose discussion is more appropriate later.

Structure Definition

Structure, like State, is also split into an iterative process over two choices object declarations and symbol strings. Object declarations may be recognized as the first token of them is a predefined or user defined class. In this case control passes to Object_Declaration, along with all the arguments Structure received to be completed. Object_declaration also receives the name of the structure being defined, and the file type STATE. The former is needed so the function Is_Class can recognize the current structure as such and allow appropriate pointers to be defined. The file type permits Object_Declaration and its sub-modules to adjust to minor semantic differences between a state and storage object declaration (e.g. initialization is not permitted in the state case).

If a recognizable class is not seen, then the rest of the program line will be parsed, and the token names will be concatenated to form an addition to members.

Object Declaration

Object_Declaration's main purpose is to verify receipt of a valid class with Is_Class. The construction '$logical name' may also provide a valid class. If the class abstraction operator, '$', is seen Logical_Name will be called to return the logical name following the '$'. Further discussion of Object_Declaration is delayed until Logical_Name is defined.

Logical Name

Logical_Name receives the dptr, an empty array (unless this is a recursive call), and the structure code discussed earlier. It proceed with an algorithm, generated by the language's syntax, to parse a logical name concatenating the tokens it receives onto the array. Logical name performs all the necessary conversions to proper C code, with the exception of array subscripts. A call to Expression fills code with the correct C syntax. The logical name in its interpreted form, along with the subscript translation are passed back to the calling module.

Object Declaration Revisited

Once the array containing the logical name is returned the function Class extracts the class of the name by parsing

the array in a similar manner, and compares the members of the name to defined structures. Class (the function) will return the class of the logical name, if one exists, and a value indicating the following conditions:

* 0 - the array is a valid, defined logical name (class=class)
* 1 - the logical name is not defined (class=null)
* 2 - the logical name contains a member which is not defined (class=the undefined member)

If no errors occur Object_Declaration makes repeated calls to Declarator, so long as each successful parse of Declarator is followed by the token ','. After normal termination of the Declarator calls, Object_Declaration returns to whomever called it (Structure or Storage) with the appropriate data areas, code and members, filled in. If an error occurs Object_Declaration will still return, but code and members will be incomplete, and will be rejected by the calling routine.

<u>Declarator</u>

Declarator parses and codes the declaration of one object including the optional initialization. Members, passed to Declarator by Object_Declaration, is filled in completely at this level. Code may also be completed, if initialization or array subscripting does not occur. Should such take place code is shipped off the Expression for completion. The file type, as mentioned above, is propagated to this level; hence, if a variable is initialized,

and the file type is STATE an error is generated.

Other arguments received by Declarator include dptr, the current structure being defined (if the route to this module was through Structure), and root. Though not used by this routine, root is needed by Expression, while the current structure is needed by the function Is_Class. Is_Class will be used in Declarator to determine when functions are being defined, as the usual indicator, a token '(', is not present in D.

Expression

Expression is the crux of the interpreter's code generation. Though it is only one out of the twenty-nine parsing modules it will create over half the code. Three of the four file types depend on Expression, yet its decomposition is basically a multiple if-elseif-else statement inside a while loop. Expression receives dptr, root, and code as its arguments, which it passes some or all to its worker modules.

Each of these workers is a different form of an expression, and will fill in the appropriate C code. Three of the six, namely Logical_Name, Func_or_Op_Name, and Parenthesis, call Expression recursively. The other three are Single_Quoted_Literal, Double_Qouted_Literal, and Accept. The most complicated of the six, Logical_Name, has already been discussed. The rest are simple enough to transfer directly into code; again their function is to parse and generate the

code for the respective expression type their name implies.

Storage

Moving back to the file level the next routine to be discussed is Storage. Storage is called by Interpret with the same arguments as State, namely cptr, dptr, and root. Its function is next to bundles in simplicity, while its data structures are the most complicated after the file record.

Storage consists of multiple iterations of Object_Declaration until the file's terminating period, or a file keyword is detected. Since Object_Declaration handles all error conditions Storage need only accept or reject the values returned depending on the condition of the error flag. If accepted the structure members is attacked by the function Find_Member which extracts the information recorded about one variable, function, or operator. The declared object is then added to the appropriate declaration list. Find_member is repeatedly called until all declared objects have been removed from the members structure.

Two types of declaration lists are part of the file record. The vdec list organizes declared variables, while the fodec list handles functions and operators. Vdec has the following composition: a name, a type (or class), two integers, and a pointer(s) to the next record in the list (tree). The integers in vdec are used to annotate the level of indirection of the variable; i.e. it is a count of the

number of pointer references must proceed it to get the objects value. The second integer indicates the dimentionality of the variable, whether it is a scalar (value = 0), an array (value = 1), a matrix (value = 2), etc. The names of these integer locations are pointer and array respectively.

The fodec (and fodef which is identical) structure is suited to the possible attributes a function may have namely the function or operator name, its input type, its output type, three structures of type vdec, a renaming tree, and appropriate pointers to the next structures.

The three vdec structures hold information on the function or operators arguments, and results. These may contain null values as arguments and results are not always required. It is admitted that the input and output types are not needed as this information would also be carried in the vdec structures; however, they are added to facilitate the module Expression in its type checking. While the rename tree is needed for the local remaning of objects within a function's or operator's scope.

The functions which add the declared objects to their respective lists are patterned in the same manner as those used in manipulating the file structure, and the vedf (variables defined) structure.

Actor

The final subgroup of modules under Interpret to be

discussed are those whose functions are needed to parse and code the actor file. The routine which controls the sequencing is Actor. Actor is responsible for an iterative calling of either Function_Declaration or Operator_Declaration, Linkage, and Block, in the named order. Its other responsibilities include writing the code produced by the declaration routines, and producing the C function shell:

```
func_type func_name(argument_list)
argument_declaration;
{ variables;  /* provided by the share or copy */
  C code      /* produced by Block             */
}             /* ending right brace            */
```

Function Declaration

Function_Declaration controls the calls to Declarator as it sees the keywords in and out. The receipt of the keywords is not actually necessary as the system is already aware of the types of the arguments and results through the information received in the storage file, wherein the function or operator was declared. The program will key on these special tokens (in and out) though in order to maintain consistency with the rest of the system.

Declarator will be called in the same manner as Object_ Declaration called it. The difference lies in that Function_ Declaration will not iterate over the call.

Function_Declaration will also handle the placing of the locally declared input variable's information into the functions fodef.argl node. The fodef node has the same structure as the fodec node defined under the subsection

Storage. In addition, the functions result will, in a like manner, be placed in fodef.result. Note that the routine Find_Member will be utilized, as it was under the control of Storage, to accomplish these tasks.

Operator Declaration

Operator_Declaration accomplishes the same process that Function_Declaration does. Its difference lies in that it must be able to handle the addition of another input variable. Following the syntax of D a minimum of two calls to Declarator must take place (minimum of one for in, and one for out). Like its counterpart for functions Operator_Declaration will set the vdec nodes in fodef, and will return the appropriate C code to Actor.

Actor's Linkage

While Actor will call the same Linkage which Interpret calls, the type set will be different from the file types.

For example, if the type is integer then bundle, state, storage, and actor would be one through four respectively. The file type specified by the Actor call to Linkage could then be five to differentiate it from the files.

This is so that the Share and Copy functions may not be used but the code for renaming need not be duplicated.

Block

The module Block handles all the sequencing of the statement types, along with the optional guard and continua-

tion characters enclosing the statement. Its structure deviates from the modeling of D's, because it is here that Wirth's second restriction (mentioned on page IV-8) fails to apply. Therefore, guard and statement become intermingled.

Block's flow of control is to linkage first, whereupon local variables are generated using the code sections of the state and storage file records. Completing this, Block expects zero or more iterations of the sequence guard, statement, continuation. The end two of which may be empty irrespective of the other. Studying D's syntax one notes that guard can share the same first action as statement, that of needing an expression. This is what leads to the structure aberation.

If the code following the return of Linkage, or the parsing and code generation of a line, is not an expression indicated by the token "return", "new", or "{", then the respective function Return, New, or Block is called. Should the token "$" follow the return of Expression, Block will expect one of the four statement types, otherwise Block realizes it parsed and coded a statement which was an expression, and skips the section of code which requires a statement.

Block then parses and codes the continuation, if it exists. Once completed it loops back to the check for a guard or statement, and continues this process until it detects the token "}" signifying the end of the block.

Return and New

Return and new generate, and print the C code necessary to perform these functions. They are both simple enough to code directly, because the majority of the code produced is by a call to Expression.

Error

The majority of code this program will deal with will contain errors; therefore the error handling should be worked into the design. Basically there are two actions the error routine produces. The first is the writing of error messages. For this operation Error needs three items of information: the line number where the error was detected, the token on which the error was detected, and the type of error committed. The form of this operation is a simple case statement.

The second action Error must complete is considerably more difficult. This is the restart action; Error must read through the D code and detect where is appropriate to start the parsing of the program again. The actual decision on what restart symbols (or combination of symbols) are acceptable will be handled be that section of the code which detected the error. Therefore the fourth argument passed to Error will be the restart action to be taken. These actions may range from 'do nothing' to 'quit the program'.

Other Functions

The remaining functions were implicitly or directly mentioned throughout the design, but their discussion has been delayed until now. This was done, because the injection of these functions would have fragmented the discussion of the design.

C unlike many structured languages is unstructured in its organization of modules permitting them to be easily accessed from anywhere in the system. This unstructured approach allows the creation of a new mindset for the designer. One may see C as having two distinct function types, those that control the program sequencing, and those that do not. The latter group may be thought of as tools providing, in a sense, macro statements within the language. The algorithms these tools use are not important so long as their inputs and outputs are clearly defined.

For example, the function getc(ch), in C, assigns ch the value of a character, and returns the integer representation of that character (or -1 for end of file) each time it is called. The program using getc need not know how getc links to the operating system on order to get the value, only the task that it accomplishes. Therefore, the tools in this system, which operate on the structures built, will be defined in terms of their inputs and outputs only.

* name_change - accepts two character arrays. The function copies whatever is in the second array to the first. If the second array ended with the character sequence '.d' the 'd' will be changed to a 'c' in the

first array, otherwise a '.c' will be concatenated to end of the first array.

* is_file - accepts a character string as input and determines if the string is one of the file keywords. If it is the function returns a one, otherwise it returns a zero.

* in_list - accepts a character string as a name, and the root of the file record structure. It will search the structure for a file of the given name, and return one if it found it and zero if it did not.

* put_in_file_list - accepts the root of the file record structure, a file's name, and its type. It will create a new node for the named file in the proper alphebetical order. All other sebsections of the file node are set to null, and will be filled in bey other tools. This function returns the root of the list. This is handy if the initial list is a null pointer.

* add_rename - takes the root of the renaming list and a pair of names, along with a pointer to an integer. It adds the names to the renaming list by the alphebetical ordering of the second name, and returns the root of the list. If a collision occurs, that is to say the second name exists in the list already, the integer that is pointed to is set ot one, and the names are not added to the list.

* find_member - accepts three character strings, and three pointers to integers, along with a structure of type members (see pg IV-11). The strings are member, the onput and output classes respectively, while the integers pointed to are pointer, array, and function respectively. Each variable (whether string or integer) is filled in with the appropriate information of the object named member. This function returns members, altered by writing spaces over the information extracted, if a member is found, and null if not

* add_class - takes the root of a vdef list, the class' name, the class's members structure, a lower and upper bound. It adds the class by name alphebetically to the vdef list.

* add_var - takes the root of a vdec list, the variables' name, its class (both character strings), its level of indirection, its dimentionality (both integers), and adds the variable by name alphebetically to the vdec list.

* add_fo - takes the root of a fodec or fodef list, a function or operator name, and the input and output classes (the last three being character strings), and adds the function or operator to the appropriate list alphebetically by name first, then by input class, then by output class.

The above three functions also receive a pointer to an integer as the last argument, which gets set if the named objects already exists in the list it is being added to. In addition, all three also return the root of the list (why? see put_in_ file_list).

* get_file - accepts a file's name, and the root of the file record structure. It returns a pointer to the named file's node or a null depending on whether it matched the name with one in the list, or not.

* is_class - accepts a name, a tempory class (both character strings), and the root of a vdef list. If the name is the temporary class, "null", "int", or matches any of the names in the vdef list the value returned is one, otherwise zero is returned.

* find_class - accepts a name, and the root of a vdef list, and returns a pointer to a vdef node with the name portion the same as the input name, or null depending on whether a match was found, or not.

* find_fo - accepts a name, and the root of fodef or fodec list, and returns a pointer to the appropriate type node with the name portion the same as the input name, or null depending on whether a match was made, or not.

* find_var - accepts a name, and the root of a vdec list, and returns a pointer to a vdec node with the name portion the same as the input name, or null depending on whether a match was found, or not.
* class - accepts a logical name with C's syntax, and a character string called class. It extracts the class of the logical name by parsing the array, and comparing the members of the name to defined structures. Class (the function) will return the class of the logical name, if one exists, and a value indicating the following conditions:
  * 0 - the array is a valid, defined logical name (class=class)

* 1 - the logical name is not defined (class=null)
* 2 - the logical name contains a member which is not defined (class=the undefined member)

* add_label - accepts a name, an integer, and the root of the lable list, and adds the name to the list with its integer by alphebetic ordering. The integer is the level of the label (the name). The function returns the root of the label list.

* is_label - accepts a label name, a level, and the root of the label list. This function returns a one if it finds a match of the label name with those in the list and the level is greater than or equal to the level of the matched name, otherwise a zero is returned.

* is_continuation - accepts a character string, and returns one if that string is ";", "..", or "\", otherwise the function returns a zero.

Conclusion

An overall picture of the program design is provided in Figure 6 to give the reader a better feel for the program structure. The tool functions are not included as their representation would clutter the picture. Figure 6 broken into multiple sections, including the tool functions, is available in Appendix E.

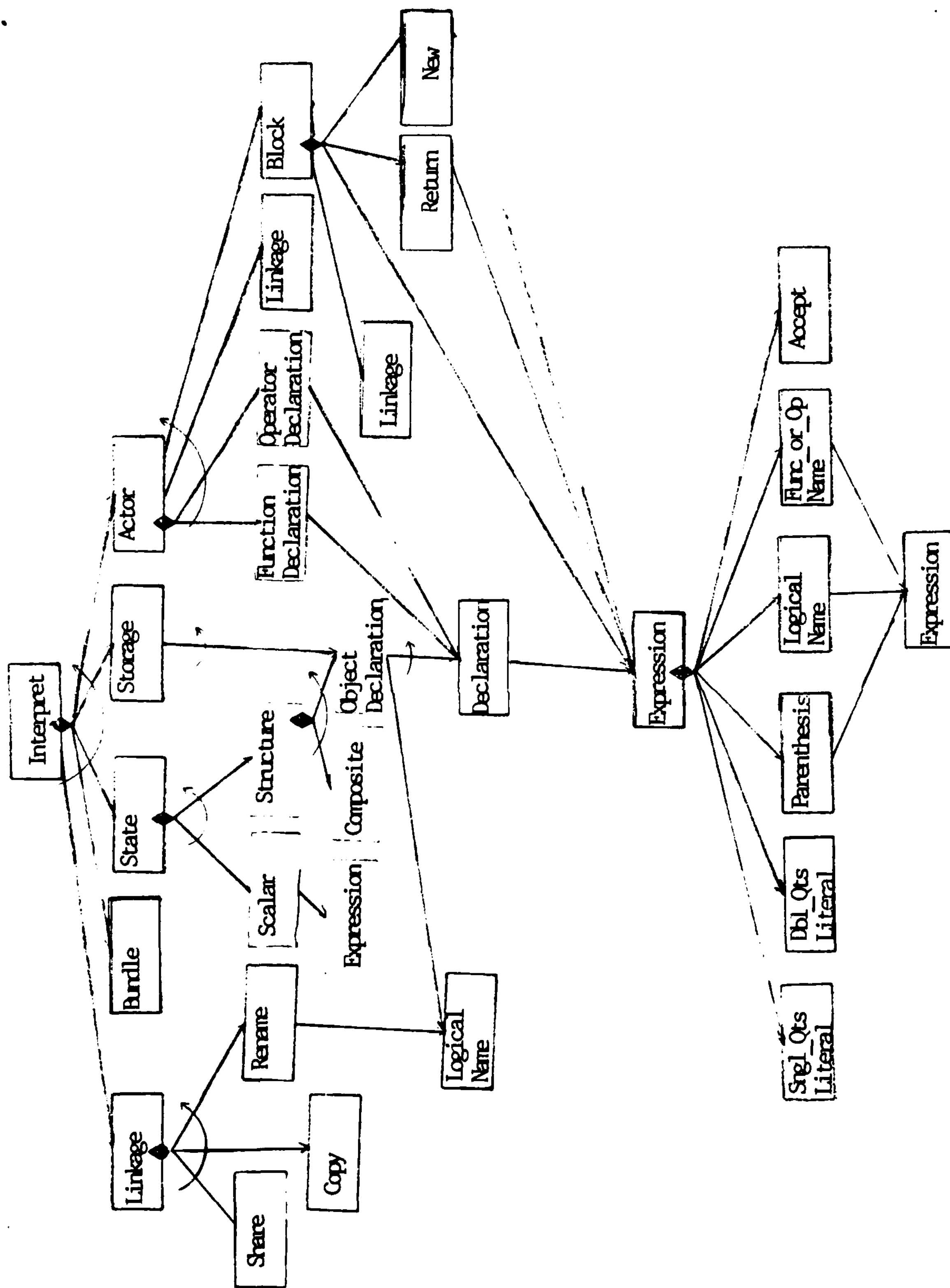


Figure 6: The Program Control Structure

V. Results and Analysis

This section seeks to answer three questions: what are the D language's strengths/weaknesses; how was D changed during implementation; and how well does the interpreter meet the expected goals.

An Analysis of D

As stated in earlier sections the purpose of D as a programming language was twofold. Capt. Jennings believed C was somewhat unstructured, limited in its algorithim specification, and nonexistent in its ability to control parallel processing; hence, he attempted to inprove upon C. In addition, he found Ada had all the proper enhancments, but in an overly large language. His solution was to combine ideas from the two, hopefully resulting in a language resembling C in programming simplicity, and Ada in programming power. This goal was basically met, in that generic algorithm specification, parallel processing, and structured programming were achieved, while maintaining a fairly compact language. Although, in an effort to keep the language brief certain needed or desirable traits were omitted; these form the basis for D's flaws.

Weaknesses

One of the most severly chopped sections of the language was its predefined classes. Integer types must necessarily be defined in order for all numeric types following

to be elaborated. As in Ada the integer class is sufficient if enumeration types are a part of the language. The enumeration types permit a mapping of integer to character. Unfortunately enumeration types are not part of the D language; therefore, the relationship between integers and characters must be an abstract one formed by the user. Not only is this cumbersome for the user, but even worse, the character string literals cannot be used, because the declaration of a variable to relate to them is not possible. This problem was probably an oversight, rather than a design mistake, and is the only problem which forces a change.

The other weaknesses do not restrain the user to a reduced functionality, but they might hinder a programmer from accepting D. These nuisances are limited argument passing and awkward type construction.

In every commonly used HOL functions, or procedures, will accept more than one argument. The restriction allowing a maximum of one argument to D functions, while simplifying the interpreter, forces the user to define a greater number of structures than is usually necessary. Many of these structures will very nearly be duplicates of each other. On the other hand, it could result in the definition of a few structures, but allow the passing of unneeded variables.

For example, given Function A needs variables x, y, and z to operate, Function B needs variables x, y, and w, and

Function C needs only x and y, three structures need to be created. A programmer wishing to avoid this could create one structure containing x, y, w, and z, but this would violate a software engineering principle: do not allow a module to access data, unless it needs it. While the creation of three structures is a trival case, the problem becomes increasingly menacing when the combinations of four, five or a greater number of arguments are used.

The second undesirable quality of D is that class construction is not entirely contained within the state definition file. In order to create a n-dimensional array one must also create n-1 other arrays their dimensions running from n-1 down to 1.

For example, consider the program which has as data element a 10x20x5 space of integers. The declaration is as follows:

```
int : x[10]
$x  : y[20]
$y  : space[5]
```

The intermediate variables x and y were necessary to achieve the proper declaration. They are instantiated; therefore, memory space is allocated for them, yet they may it is possible they are never needed. While runtime memory is rarely a problem with many systems today, this method of declaring variables is wastefull. Furthermore this problem is not limited to the declaration of multidimensional arrays, but also includes declarations of pointers which differ from

default constructs, e.g. a pointer to an array.

Though possibly a problem it is, perhaps, only a misunderstanding. The '?' punctuation preceeding a block appears to allow a race condition to occur, concerning the control of a block following guard evaluation (see ref 5 pg 59). Though this anomaly was detected early, no contact was made with Capt. Jennings to clarify the matter; hence, '?' has been left unimplemented.

Finally the accept function is ill defined. One attempts to compare it with Ada'a accept statement, yet the definitions do not seem to agree. The definition has been left open for further work, because C does not provide facilities for tasking. If a future implementation translates D into Ada the accept statement could then be strictly defined. For this version a D program containing the accept construct will not be tagged as wrong, though a message will be generated indicating the function is not implemented.

Strengths

While D contains flaws in its object definition and declaration, it is flexible and strong in its algorithim specification. The actor file, barring those problems mentioned above, is well defined. Its increased functionality ove most languages provides for generic algorithms, and parallel expression evaluation. Furthermore, since an expression may be a function call, D actually provides a true multitasking capability.

The addition of the operator actor, combined with the ability to overload operator names, creates very readable code using, on most occasions, standard mathematical prefix, infix, and suffix operators.

For example an operator could be defined, such that, the expression n! would be understood by anyone familiar with factorial notation. Hence, a programmer supplied with the necessary operator packages could implement fomulas in the same manner with which they appear in textbooks.

Finally overloading of function or operator names, combined with generic specification, is a powerful tool for algorithim representation.

Changes to D

As introduced, D was not fully developed upon the completion of Capt. Jennings thesis; therefore, enhancements (mostly completed definitions) have been added to the language as it reached the implementation phase.

The addition of the predefined class 'char' was added for reasons discussed under the subsection Weaknesses. With this addition, and the defining of quoted literal string as contants of type char it becomes possible to use the latter in expressions. Enumeration types would also have provided a solution the problem, but this would have extended the language beyond its original boundries: a property desirably maintained for the purpose of analyzing D.

In order to simplyfy the interpreters task and provide

a greater degree of semantic error checking, an order was imposed on files, object, and actors (functions and operators). Files must be declared in a bundle file before they are used. Bundle files themselves are excepted; they and their collection of files must be defined before inclusion in another bundle's linkage or file list. The order in which an object or actor may appear within a D program is through declaration (storage file) then use (actor file). The classes or types, except those predefined, must appear in a state file prior to their use in a storage or actor file.

With the enforcement of a file order arose the need to define a 'null' class for actors which do not accept nor return values. Otherwise the declaration of such actors would be indistinguishable from that of an object. An illustration will better clarify this idea.

```
                              OLD
int  : get_number    ~~ could be a function returning an
                     ~~ integer or an integer object

                              NEW
int  : get_number null    ~~ a function returning an in-
                          ~~ teger while accepting no input
null : put_number int     ~~ a function accepting an
                          ~~ integer and returning no value
null : some_action null   ~~ a function which neither ac-
                          ~~ cepts input nor returns output
```

Finally the definitions of sharing, copying, and renaming were strengthend to include a semi-global cability when used in a bundle file. While an addition of renaming following an actor specification, alluded to on page 40 refer-

ence 5, yet not defined nor allowed in the syntax, provides a local renaming function.

A share, copy, or rename within a bundle file's linkage performs its respective function on all files subsequently named in the bundle file's body. While the local renaming will accomplish its function with object and class names only within the scope of its enclosing actor's definition.

The complete set of usage rules for all changes to the D language are fully integreted with the language description given in Section Three.

Function vs. Specification

Use of the interpreter built will quickly reveal the system is incomplete, yet many of the stated goals have been achieved. One of these goals is the error checking and recovery. The interpreter will detect any syntax error present, provided a previous error recovery attempt does not skip the interpretation of code, or derail the system into creating erronious errors. At a minimum then at least one valid error will be generated with each run of the interpreter (if the code contains errors). Furthermore, the semantic error checking includes: class checking, verification of declared variables, check of members within structures, and redefinition or redeclaration of files, classes, and objects. The only common semantic check, done by most compilers, not achieved is type checking of expressions. This was not implemented, because a full implementation of the

storage file is yet to be completed.

Of D's four file types two have been entirely implemented (including code generation), one is about 85% done, while the fourth remains only 50-55% complete.

The bundle file is one of the two which is fully operational. It does not generate code as its function is to group D files together into related packages; which is an unknown function in C. The purpose of the D file, initially, was thought to provide the Unix include facility within the language itself; although, after study of the Ada language it became evident that bundle file was in packaging D's files logically. This implementation accomplishes that function.

The state file is the other of the fully implemented files. Though it cannot provide the same function as intended, because of C's shortcommings, the restrictions are nominal. C defines all its functions as external; the functions therefore are globally accessable. This requires that the definition of the function's output class also be globally assessable. Hence all structure definitions must be coded in C as external to every function. This is not as restrictive as it appears. It removes the possibility of overloading class names within different state files, but with the renaming function name collisions may be quickly dispached.

Other than this restriction the interpreter will pro-

duce all the necessary C code in the form of typedef constructs. In addition, it will create the necessary structures for storing the class information in the event the information is required by the other files.

The storage file is the first of the incomplete D files. It is incomplete, because the structures used to define declared variables are not sufficient to hold all the possible combinations with which a variable may be declared. While it is possible to declare and receive the proper code for nearly every variable one could imagine, there still exist a few for which this is not possible, e.g. a pointer to a pointer to an array. This defficiency stems from the impotency of the state file to define all classes.

In addition, the present code generated by the state file produces external variables. This action voids the effectiveness of the copy function. An easy solution for this problem is available once the proper structures needed for the storage of such variables are defined. This solution will be discussed in the concluding remarks as a guide to further work.

Lastly the actor file is the farthest from its final form. Its working functions include parsing of the D code with error checking, and creation of the function specifications in C code. It is here that the type checking need yet to be implemented. Within this file lies the strength of D. It must be opened by interpretation to determine if the

promises of a better language are true.

Other actions which were planned at the start of this project, but left for later work due to time constraints, were the library of D functions, the data dictionary, and the mapping of C reserved words and operator names into the artifical name set (three underscores). In addition, the possible command line options mentioned in the requirements await the implementation of the full interpreter.

VI. Conclusion

The prime motivation for this thesis was the implementation of the D language. Since this purpose was not completed the conclusion seeks to provide instruction to any follow on work. Concrete statements cannot be supplied as to the worth of D at this stage, but this thesis brings us one step closer to the answer. D shows promise, and deserves a chance at life.

Following on

First of all it should be noted that sufficient work remains to provide the basis for another thesis. The actor file's code generation and the variable declaration structures need to be implemented. Additionally, once accomplished a dynamic testing of D should be performed to verify statements made both in this thesis and Capt Jennings' as to the languages utility in a world glutted with means for algorithim expression.

Do It In Ada

In order to accomplish such testing it is recommended the future student rework the code generation to produce Ada code. Within the final month of this report an unvalidated version of Ada was installed on the Unix Vax computer. It is projected that within the next few months (approx. Feb-Mar 85) a fully validated Ada compiler will be available. Since Ada permits tasking, overloading of operator and func-

tion names, sub ranges, and generic algorithm specification, in addition to all functions which C provides, a full implementation of D would be possible. This is much preferred over a subset as a true analysis of D could take place. Furthermore, little time would be lost as only the state file's code generation is complete.

The Copy Problem Solved

In the event further work continues to map D into C the global variable declaration problem, mentioned in Section Five under storage files, can be solved.

If one separates the actor declarations from the variable declarations the variables may then be bound into a structure. Referencing for variables is then accomplished by 'storage_file_name.declared_variable'. This creates the unfortunate restriction of allowing six (seven if no more than 30-40 copies are needed) significant characters in the storage file name. This would supply a means of diferentiation between the multiple copies by the addition of two (one) characters following the first six (seven) characters in the storage file name. Additional functions would have to be created to handle the naming irregularity.

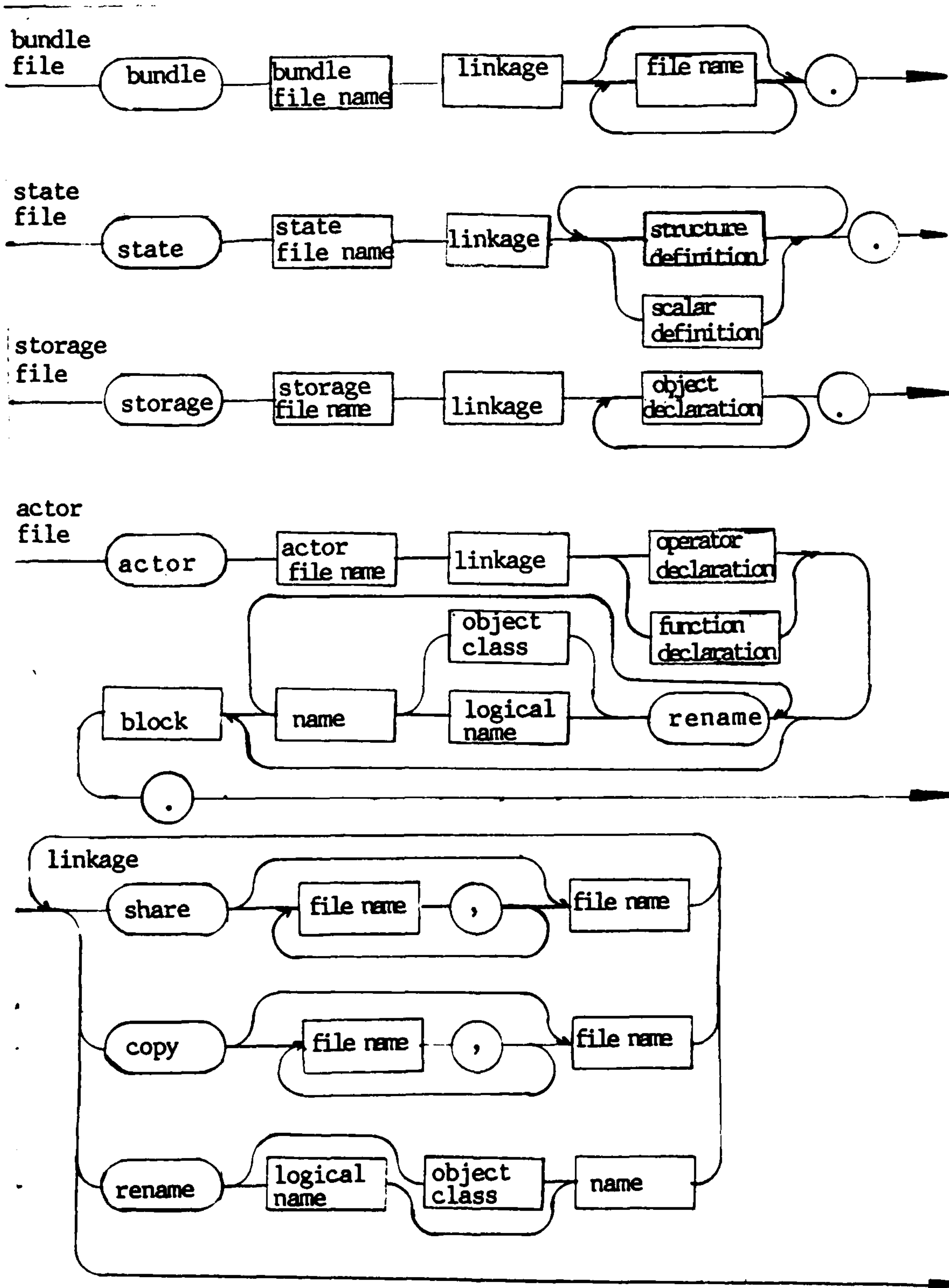
Generics in C

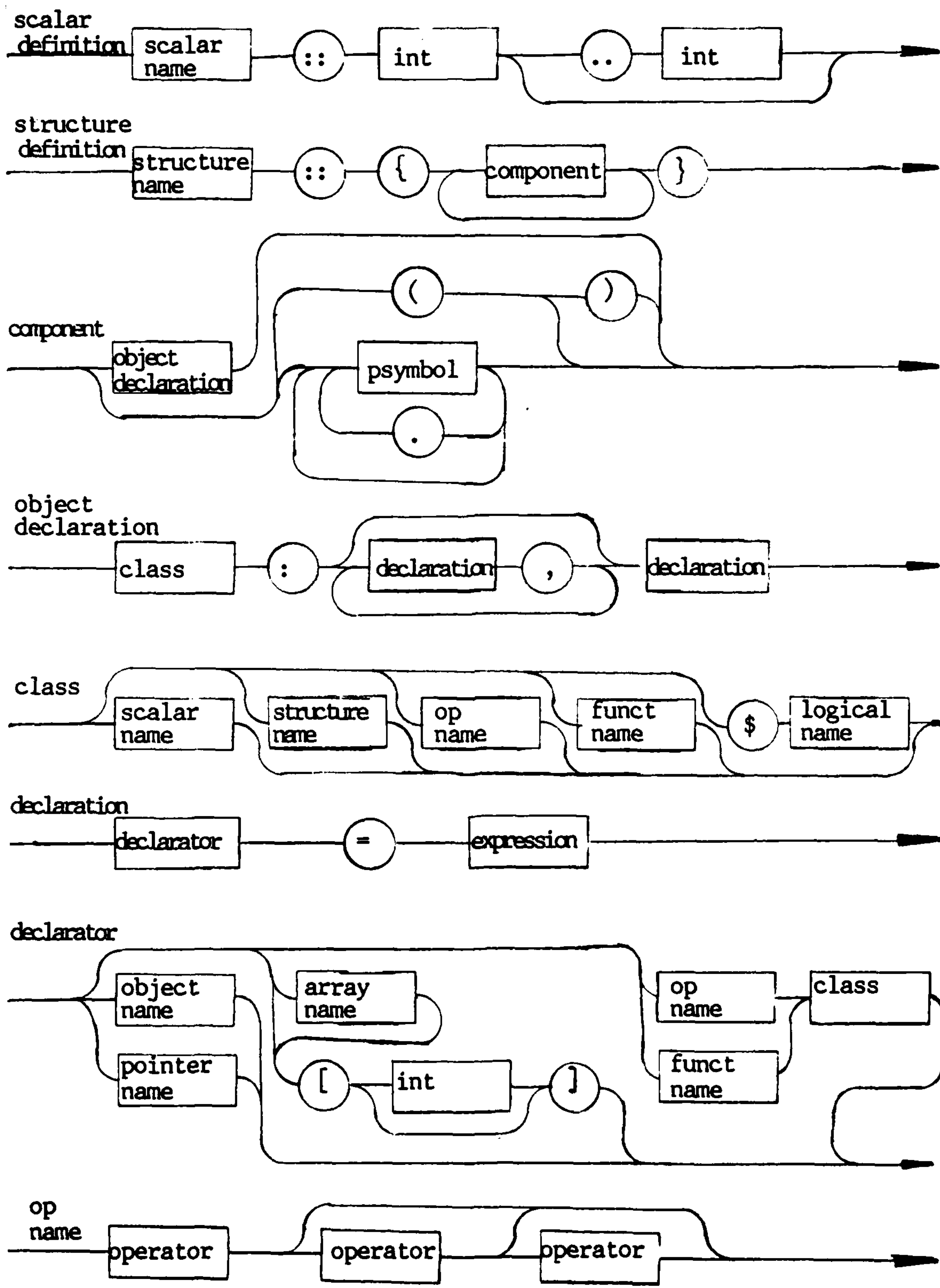
Though generic algorithims are not permitted in C it is possible to interpret the D program such that generics are supported. To do so the program must keep track of all

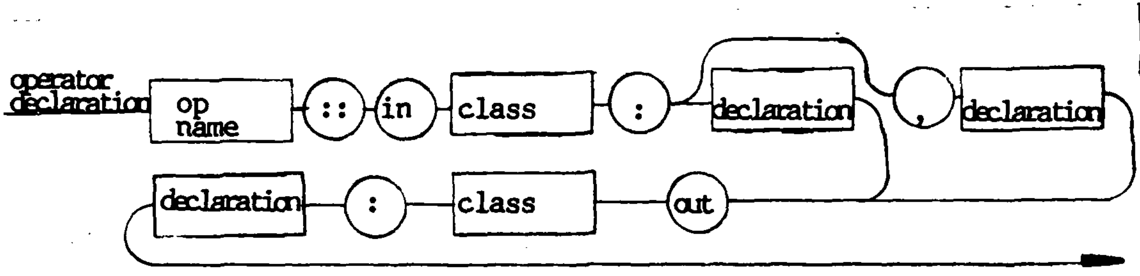
different class types which use the actor. The interpreter then generates the required number of copies in C code allowing for the differences in types. Again a naming function, as with copied variables, will be needed to prevent name collisions in the C code.

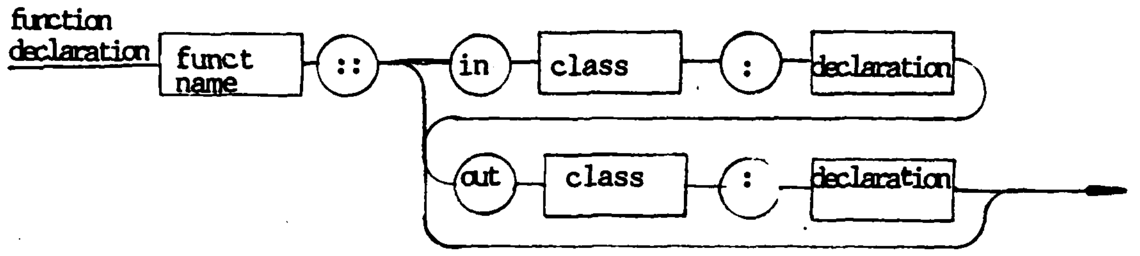
In summation follow on work is necessary for the full impact and worth of the D language, and this thesis, to be realized.

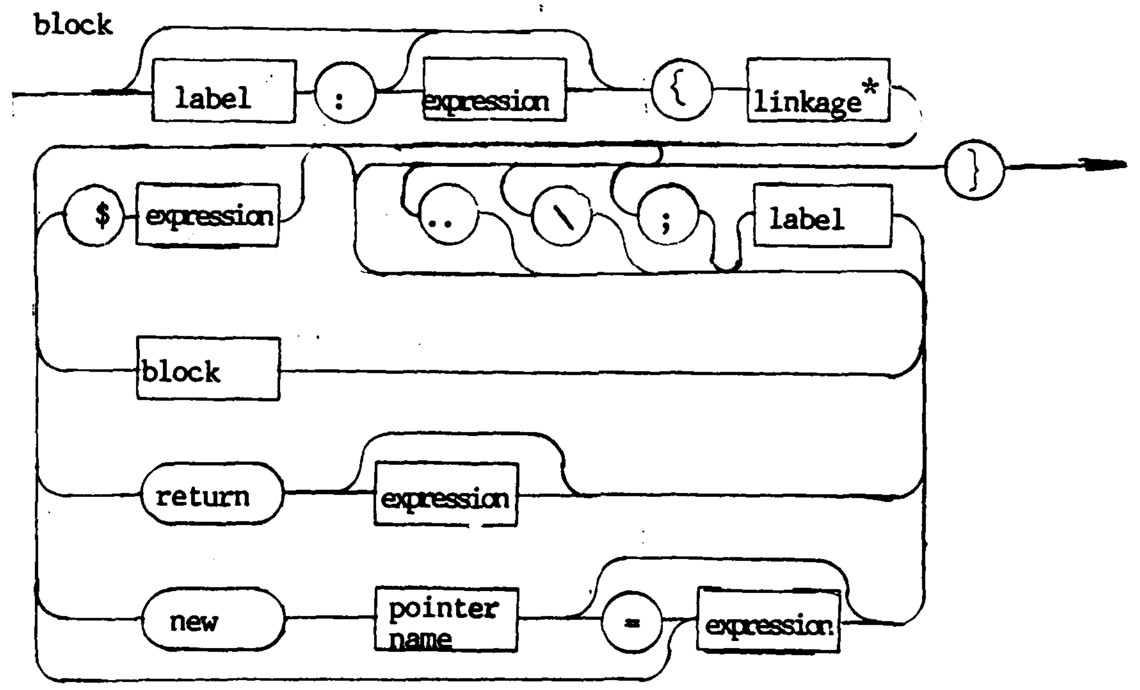
Appendix A: D Syntax Diagrams

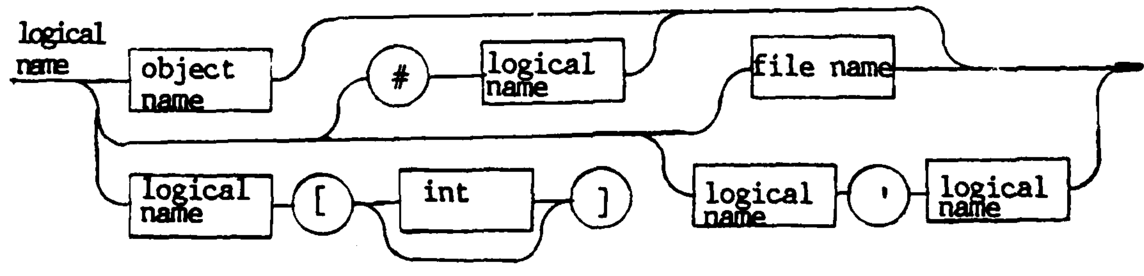



scalar definition
scalar name
::
int
..
int
structure definition
structure name
::
{
component
}
component
object declaration
(
)
psymbol
.
object declaration
class
:
declaration
,
declaration
class
scalar name
structure name
op name
funct name
$
logical name
declaration
declarator
=
expression
declarator
object name
pointer name
array name
[
int
]
op name
funct name
class
op name
operator
operator
operator

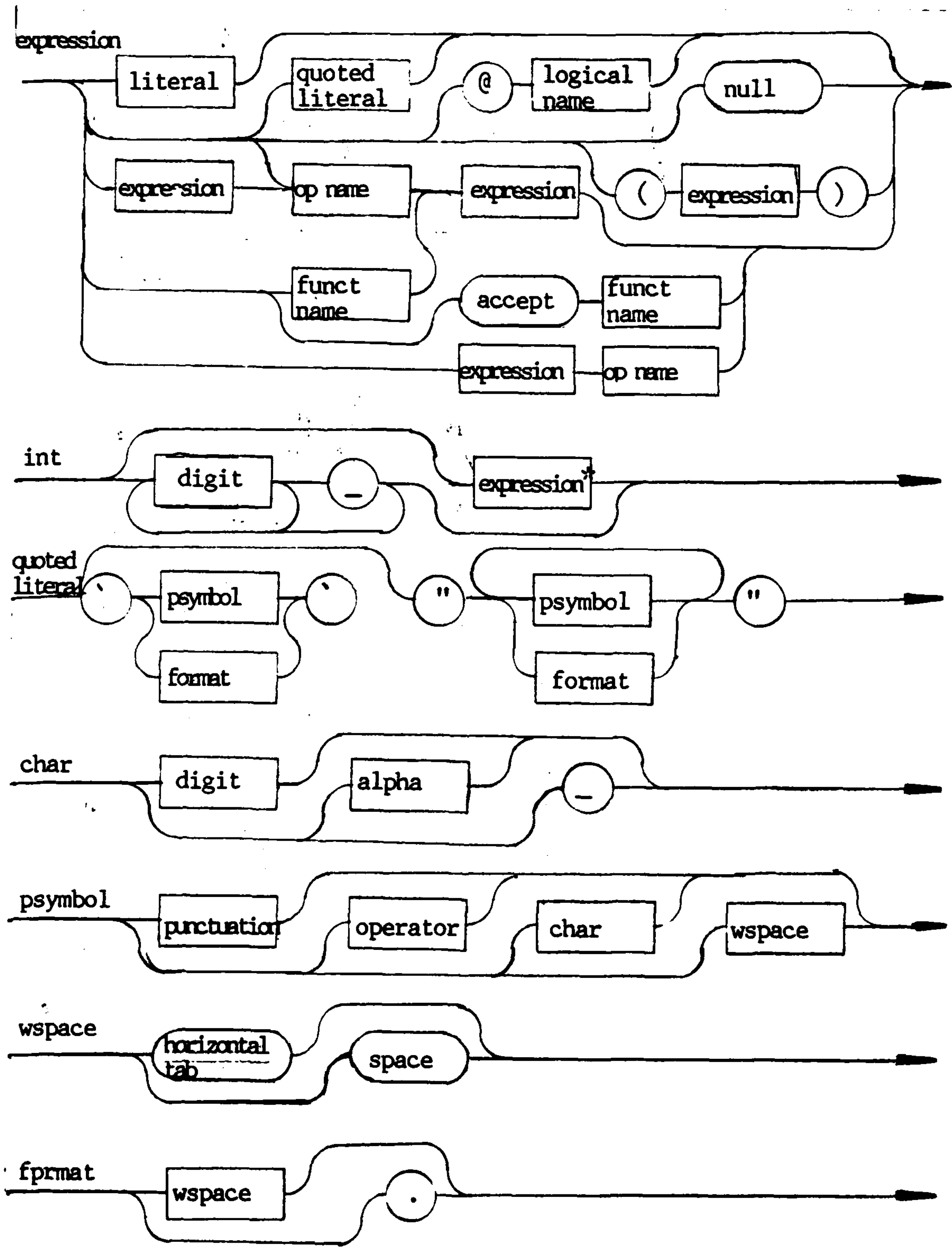




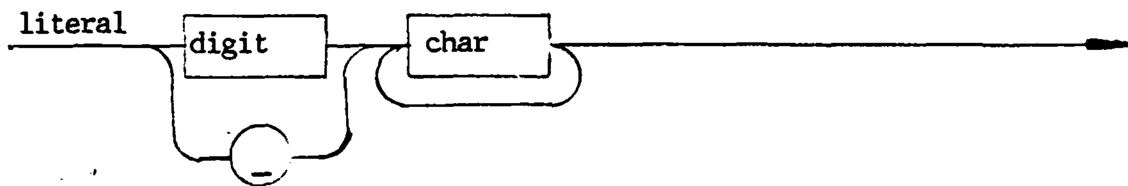


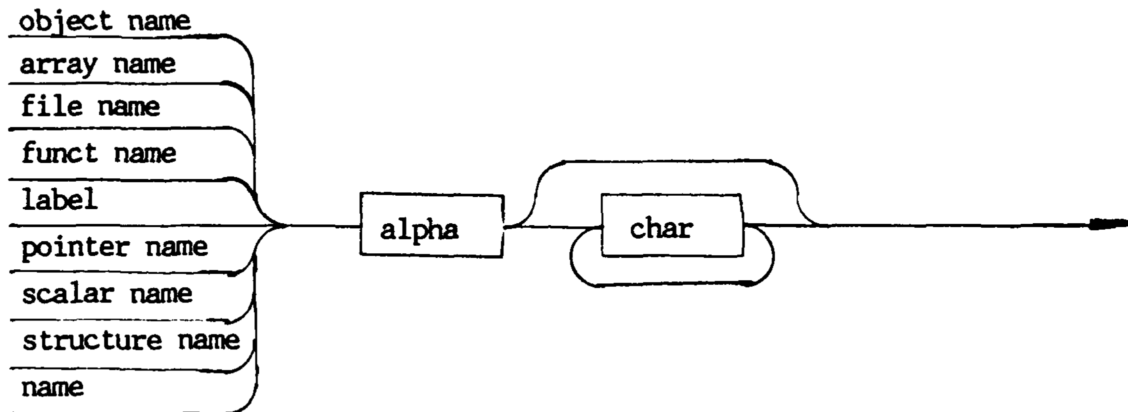


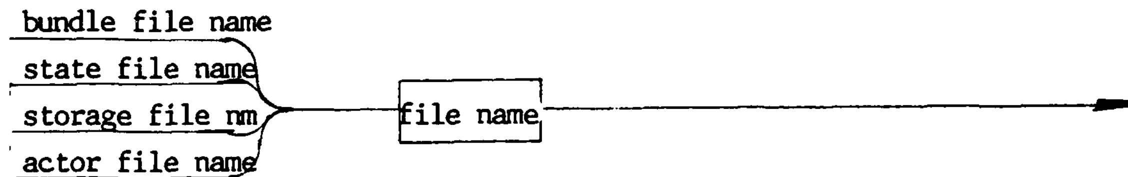
*renaming not permitted here



*must evaluate to an integer







alpha = A - Z and a - z

digit = 0 - 9

punctuation = [,], (,), ,, :, ::, ', ", @, ?, #, $, ;, ..,
{, }, `, \, ~~

operator = !, *, /, %, +, -, &, |, <, >, =, ~, ^

Appendix B: Keywords and Operators

***** Keywords *****

accept	actor	bundle	copy	in	new	null
out	rename	return	share	state	storage	

***** Operators *****

operator	function
!	user defined
^	" "
*	" "
/	" "
~	" "
%	" "
+	integer addition and user defined
-	integer subtraction and user defined
&	user defined
|	" "
<	" "
>	" "
=	assignment

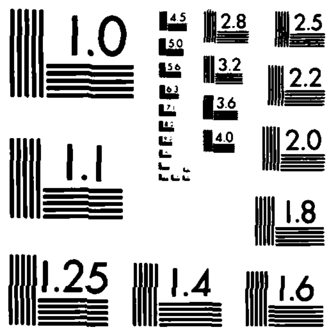
combination of up to three operator symbols is possible; precedence then follows the rule for the right most symbol

precedence is from the top of the operator listing (having the highest precedence) to the bottom; each two symbols starting at the top and continuing in sets of two have the same precedence

AD-A152 105 A D TO C INTERPRETER(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING K J SHOMPER DEC 84 AFIT/GCS/ENG/84D-27 2/2

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC



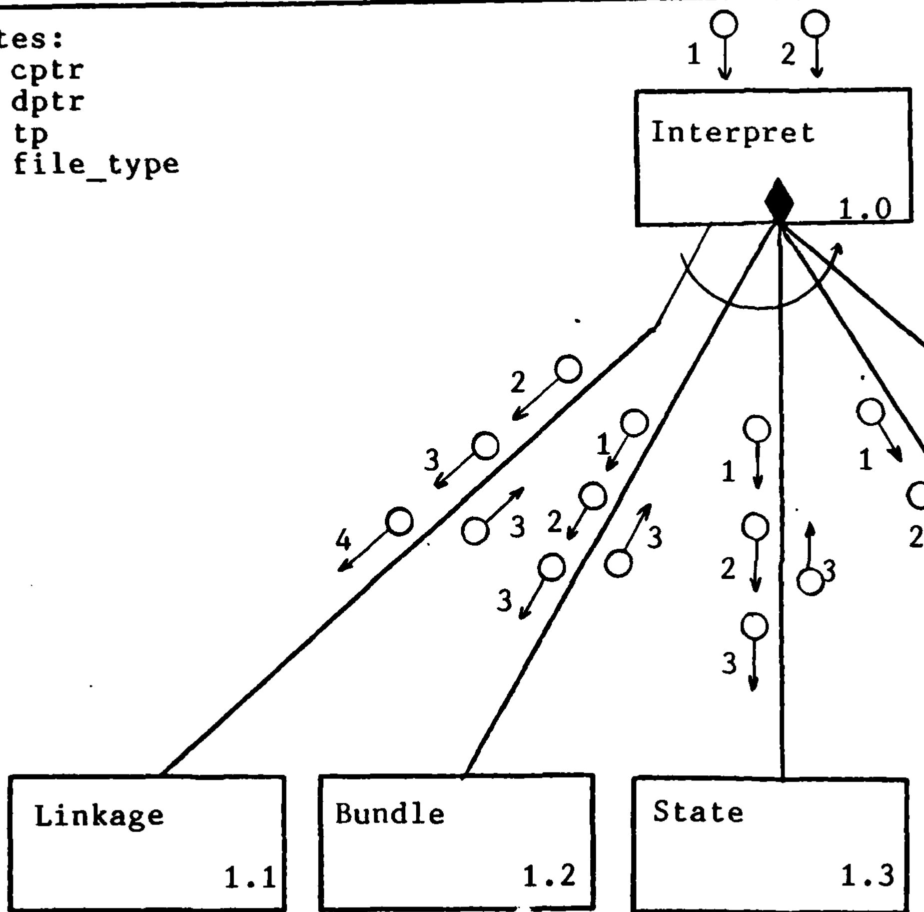
MICROCOPY RESOLUTION TEST CHART

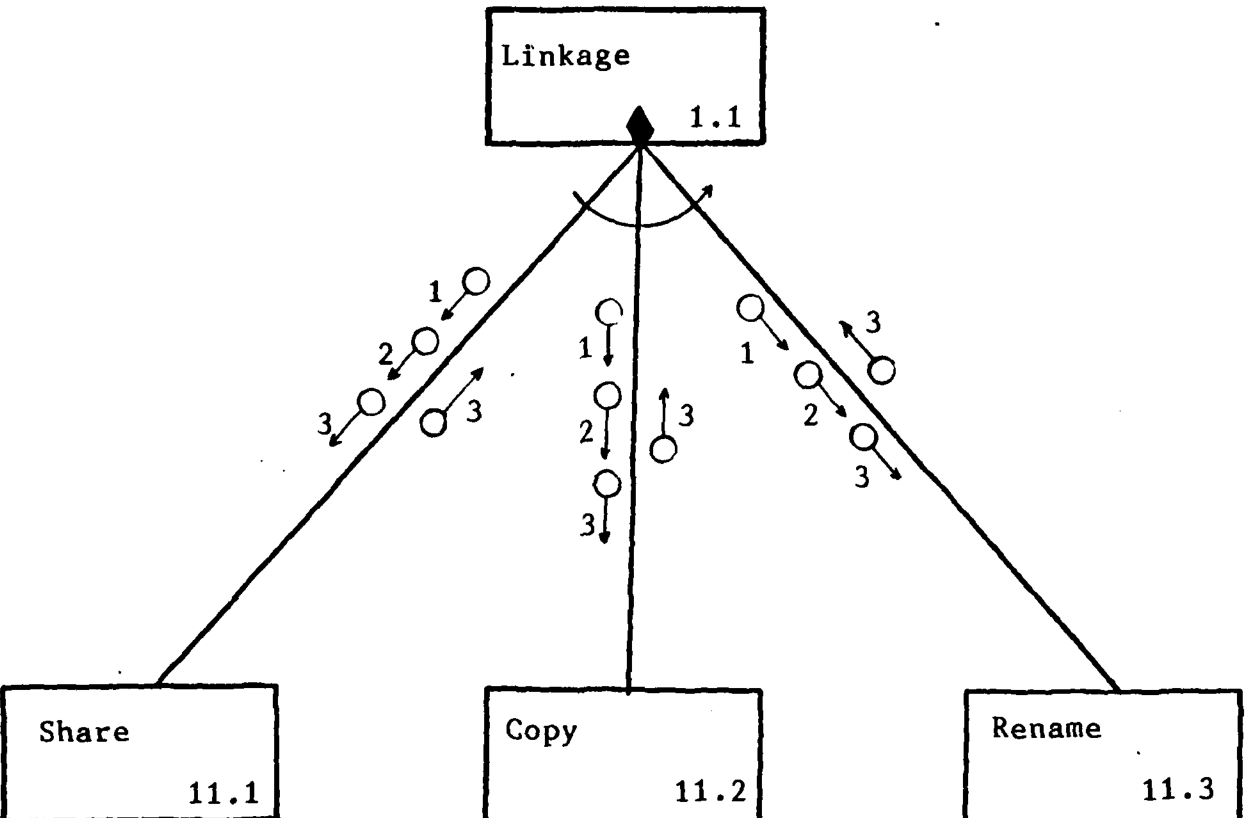
ARDS-1963-A

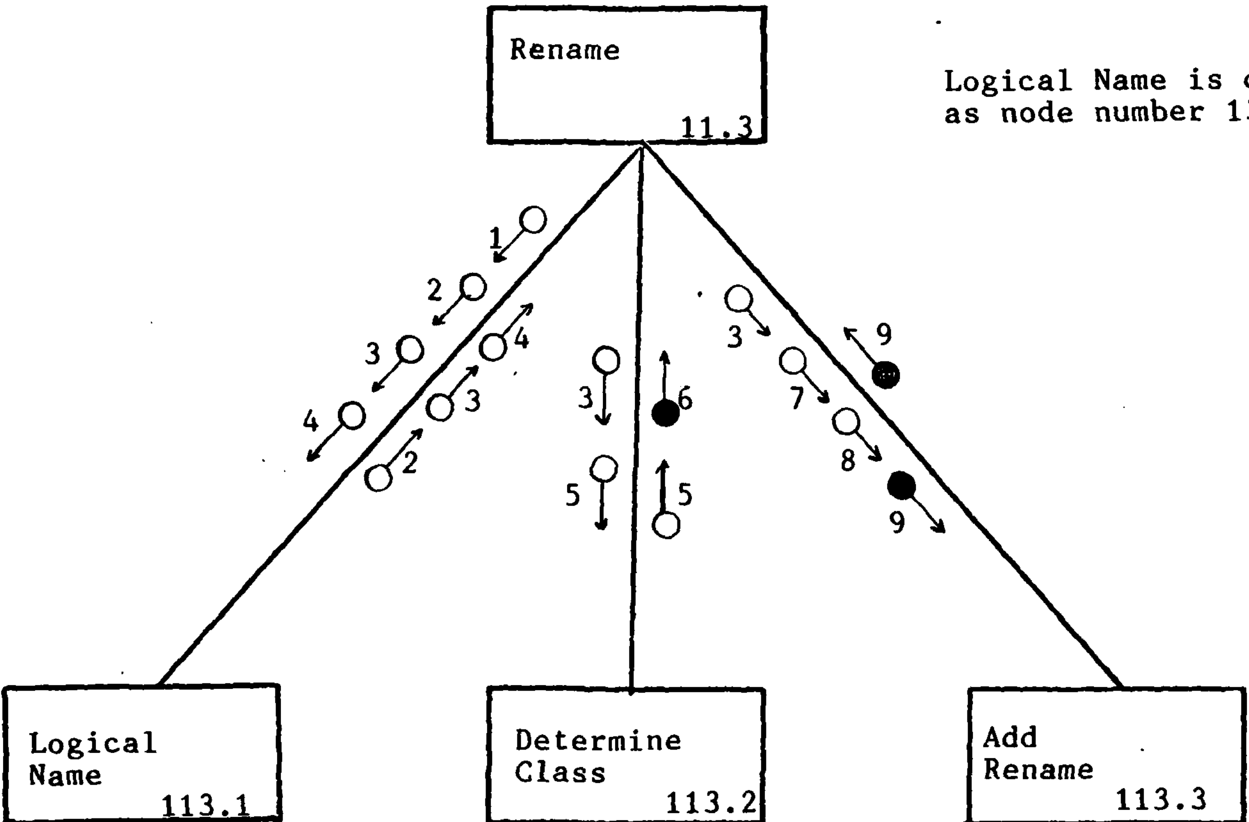
Appendix C: Design Structure Charts

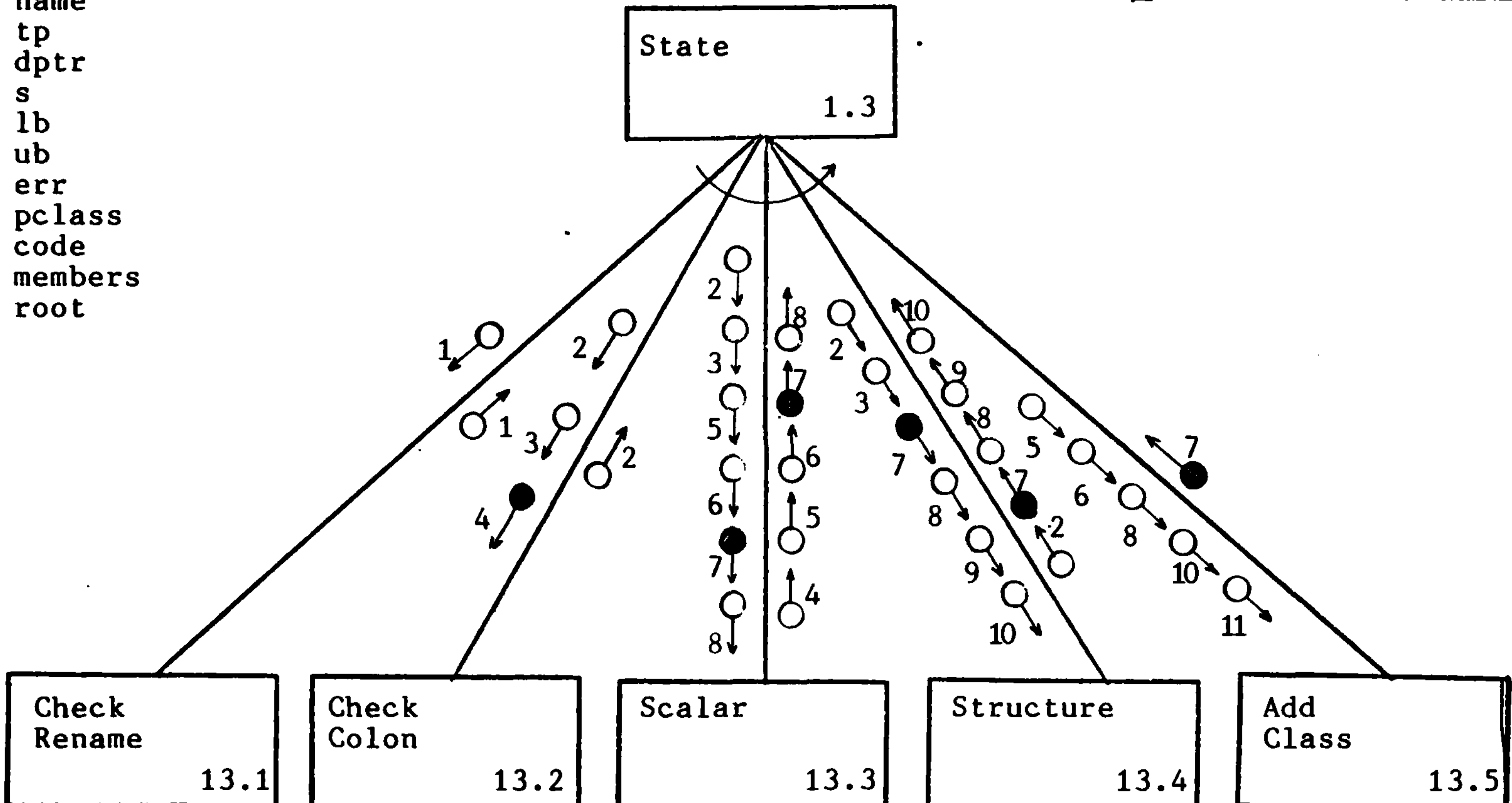
Node Index

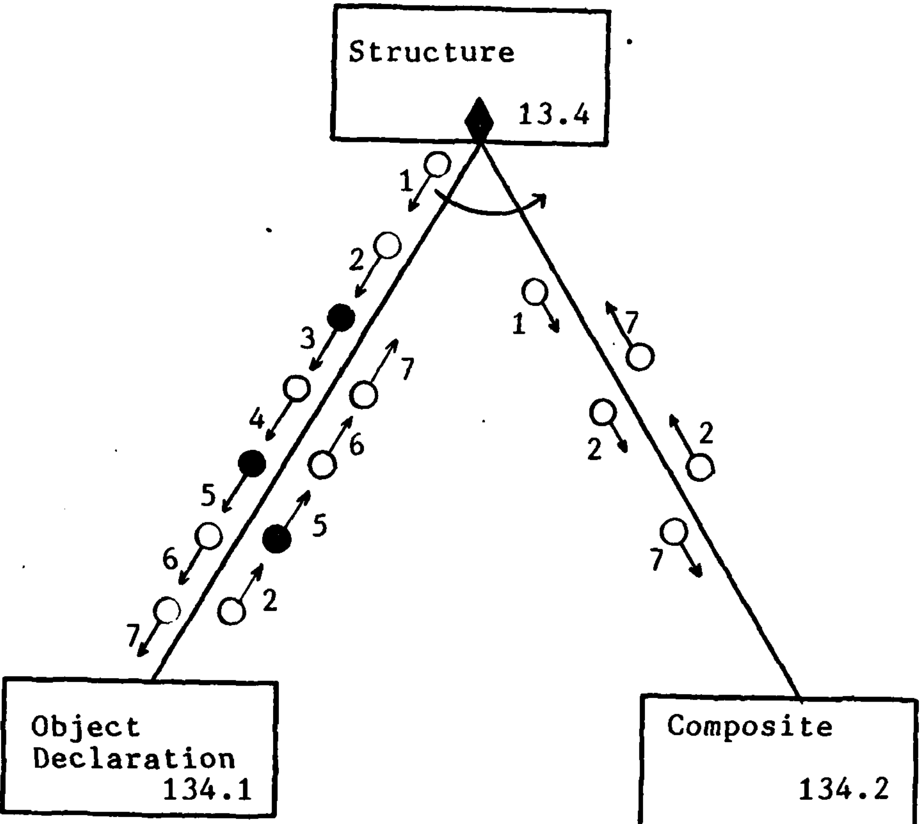
```
A1     1.0 Interpret
A2       1.1 Linkage
           11.1 Share
           11.2 Copy
A3         11.3 Rename
             113.1 Logical Name
             113.2 Determine Class
             113.3 Add Rename
A5       1.3 State
           13.1 Check Rename
           13.2 Check Colon
           13.3 Scalar
A6         13.4 Structure
A7           134.1 Object Declaration
               1341.1 Get Class
A8             1341.2 Declaration
A9               13412.1 Expression
                   134121.1 Double Quotes
                   134121.2 Single Quotes
A10                134121.3 Logical Name
                   134121.4 Parenthesis
                   134121.5 Function or Operator
                   134121.6 Accept
             134.2 Composite
           13.5 Add Class
A11      1.4 Storage
           14.1 Object Declaration
           14.2 Find Member
           14.3 Add Fo
           14.4 Add Class
A12      1.5 Actor
A13        15.1 Function Declaration
A14        15.2 Operator Declaration
A15        15.3 Block
             153.1 Return
             153.2 New
```


Notes:
1: cptr
2: dptr
3: tp
4: file_type
PROJECT:
D TO C INTERPRETER
Interpret
1.0
Linkage
1.1
Bundle
1.2
State
1.3
Storage
1.4
Actor
1.5
AUTHOR:
Lt.Kevin J. Shomper
TITLE:
Interpret
NODE:
A1


Notes:
1: type
2: dptr
3: tp
PROJECT:
D TO C INTERPRETER
Linkage
1.1
Share
11.1
Copy
11.2
Rename
11.3
1
2
3
AUTHOR:
Lt.Kevin J. Shomper
TITLE:
Linkage
NODE:
A2


PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: lname
4: code
5: class
6: err
7: name
8: type
9: defined
Rename
11.3
Logical Name is decomposed
as node number 134121.3.
1
2
3
4
4
3
2
3
6
5
5
3
7
8
9
9
Logical
Name
113.1
Determine
Class
113.2
Add
Rename
113.3
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Rename
NODE:
A3

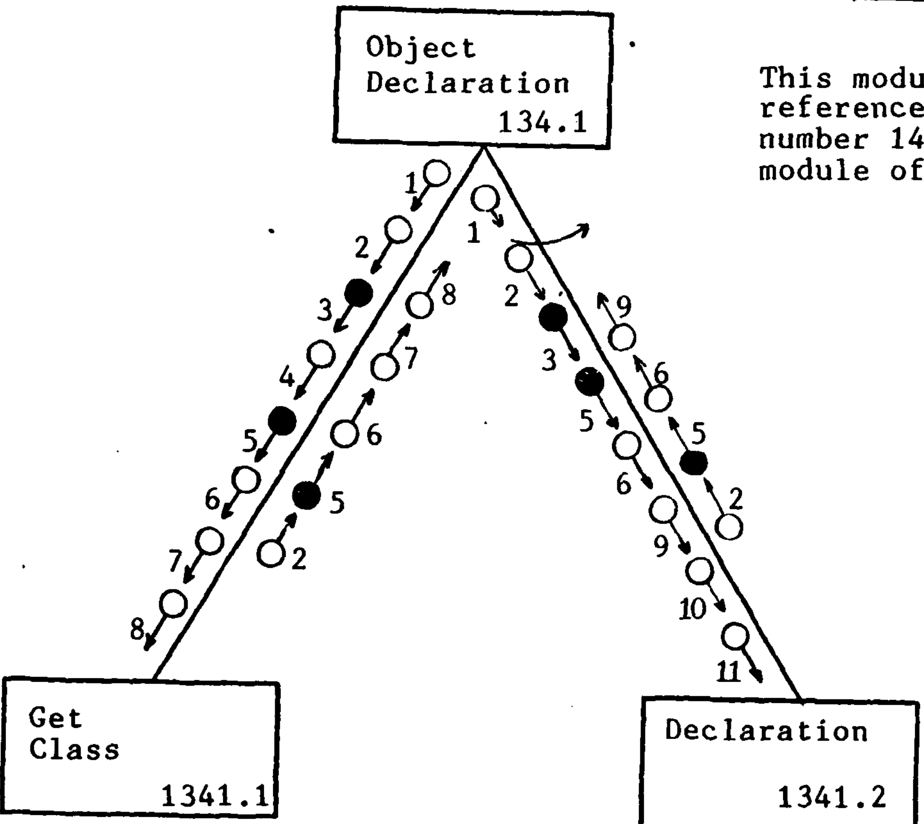

PROJECT:
D TO C INTERPRETER
Notes:
1: name
2: tp
3: dptr
4: s
5: lb
6: ub
7: err
8: pclass
9: code
10: members
11: root
State
1.3
Check
Rename
13.1
Check
Colon
13.2
Scalar
13.3
Structure
13.4
Add
Class
13.5
AUTHOR:
Lt.Kevin J. Shomper
TITLE:
State
NODE:
A5


PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: ft
4: pclass
5: perr
6: code
7: members
Structure
13.4
Object
Declaration
134.1
Composite
134.2
1
2
3
4
5
6
7
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Structure
NODE: A6

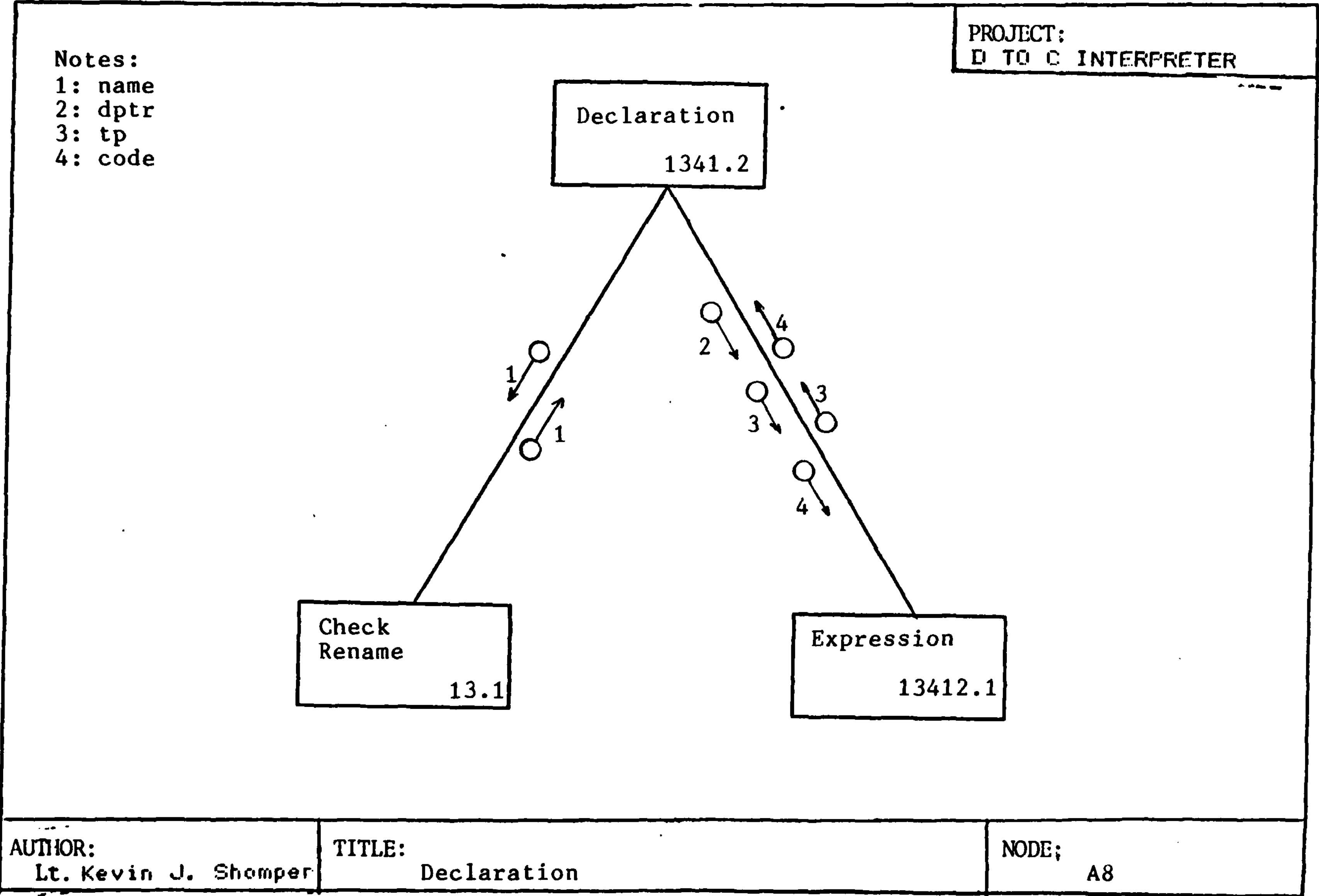
PROJECT:
D TO C INTERPRETER

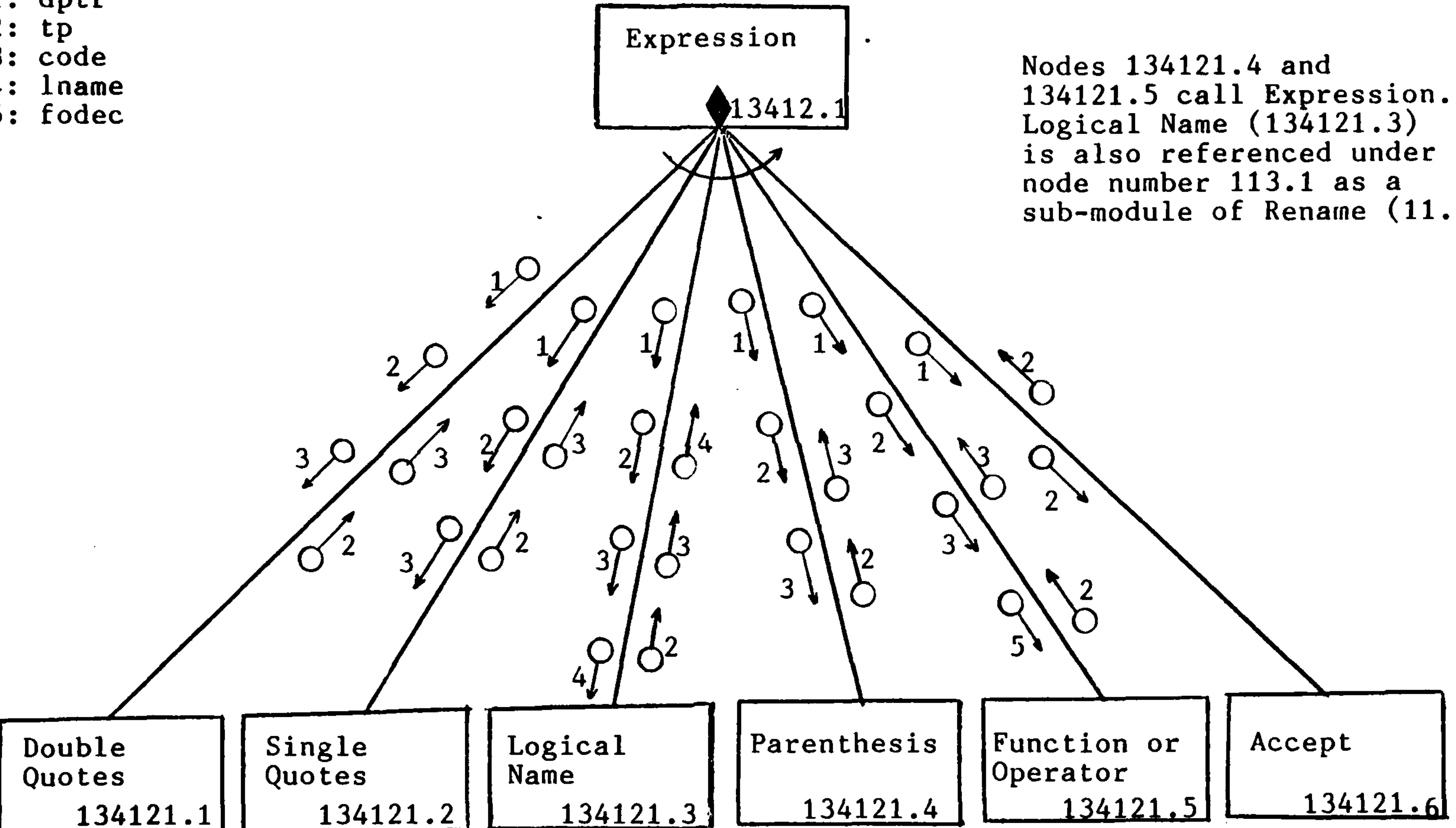
Notes:
1: dptr
2: tp
3: ft
4: sos
5: perr
6: code
7: class
8: vptr
9: members
10: pointer
11: array

This module is also referenced under node number 14.1 as a sub-module of Storage (1.4).



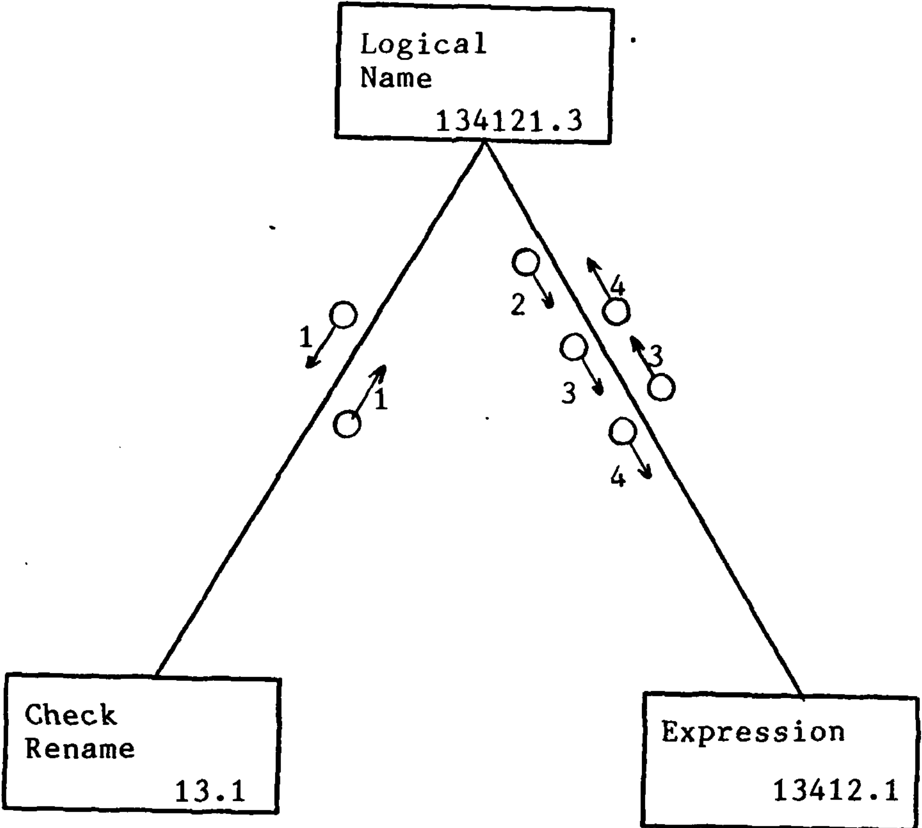
AUTHOR:	TITLE:	NODE:
Lt. Kevin J. Shomper	Object Declaration	A7


Notes:
1: name
2: dptr
3: tp
4: code
PROJECT:
D TO C INTERPRETER
Declaration
1341.2
1
1
2
4
3
3
4
Check
Rename
13.1
Expression
13412.1
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Declaration
NODE;
A8

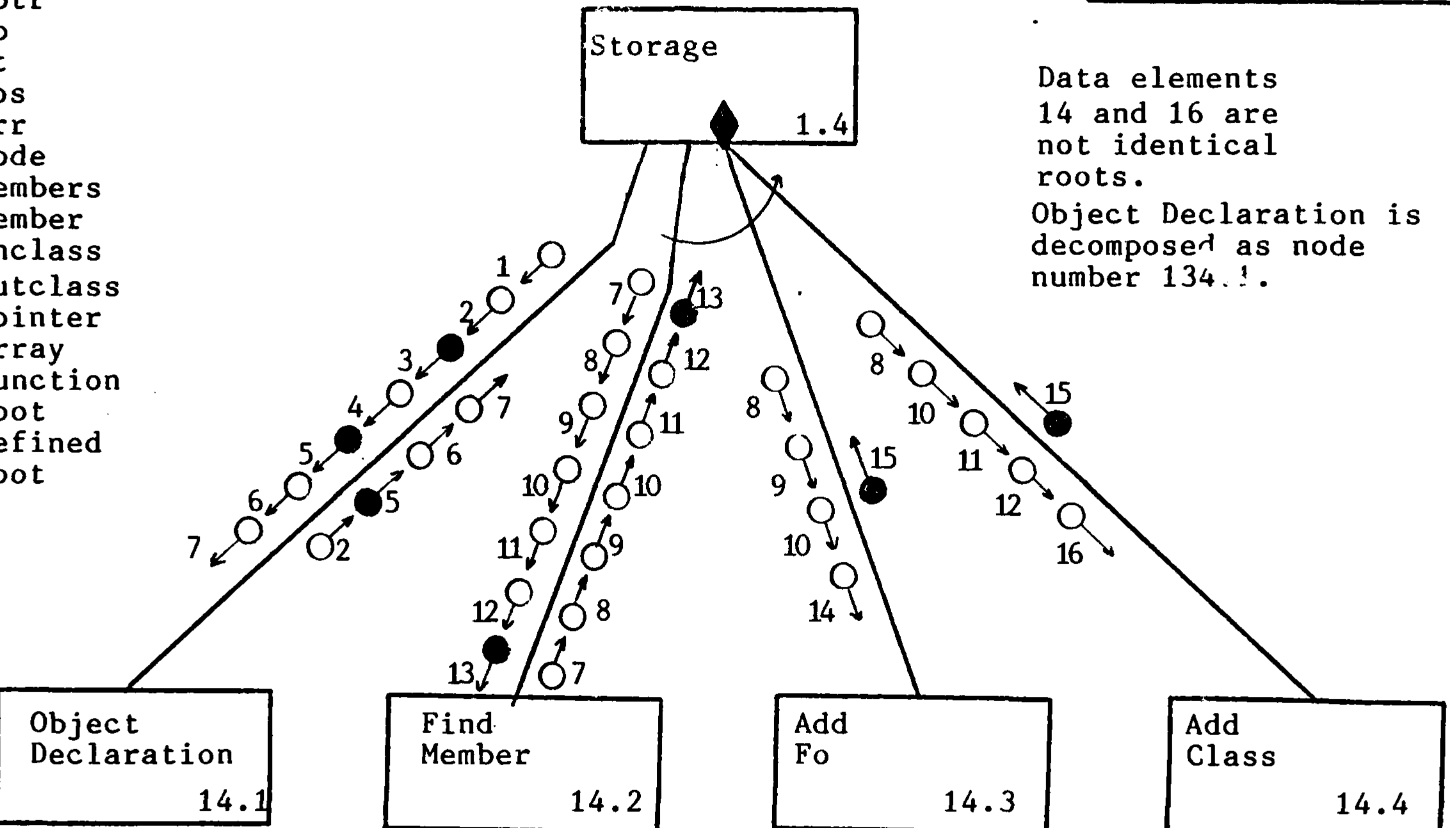

PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: code
4: lname
5: fodec
Expression
13412.1
Nodes 134121.4 and 134121.5 call Expression. Logical Name (134121.3) is also referenced under node number 113.1 as a sub-module of Rename (11.3).
Double Quotes 134121.1
Single Quotes 134121.2
Logical Name 134121.3
Parenthesis 134121.4
Function or Operator 134121.5
Accept 134121.6
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Expression
NODE:
A9

Notes:
1: name
2: dptr
3: tp
4: code

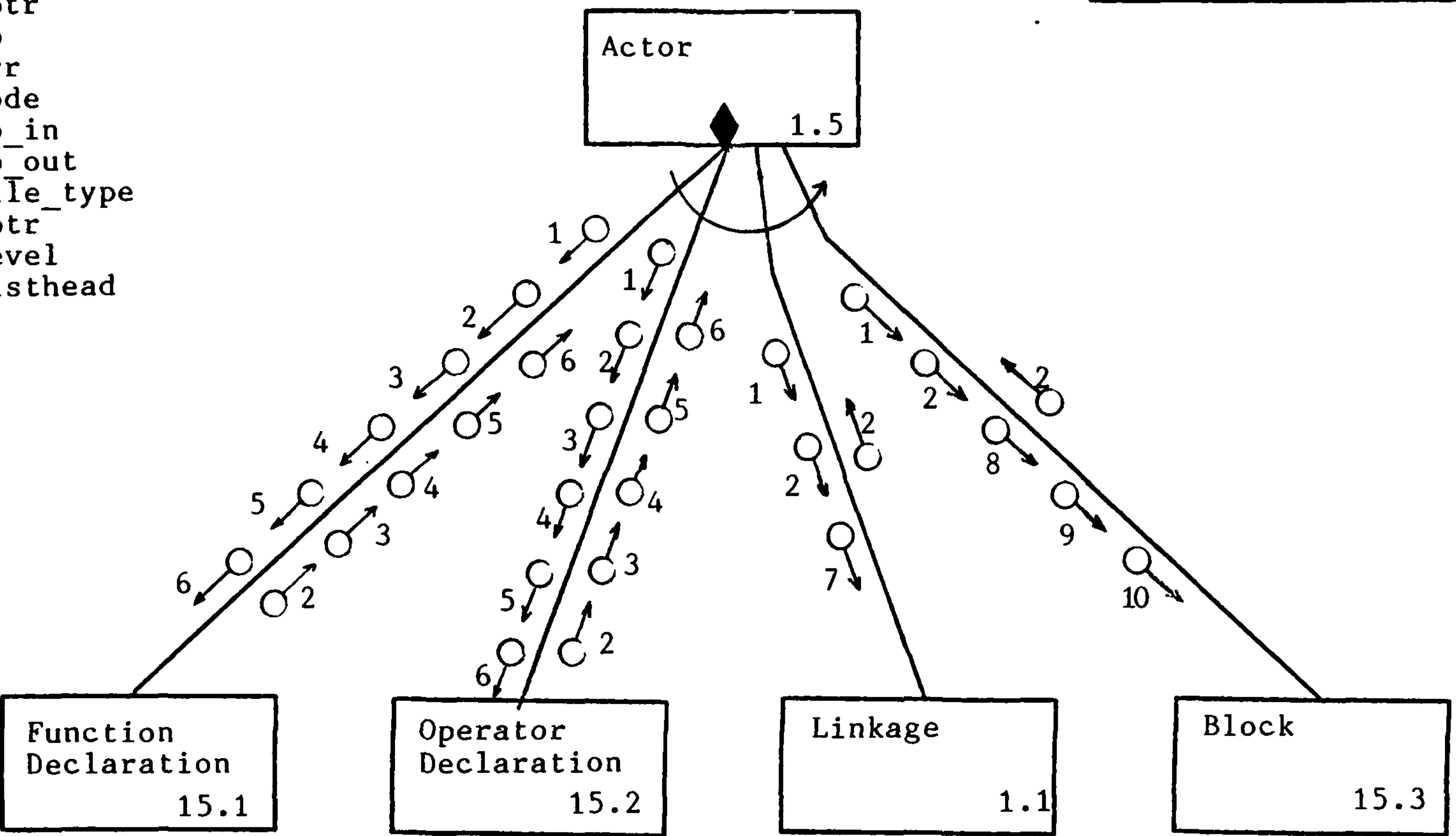
PROJECT:
D TO C INTERPRETER

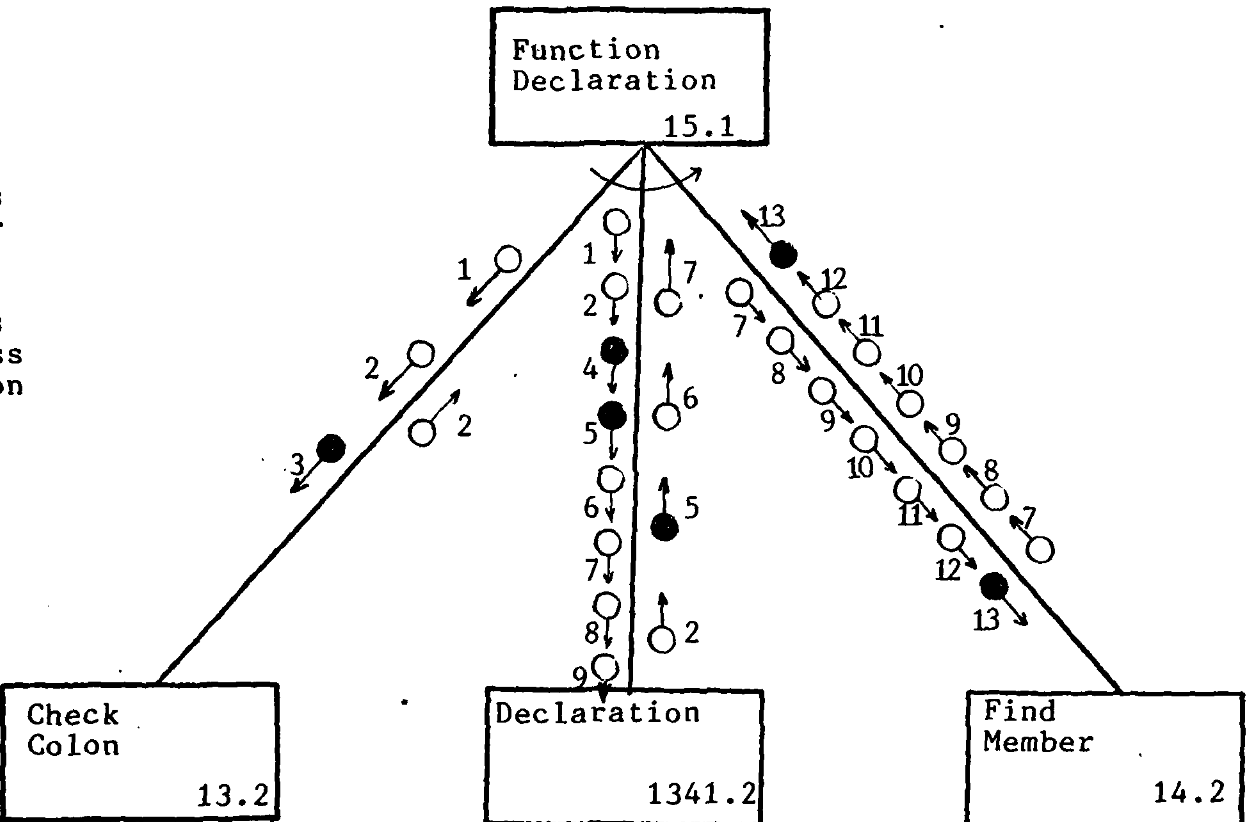


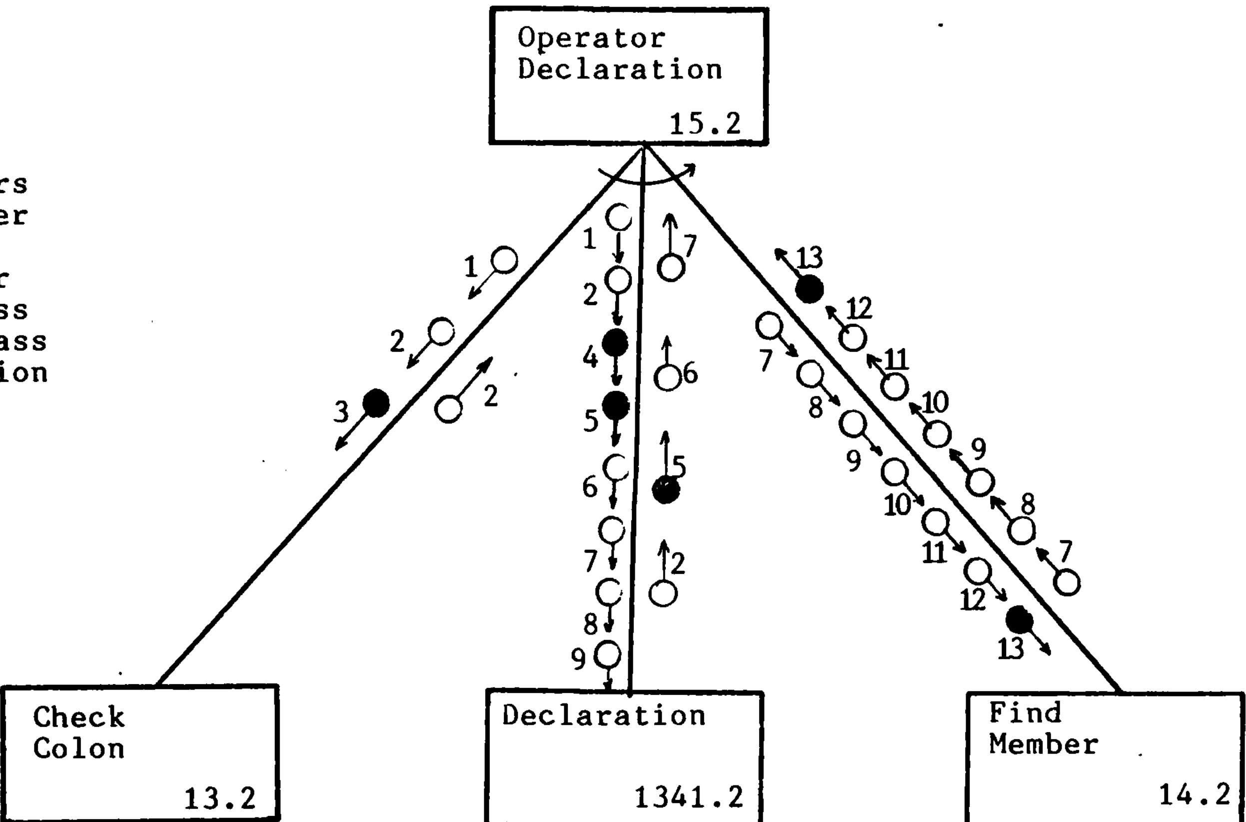
AUTHOR: Lt. Kevin J. Shomper	TITLE: Logical Name	NODE: A10



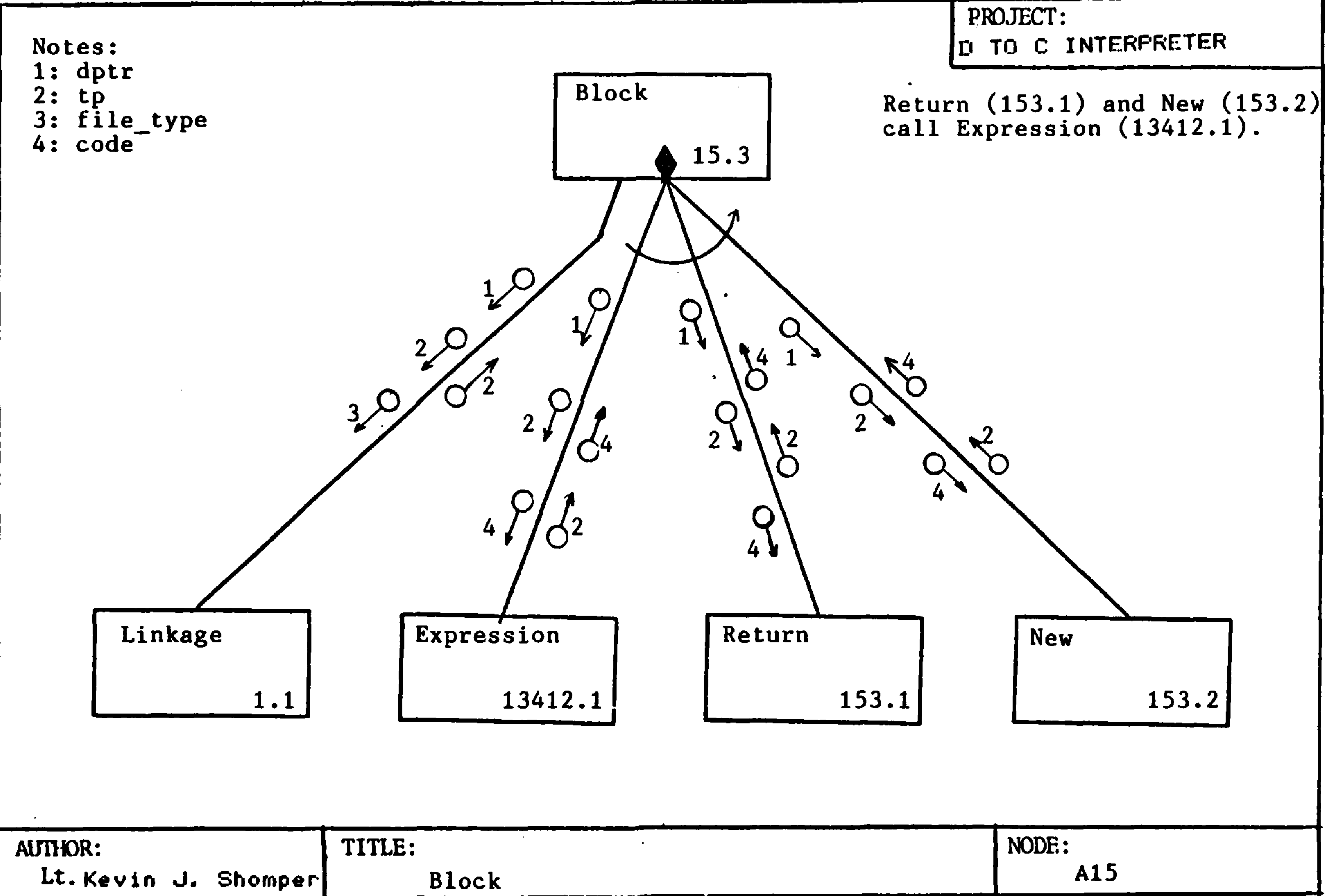
PROJECT:
D TO C INTERPRETER
Notes:
1 : dptr
2 : tp
3 : ft
4 : sos
5 : err
6 : code
7 : members
8 : member
9 : inclass
10: outclass
11: pointer
12: array
13: function
14: root
15: defined
16: root
Storage 1.4
Data elements 14 and 16 are not identical roots.
Object Declaration is decomposed as node number 134.1.
Object Declaration 14.1
Find Member 14.2
Add Fo 14.3
Add Class 14.4
AUTHOR: Lt. Kevin J. Shomper
TITLE: Storage
NODE: A11


PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: err
4: code
5: fo_in
6: fo_out
7: file_type
8: cptr
9: level
10: listhead
Actor
1.5
Function Declaration
15.1
Operator Declaration
15.2
Linkage
1.1
Block
15.3
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Actor
NODE:
A12


PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: s
4: ft
5: perr
6: code
7: members
8: pointer
9: array
10: member
11: inclass
12: outclass
13: function
Function Declaration 15.1
Check Colon 13.2
Declaration 1341.2
Find Member 14.2
AUTHOR:
Lt.Kevin J. Shomper
TITLE:
Function Declaration
NODE:
A13



PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: s
4: ft
5: perr
6: code
7: members
8: pointer
9: array
10: member
11: inclass
12: outclass
13: function
Operator Declaration 15.2
Check Colon 13.2
Declaration 1341.2
Find Member 14.2
AUTHOR:
Lt.Kevin J. Shomper
TITLE:
Operator Declaration
NODE:
A 14


PROJECT:
D TO C INTERPRETER
Notes:
1: dptr
2: tp
3: file_type
4: code
Return (153.1) and New (153.2) call Expression (13412.1).
Block 15.3
Linkage 1.1
Expression 13412.1
Return 153.1
New 153.2
AUTHOR:
Lt. Kevin J. Shomper
TITLE:
Block
NODE:
A15

```c
/***************************************************************************
        Date: 8 Dec 84
        Version: 1
        Title: D to C Interpreter
        Filename: dtoc.c (source) / dc (object)
        Software System:
        Operating System: Unix 4.1 (UCB)
        Language: C
        Use:
        Contents: Too numerous to mention
        Function: To interpret a D program into an equivelent C program
**************************************************************************/

#include <stdio.h>

#define LITERAL 1          /*********************************/
#define INTEGER 2          /*                               */
#define NAME 3             /*                               */
#define KEYWORD 4          /*                               */
#define PUNCTUATION 5      /*          token types          */
#define OPERATOR 6         /*                               */
#define COMMA 7            /*                               */
#define LTERM 8            /*********************************/

#define ILLEGAL_FILE Ø     /*********************************/
#define BUNDLE 1           /*                               */
#define STATE 2            /*          file types           */
#define STORAGE 3          /*                               */
#define ACTOR 4            /*********************************/

#define FALSE Ø            /*********************************/
#define TRUE 1             /*  comma & file ending options  */
#define OPTIONAL 2         /*********************************/

#define NONE Ø             /*********************************/
#define QUIT 1             /*                               */
#define END_LINE 2         /*                               */
#define END_LINK 3         /*        restart options        */
#define END_DEF 3          /*                               */
#define END_BLOCK 4        /*                               */
#define END_FILE 5         /*********************************/

#define NULL Ø
#define PMODE 0644         /*  protection mode  rw-r--r--   */
#define LINESIZE 256
#define MAX_UNDLT_TOKEN 5
#define MAX_OP_NAME_SIZE 3
#define digit_to_ascii(p) p + 48

#define HT   '\011'        /* ASCII horizontal tab   */
#define SP   '\040'        /* ASCII space            */
#define LF   '\012'        /* ASCII line feed        */
#define VT   '\013'        /* ASCII vertical tab     */
#define FF   '\014'        /* ASCII form feed        */

typedef struct linked_list { char array[LINESIZE+1];
                             struct linked_list *next;
                           } LLNODE, *LLPTR;

typedef struct file_list { char name[LINESIZE+1];
                           int type;
                           struct file_list *left, *right;
                         } FLNODE, *FLPTR;
```

```
typedef struct sosnode { char name[LINESIZE+1];
                         LLPTR members;
                         int lb, ub;
                         struct sosnode *left, *right;
                       } SNODE, *SPTR;

typedef struct varnode { char name[LINESIZE+1], type[LINESIZE+1];
                         int pointer, array;
                         struct varnode *left, *right;
                       } VNODE, *VPTR;

typedef struct rename_tree { char lname[LINESIZE+1], name[LINESIZE+1];
                             struct rename_tree *left, *right;
                           } RNODE, *RPTR;

typedef struct func_op_node { char name[LINESIZE+1];
                              char intype[LINESIZE+1], outtype[LINESIZE+1];
                              VPTR arg1, arg2, result;
                              RPTR rtree;
                              struct func_op_node *left, *right;
                            } FONODE, *FOPTR;

typedef struct filenode { char name[LINESIZE+1];
                          int type;
                          LLPTR code;
                          SPTR vdef;
                          VPTR vdec;
                          FOPTR fodef, fodec;
                          RPTR rtree;
                          struct filenode *left, *right;
                        } FNODE, *FPTR;

typedef struct bundle_node { char name[LINESIZE+1];
                             RPTR rtree;
                             FLPTR share, copy;
                             FPTR files;
                             struct bundle_node *left, *right;
                           } BNODE, *BPTR;

typedef struct position { char bundle_name[LINESIZE+1];
                          BPTR bptr, root;
                          char filename[LINESIZE+1];
                          FPTR fptr;
                          char fo_name[LINESIZE+1];
                          FOPTR foptr;
                          FLPTR file_tree;
                        } PNODE;

typedef struct tokennode { char token[LINESIZE+1];
                           int type, line;
                           struct tokennode *prior, *next;
                         } TNODE, *TPTR;

typedef struct listnode {
                          char label[LINESIZE+1];
                          int level;
                          struct listnode *next;
                        } LNODE, *LPTR;

PNODE position;

/***    *********************  ********************************************
        Date: 9 Dec 84
        Version:
```

```
        Name: Main
        Module Number: 0.0
        Function: Receive command line arguments, set up the necessary
                  files, and invoke the interpreter.
        Inputs: argv, argc
        Output : None
        Global Variables Used: None
        Global Variables Changed: None
        Files Read: None
        Files Written: None
        Modules Called: name_change, interpret, creat, close, open
        Calling Modules: None

        Author: 2Lt. Kevin J. Shomper
        History: None
**************************************************************************/

main(argc,argv)
int argc;
char *argv[];
{ char c_filename[LINESIZE+1];
  FILE *d_fileptr, *c_fileptr, *fopen();

  if (argc == 1 || argc >= 3)
    printf("usage:  dtoc d_file\n");
  else if ((d_fileptr = fopen(*++argv,"r")) == NULL)
    printf("dtoc:  can't open %s\n",*argv);
  else
    { name_change(c_filename,*argv);
      creat(c_filename,PMODE);
      close(c_filename);
      c_fileptr = fopen(c_filename,"w");
      interpret(c_fileptr,d_fileptr);
      close(*argv);
      close(c_filename);
    }
}

/************************************************************************
        Name: Name Change
        Function: To add a .c extension onto the source filename.
        Inputs: c, d
        Outputs: c
        Modules Called: None
        Calling Modules: main

        Author: 2Lt. Kevin J. Shomper
        History: None
**************************************************************************/

name_change(c,d)
char *c, *d;
{ while (*c++ = *d++);
  if (*(c-3) == '.' && *(c-2) != 'c')  /* if it ends in .somthing other than .c change the something */
    *(c-2) = 'c';
  else                                 /* otherwise add a .c */
    { *(c-1) = '.';
      c++ = 'c';
      *c = '\0';
    }
}

/*************************************************************************
        Date: 6 Dec 84
```

```
        Version:
        Name: interpret
        Module Number: 1.0
        Function: To control the sequencing of the files, and to initialize
                  variables, and to set up the D driver routine.
        Inputs: cptr, dptr
        Outputs: none
        Global Variables Used: position
        Global Variables Changed: position
        Files Read: Whatever dptr is pointing to (hereafter I'll just say dptr)
        Files Written: Wherever cptr is pointing (hereafter I'll just say cptr)
        Modules Called: init_ring, get_token, is_file, error, in_file_tree,
                        build_file_tree, strcpy, build_bundle_tree, get_bundle
                        get_file, linkage, state, storage, bundle, actor,
                        unget_token, strcmp
        Calling Modules: main

        Author: 2Lt. Kevin J. Shomper
        History: None
*****************************************************************************/

interpret(cptr,dptr)
FILE *cptr, *dptr;
{ int file_type;
  FPTR put_in_file_tree(), get_file();
  FLPTR build_file_tree();
  FLPTR file_tree = NULL;
  BPTR build_bundle_tree(), get_bundle();
  BPTR root = NULL;
  TPTR tp, get_token(), unget_token(), init_ring(), linkage(), error();
  TPTR bundle(), state(), storage(), actor();

  tp = init_ring(MAX_UNGET_TOKEN);     /* sets up the token ring */
  while ((tp = get_token(tp,dptr,TRUE,FALSE))->type != EOF)
    { if (!(file_type = is_file(tp->token)))
        tp = error(5,tp,END_FILE,dptr,"");  /* file type expected */
      else
        if ((tp = get_token(tp,dptr,FALSE,OPTIONAL))->type == NAME)
          { if (in_file_tree(tp->token,file_tree))
              tp = error(6,tp,END_FILE,dptr,""); /* duplicate file name */
            else
              { file_tree = build_file_tree(file_tree,tp->token.file_type);
                position.file_tree = file_tree;    /****************************/
                position.filename[0] = '\0';       /*                          */
                position.fptr = NULL;              /*       initialize         */
                position.fo_name[0] = '\0';        /*        position          */
                position.foptr = NULL;             /*                          */
                if (file_type == BUNDLE)           /****************************/
                  { strcpy(position.bundle_name,tp->token);
                    root = build_bundle_tree(root,tp->token);
                    position.bptr = get_bundle(tp->token,root);
                  }
                else
                  { strcpy(position.filename,tp->token);
                    if ((position.fptr = get_file(tp->token,position.bptr->files)) != NULL)
                      position.fptr->type = file_type;
                  }
                if (position.fptr != NULL || file_type == BUNDLE)
                  { if (position.bptr->share != NULL && file_type != BUNDLE)
                      link_globals(position.bptr->share,tp);                 /* the semi-global sharing */
                    if (position.bptr->copy != NULL && file_type != BUNDLE)
                      link_globals(position.bptr->copy,tp);                  /* the semi-global copying */
                    tp = linkage(dptr,tp,file_type);
                    switch(file_type) {
```

```
                case BUNDLE : tp = bundle(dptr,tp);
                              break;
                case STATE  : tp = state(cptr,dptr,tp);
                              break;
                case STORAGE: tp = storage(cptr,dptr,tp);
                              break;
                case ACTOR  : tp = actor(cptr,dptr,tp);
                              break;
                }
              if (strcmp((tp=get_token(tp,dptr,FALSE,FALSE))->token,"."))
                tp = error(12,tp,END_FILE,dptr,position.filename);  /* missing terminating . */
            }
          else
            tp = error(46,tp,END_FILE,dptr,position.filename);  /* the file does not appear in the bundle's body */
        }
      }
    else
      tp = error(7,tp,END_FILE,dptr,"");  /* illegal name */
    }
  generate_main(cptr);
}

/*****************************************************************************
        Name: generate_main
        Function: To produce the main program which will drive the D en-
                  vironment.  This will be written to cptr.  It's a stub
                  presently.
        Inputs: cptr
        Outputs: None
        Modules Called: None
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
*****************************************************************************/

generate_main(cptr)
FILE *cptr;
{ fprintf(cptr,"\n\n");
  fprintf(cptr,"main()\n");
  fprintf(cptr,"{\n");
  fprintf(cptr,"  /**************************************************/\n");
  fprintf(cptr,"  /*                                                */\n");
  fprintf(cptr,"  /*              D program driver code             */\n");
  fprintf(cptr,"  /*                                                */\n");
  fprintf(cptr,"  /**************************************************/\n");
  fprintf(cptr,"}\n");
}

/*****************************************************************************
        Name: init_ring
        Function: To build the token ring.
        Inputs: size
        Output : a pointer to one node of the ring
        Modules Called: None
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
*****************************************************************************/

TPTR init_ring(size)
int size;
{
```

```
  TPTR start, temp;
  char *calloc();
  int i;

  start = temp = (TPTR)calloc(1,sizeof(TNODE));
  start->token[0] = '\Ø';
  start->type = start->line = Ø;
  start->next = start;
  for (i = 1; i <= size; i++)
    { temp->next = (TPTR)calloc(1,sizeof(TNODE));
      temp->next->prior = temp;
      temp = temp->next;
      temp->token[Ø] = '\Ø';
      temp->type = temp->line = Ø;
    }
  temp->next = start;
  start->prior = temp;

  return(start);
}

/*************************************************************************
        Name: is_file
        Function: To determine if the given character string is one of D's
                  four file types.
        Inputs: t
        Output : 1 = TRUE, Ø = FALSE
        Modules Called: None
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
*************************************************************************/

is_file(t)
char *t;
{
  int ft = ILLEGAL_FILE;

  if (!strcmp(t,"bundle"))
    ft = BUNDLE;
  else if (!strcmp(t,"state"))
    ft = STATE;
  else if (!strcmp(t,"storage"))
    ft = STORAGE;
  else if (!strcmp(t,"actor"))
    ft = ACTOR;

  return(ft);
}

/*************************************************************************
        Name: in_file_tree
        Function: To determine if the given file is in the a file tree.
        Inputs: name, root (of type FLPTR)
        Output : 1 = TRUE, Ø = FALSE
        Modules Called: None
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
*************************************************************************/

in_file_tree(name,root)
```

```
char *name;
FPTR root;
{ int x;

  if (root == NULL)
    x = 0;
  else if (strcmp(root->name,name) > 0)
    x = in_file_tree(name,root->right);
  else if (strcmp(root->name,name) < 0)
    x = in_file_tree(name,root->left);
  else
    x = 1;

  return(x);
}

/**************************************************************************
        Name: put_in_file_tree
        Function: To create a new location in a file tree (type FPTR) for
                  the given name of a file.
        Inputs: name, root
        Output : root
        Module  Called: calloc
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
**************************************************************************/

FPTR put_in_file_tree(root,name)
FPTR root;
char *name;
{ char *calloc();

  if (root == NULL)
    { root = (FPTR)calloc(1,sizeof(FNODE));
      strcpy(root->name,name);
      root->type = NULL;
      root->[illegible] = NULL;
      root->vd[illegible]f = NULL;
      root->vd[illegible] = NULL;
      root->fo[illegible]f = root->fodec = NULL;
      root->[illegible]tree = NULL;
      root->left = root->right = NULL;
    }
  else if (strcmp(name,root->name) > 0)
    root->right = put_in_file_tree(root->right,name);
  else if (strcmp(name,root->name) < 0)
    root->left = put_in_file_tree(root->left,name);

  return(root);
}

/**************************************************************************
        Name: build_bundle_tree
        Function: To create a new location in a file tree (type BPTR) for
                  the given name of a file.
        Inputs: name, root
        Output : root
        Module  Called: calloc
        Calling Modules: interpret

        Author: 2Lt. Kevin J. Shomper
        History: None
```

```
*************************************************************************/

BPTR build_bundle_tree(root,name)
BPTR root;
char *name;
{ char *calloc();

  if (root == NULL)
    { root = (BPTR)calloc(1,sizeof(BNODE));
      strcpy(root->name,name);
      root->title = NULL;
      root->share = root->copy = NULL;
      root->files = NULL;
      root->left = root->right = NULL;
    }
  else if (strcmp(name,root->name) > Ø)
    root->right = build_bundle_tree(root->right,name);
  else if (strcmp(name,root->name) < Ø)
    root->left = build_bundle_tree(root->left,name);

  return(root);
}

/************************************************************************
        Name: build_file_tree
        Function: To create a new location in a file tree (type FLPTR) for
                  the given name of a file.
        Inputs: name, root, type
        Output : root
        Modules Called: calloc
        Calling Modules: interpret

        Author: 1Lt. Kevin J. Shomper
        History: None
*************************************************************************/

FLPTR build_file_tree(root,name,type)
FLPTR root;
char *name;
int type;
{ char *calloc();

  if (root == NULL)
    { root = (FLPTR)calloc(1,sizeof(FLNODE));
      strcpy(root->name,name);
      root->type = type;
      root->left = root->right = NULL;
    }
  else if (strcmp(name,root->name) > Ø)
    root->right = build_file_tree(root->right,name,type);
  else if (strcmp(name,root->name) < Ø)
    root->left = build_file_tree(root->left,name,type);

  return(root);
}

/************************************************************************
        Name: link_globals
        Function: To perform the global sharing and copying functions.
        Inputs: global_root, tp
        Outputs: None
        Modules Called: common, error
        Calling Modules: interpret
```

```
        Author: 2Lt. Kevin J. Shomper
        History: None
***************************************************************************/

find_globals(global_root,tp)
ELPTR global_root;
TPTR tp;
{ int err;

  if (global_root->left != NULL)
    find_globals(global_root->left);
  common(global_root->name,&err);
  if (err == 1 || err == 3)
    error(30,err,tp,NONE,NULL,position.bundle_name);
  if (global_root->right != NULL)
    find_globals(global_root->right);
}

/**************************************************************************
        Date: 8 Dec 83
        Version:
        Name: linkage
        Module Number: 1.1
        Function: To provide parsing and control of the D linkage facility.
        Inputs: dptr, tp, ft
        Outputs: tp
        Global Variables Used: position
        Global Variables Changed: None
        Files Read: dptr
        Files Written: None
        Modules Called: get_token, unget_token, error, share, copy, rename,
                        strcmp, strcpy, prior, next
        Calling Modules: interpret, actor, block

        Author: 2Lt. Kevin J. Shomper
        History: None
***************************************************************************/

TPTR linkage(dptr,tp,ft)
FILE *dptr;
TPTR tp;
int ft;
{
  TPTR get_token(), unget_token(), share(), copy(), rename();
  TPTR error();
  int line, temp;
  char type[2], str1[LINESIZE+1], str2[LINESIZE+1], temptoken[LINESIZE+1];
  char *cp, *prior(), *next();

  type[0] = digit_to_ascii(ft);
  type[1] = '\0';
  while ((tp = get_token(tp,dptr,FALSE,FALSE))->type == KEYWORD)
      if (!strcmp(tp->token,"share") && (ft != 5))  /* sharing is not permitted immediatly following function or operator sp
        tp = share(dptr,tp,type);
      else if (!strcmp(tp->token,"copy") && (ft != 5)) /* copying is not permitted as above */
        tp = copy(dptr,tp,type);
      else if (!strcmp(tp->token,"rename") && (ft != 6)) /* renaming is not permitted inside blocks */
        tp = rename(dptr,tp,type);
      else if (!strcmp(tp->token,"share") || !strcmp(tp->token,"copy") || !strcmp(tp->token,"rename"))
        tp = error(32,tp,END_LINK,dptr,type);   /* indicates the above comments to the user */
      else
        break;
  cp = prior(tp);              /*****************************************/
  strcpy(str1,cp);             /* saving current location for comparison  /
```

```
strcpy(str2,tp->token);     /**************************************/
[illegible]                 /* saves the line number in the event an error occurs */
[illegible]= error(0,tp.END_LINK,dptr,type);  /* printing no error, but trying to restart on a valid pattern follow linkage */
[illegible]= next(tp);
[illegible] (strcmp(str2,tp)) /* if at same location linkage section is ended */
  { temp = tp->line;                       /* otherwise an error in linkage occured */
    [illegible]
    [illegible]
    strcpy(tp->token,str2);
    [illegible]                            /* indicates to the user a linkage error occured */
    [illegible]
    [illegible]                            continue the linkage section of D code */

[illegible]

/*****************************************************************************
    [illegible]
    Section Number: 11.1
    Function: to [illegible] the [illegible]ring function
    [illegible]
    [illegible]
    [illegible] position
    [illegible] position
    [illegible]: dptr
    Files Written: None
    Modules Called: [illegible]_token, unget_token, error, build_file_tree, strcmp,
                    [illegible]
    Calling Modules: linkage

    Author: 2Lt. [illegible] J. Shomper
    History: None
*****************************************************************************/

[illegible]

      [illegible]_token(), error();

      [illegible](tp,dptr,FALSE,OPTIONAL))->type != NAME)
         [illegible](0,[illegible]_LINK,dptr,type); /* wants a file name */

      [illegible] /* if the type is BUNDLE the share is semi-global so it goes into the bundles share list */
         [illegible]share = build_file_tree(position.bptr->share,tp->token,NULL);

      [illegible]
         [illegible](tp.NONE,dptr,"copy"); /* either the file is already defined, the file is not defined, */
                                           /* or the file is not in the same bundle */
      [illegible] FALSE;

   while [illegible](tp,dptr,FALSE,TRUE)) && (!strcmp(tp->token,",")));
 if (! [illegible])
   tp [illegible]
```

```
    return(tp);

/* [illegible] *************************************************************
        [illegible]
        [illegible]
        [illegible]
        [illegible]: 11.[illegible]
        [illegible] to implement the copyng function
        Input: dptr, [illegible], type
        Output: tp
        Global Variable Used: position
        Global Variable Changed: position
        [illegible]
        [illegible]
        [illegible]   get_token, unget_token, error, build_file_tree, strcmp,
                      [illegible]
        [illegible]   [illegible]

        [illegible] Kevin J. Thompson
        [illegible]
******** [illegible] ******************************************************/

TPTR [illegible]
FILE [illegible];
[illegible];
[illegible];
{
  TPTR get_token(), unget_token(), error();
  FLPTR build_file_tree();
  int [illegible] = FALSE;

  do    if ((tp = get_token(tp,dptr,FALSE,OPTIONAL))->type != NAME)
          { tp = error(3,tp,END_LINE,dptr,type);  /* wants a file name */
            [illegible] = TRUE;
            [illegible];
          }
        if ([illegible]("1"))  /* if the type is BUNDLE the named file is semi-global: hence, it is placed in the bundle's 1
          position->dptr->copy = build_file_tree(position,bptr->copy,tp->token,NULL);
        else
          { [illegible](tp->token,&err);
            if ([illegible])
              { [illegible] 40(err,tp,NONE,dptr,"copy"); /* either the file is already defined, the file is not defined, */
                                                         /* or the file does not belong to the same bundle */
                [illegible] = FALSE;
              }
          }
  while ([illegible]get_token(tp,dptr,FALSE,TRUE)) && (! [illegible](tp->token,",")));
  if ([illegible])
    tp = unget_token(tp);

  return(tp);
}

/*** [illegible] ***********************************************************
        [illegible]
        [illegible]
        [illegible]
        [illegible]: 11.3
        [illegible]: to implement the D renaming function
        Input: dptr, tp, type
        Output: tp
        Global Variable Used: None
        Global Variable Changed: None
        [illegible]: dptr
```

```
    Files Written: None
    ... get_token, unget_token, error, logical_name, calloc,
              ... determine_class, add_rename ...
    Calls: ...

    Author: ... in J. Thom...
    History: ...
********************************************************************/

TPTR ... (dptr,tp,...)
FILE *...;
...
char ...;
{
  TPTR get_token(), unget_token(), error(), logical_name();
  char ...[], calloc();
  int ...;
  int ...;
  int ...;
  LLPT ...;
  VNODE ...;

  ...[0] = ...;
  ... calloc(1,sizeof(VNODE));
  if ((tp = get_token(...,dptr,FALSE,OPTIONAL))->type != OPERATOR)
    { if ( ...(tp->token,"%"))
        tp = unget_token(tp);
      else
        ...;
      tp = logical_name(dptr,tp,lname,code);
    }
  else
    strcpy(lname,tp->token);
  if ((tp = get_token(...,dptr,FALSE,OPTIONAL))->type != NAME)
    tp = error(...,tp,...,dptr,type);
  else
    { if (...)
        { ... = determine_class(class,lname);
          ... (class->type);
        }
      if ( ...)
        tp = error(...,tp,NAME,dptr,"");  /* either the logical name has not been declared, too many pointer */
                                          /* references, members not permitted, too many array references,   */
                                          /* members definition not available, illegal members               */
      else
        { add_rename(..., tp->token,type,&defined);
          if (...)
            ... dptr,"");  /* the name is already used for renaming */
        }
    }
  return(tp);
}

/***************************************************************************
    ...
    ... the source of the shared or copied file.
    ... file, ...
    ... None
    ... find_file, get_bundle, common_list, copy_a_list
    ... copy

    Author: ... in J. Thompson
    History: None
***************************************************************************/
```

```
common(sending_file,perr)
[illegible] sending_file;
int *perr;
{ FPTR filptr, find_file();
  BPTR bptr, [illegible]_bundle();
  FPTR [illegible] ();

  if ((filptr = find_file(sending_file.position.file_tree)) == NULL)
     [illegible]
  else if (filptr->type == BUNDLE)
    { bptr = find_bundle(sending_file.position.root);
      if ([illegible] != NULL)
         [illegible] ,perr);  /* want to share or copy all files within the bundle */
    }
  else
    if ((filptr = [illegible]_file(sending_file.position.bptr->files)) == NULL)
      [illegible]
    else
      copy_a_list(filptr,perr);  /* if we got to here we have a file to copy or share */
}

/*****************************************************************************
        Name: common_list
        Function: copies or shares the files in a bundle's file tree
        Input: root (type FPTR), perr
        Output: None
        Modules Called: copy_a_list
        Calling Modules: common

        Author: 1Lt. Kevin J. Thomper
        History: None
*****************************************************************************/

common_list(root,perr)
FPTR root;
int *perr;
{ if (root->left != NULL)  /* walk the file tree copying each file as you go */
     common_list(root->left,perr);
  copy_a_list(root,perr);
  if (root->right != NULL)
     common_list(root->right,perr);
}

/*****************************************************************************
        Name: copy_a_list
        Function: To copy the [illegible] actors, or
                  [illegible] (only one).
        Input: sndfileptr, perr
        Output: None
        Modules Called: copy_[illegible]_list, copy_[illegible]
        Calling Modules: common, common_list

        Author: 1Lt. Kevin J. Thomper
        History: None
*****************************************************************************/

copy_a_list(sndfileptr,perr)
FPTR sndfileptr;
int *perr;
{ if (sndfileptr->vdef != NULL &&  sndfileptr->type == STATE)
     copy_[illegible]_list(sndfileptr->vdef,perr);
  if (sndfileptr->vdec != NULL && sndfileptr->type == STORAGE)
     copy_var_list(sndfileptr->vdec,perr);
  if (sndfileptr->fodef != NULL && sndfileptr->type == ACTOR)
```

```
    copy_fo_list(sndfileptr->fodef,sndfileptr->type,perr);
  if (sndfileptr->fodec != NULL && sndfileptr->type == STORAGE)
    copy_fo_list(sndfileptr->fodec,sndfileptr->type,perr);
}

/**************************************************************************
        Name: copy_class_list
        Function: To copy all the defined classes from the sending file to
                  the receiving file.
        Inputs: sndroot, perr
        Output : None
        Modules Called: add_class
        Calling Modules: copy_a_list

        Author: 2Lt. Kevin J. Shomper
        History: None
**************************************************************************/

copy_class_list(sndroot,perr)
SPTR sndroot;
int *perr;
{ SPTR add_class();

  if (sndroot->left != NULL)
    copy_class_list(sndroot->left,perr);
  position.fptr->vdef = add_class(position.fptr->vdef,sndroot->name,
                        sndroot->members,sndroot->lb,sndroot->ub,perr);
  if (sndroot->right != NULL)
    copy_class_list(sndroot->right,perr);
}

/**************************************************************************
        Name: copy_var_list
        Function: To copy all the declared variables from the sending file to
                  the receiving file.
        Inputs: sndroot, perr
        Output : None
        Modules Called: add_var
        Calling Modules: copy_a_list

        Author: 2Lt. Kevin J. Shomper
        History: None
**************************************************************************/

copy_var_list(sndroot,perr)
VPTR sndroot;
int *perr;
{ VPTR add_var();

  if (sndroot->left != NULL)
    copy_var_list(sndroot->left,perr);
  position.fptr->vdec = add_var(position.fptr->vdec,sndroot->name,sndroot->type,
                        sndroot->pointer,sndroot->array,perr);
  if (sndroot->right != NULL)
    copy_var_list(sndroot->right,perr);
}

/**************************************************************************
        Name: copy_fo_list
        Function: To copy all the  declared or defined actors from the
                  sending file to the receiving file.
        Inputs: sndroot, perr
        Output : None
        Modules Called: add_fo
```

```
        Calling Modules: copy_a_list

        Author: 2Lt. Kevin J. Shomper
        History: None
***********************************************************************/

copy_fo_list(sndroot,sndfiletype,perr)
FOPTR sndroot;
int sndfiletype, *perr;
{ FOPTR add_fo();

  if (sndroot->left != NULL)
    copy_fo_list(sndroot->left,sndfiletype,perr);
  if (sndfiletype == ACTOR)
    position.fptr->fodef = add_fo(position.fptr->fodef,sndroot->name,
                                  sndroot->intype,sndroot->outtype,perr);
  else
    position.fptr->fodec = add_fo(position.fptr->fodec,sndroot->name,
                                  sndroot->intype,sndroot->outtype,perr);
  if (sndroot->right != NULL)
    copy_fo_list(sndroot->right,sndfiletype,perr);
}

/**********************************************************************
        Name: add_rename
        Function: To add a renaming pair the someone's rename tree.
        Inputs: lname, name, type, pi
        Outputs: pi
        Modules Called: add_r, strcmp
        Calling Modules: rename

        Author: 2Lt. Kevin J. Shomper
        History: None
***********************************************************************/

add_rename(lname,name,type,pi)
char *lname, *name, *type;
int *pi;
{ RPTR add_r();

  if (!strcmp(type,"5")) /* if it's local to an actor put it on the actor's renaming tree */
    position.foptr->rtree = add_r(position.foptr->rtree,lname,name,pi);
  else if (!strcmp(type,"1")) /* if is semi-global put it on the bundle's renaming tree */
    position.bptr->rtree = add_r(position.bptr->rtree,lname,name,pi);
  else /* otherwise put it on the file's renaming tree */
    position.fptr->rtree = add_r(position.fptr->rtree,lname,name,pi);
}

/**********************************************************************
        Name: add_r
        Function: the recursive part of add_rename
        Inputs: root (type RPTR), lname, name, pi
        Outputs: pi
        Modules Called: strcpy, calloc
        Calling Modules: add_rename

        Author: 2Lt. Kevin J. Shomper
        History: None
***********************************************************************/

RPTR add_r(root,lname,name,pi)
RPTR root;
char *lname, *name;
int *pi;
```

```
{ char *calloc();

  [illegible] = TRUE;
  if (root == NULL)
    { root = (RPTR)calloc(1,sizeof(RNODE));
      strcpy(root->name,name);
      strcpy(root->lname,lname);
      root->left = root->right = NULL;
      [illegible]
    }
  else if (strcmp(root->name,name) > 0)
    root->right = add_[illegible](root->right,lname,name,pi);
  else if ([illegible](root->name,name) < 0)
    [illegible] add_[illegible](root->left,lname,name,pi);

  return(r[illegible]);
}


TPTR [illegible](dptr,tp)
FILE [illegible];
TPTR [illegible];
{
  TPTR [illegible](), [illegible]t_token(), object_dec(), error();
  int [illegible], function, defined;
  char [illegible][LINESIZE+1], inclass[LINESIZE+1], outclass[LINESIZE+1];
  [illegible] calloc(), find_member();
  [illegible];
  [illegible];

  [illegible]);
  while ([illegible]((tp = get_token(tp,dptr,FALSE,FALSE))->token,".") &&
         ![illegible](position))
    { [illegible];
        [illegible]_token(tp);
      [illegible]calloc([illegible]);
      [illegible]_dec(dptr,tp,STORAGE,"",&err,code,members);
      [illegible]array[0]) move(code,";",LF);
        (!err)
      { while ((members = find_member(member,inclass,outclass,&pointer,&array,
                                      &function,members))
          if (function)
            position.fptr->fodec = add_fo(position.fptr->fodec,member,
                                          inclass,outclass,&defined);
          else
            position.fptr->vdec = add_var(position.fptr->vdec,member,
                                          outclass,pointer,array,&defined);
        if (!defined)
          tp = error(1,tp,NONE,dptr,"");
      }
    }
  tp = unget_token(tp);

rite(code,cptr);
fprintf(cptr,"\n");
  return(tp);
}

LLPTR find_member(member,inclass,outclass,pointer,array,function,members)
char *member, *inclass, *outclass;
int *pointer, *array, *function;
LLPTR members;
{ LLPTR move_string();
```

```
    *inclass = '\0';
     outclass = '\0';
     [illegible] = *[illegible] = *function = 0;
     [illegible] string(member,members);
     if ([illegible] != NULL)
       [illegible] "*"))
          [illegible] move_string(member,members);
          [illegible]
          [illegible] move_string(outclass,members);
          while [illegible] !strcmp(outclass,"[") ||
                 !strcmp(outclass,"("))
             [illegible] if [illegible]
                [illegible]pointer)++;
             [illegible] if (!strcmp(outclass,"["))
                [illegible]
             [illegible]
                 (*function)++;
                 [illegible] move_string(inclass,members);

             [illegible] move_string(outclass,members);
           }
       }

   return(members);
  }

  LLPTR move_string(string,linked_list)
  char *string;
  LLPTR linked_list;
  { int i = -1;

    if (linked_list != NULL)
      { while (linked_list->array[++i] == SP && i < LINESIZE);
        if (linked_list->array[i] == SP)
          linked_list = move_string(string,linked_list->next);
        else
          { while ((*string++ = linked_list->array[i++]) != SP && *(string-1) &&
                    i <= LINESIZE)
              linked_list->array[i-1] = SP;
            if (*(string-1) == SP)
              *(string-1) = '\0';
            else if (!(*(string-1)))
              linked_list = NULL;
            else
              linked_list = move_string(string,linked_list->next);
          }
      }

    return(linked_list);
  }

  TPTR [illegible]
  [illegible]
  [illegible]
  {
    TPTR [illegible](), unget_token(), error();
    FPTR put_in_file_tree();

    while [illegible]((tp = get_token(tp,dptr,FALSE,FALSE))->token)
           [illegible](tp->token,"."))
      if (tp->[illegible] NONE)
        [illegible] = put_in_file_tree(position.bptr->files,tp->token);
      else
        [illegible](tp,NONE,dptr,"");
```

```
  tp = unget_token(tp);
  [illegible]

[illegible]

  do [illegible]
       [illegible]
       [illegible] == COMMA || tp->type == LTERM);

  return(tp);
}

char [illegible]
TPTR [illegible]
{ char [illegible]

  do [illegible]
       [illegible]
     } while [illegible] == COMMA || tp->type == LTERM);

  return(tp);
}


TPTR [illegible](dptr,tp)
FILE [illegible]
TPTR [illegible]
{ [illegible] unget_token(), structure(), check_colon(), error();
  [illegible]
  [illegible]
  [illegible]
  [illegible]
  [illegible]
  [illegible]

  [illegible]
  [illegible] get_token(tp,dptr,FALSE,FALSE))->token) &&
       [illegible] ";"))
    [illegible]
      [illegible]
      [illegible]
        [illegible] != NAME)
        { [illegible] (7,tp,END_DEF,dptr,"");
          [illegible]
        }
      [illegible]
      { [illegible](tp->token);
          [illegible](tp->token);
        line = tp->line;
        tp = check_colon(tp,dptr,":");
        if ((tp = get_token(tp,dptr,FALSE,OPTIONAL))->type==INTEGER)
          tp = [illegible](dptr,tp,&lb,&ub,&err,pclass,code);
        else if (!strcmp(tp->token,"("))
          tp = structure(dptr,tp,pclass,&err,code,members);
        else
        { [illegible] = error(25,tp,END_DEF,dptr,"");
            err = TRUE;
        }
      }
    if (!err)
```

```
      { position.fptr->vdef = add_class(position.fptr->vdef,pclass,
                                         members,lb,ub,&err);
        if (err)
          tp = error(75,tp,NONE,dptr,pclass);
      }
    }
  tp = unget_token(tp);
  [illegible]
  [illegible] = code;

  return(tp);
}

TPTR [illegible](dptr,tp,plb,pub,perr,pclass,code)
FILE *dptr;
int *p[illegible], *pub, *perr;
TP[illegible]
char [illegible]
[illegible]
  [illegible]
  if ([illegible](tp,dptr,FALSE,OPTIONAL))->type == INTEGER
      [illegible]
    [illegible]
      [illegible] = error([illegible],tp,NONE,dptr,"");
      [illegible]
    }
  if ([illegible](tp,dptr,FALSE,OPTIONAL))->type == INTEGER)
    [illegible]
  [illegible]
      [illegible] = error([illegible],tp,[illegible],dptr,"");
      [illegible]
    }

  if ([illegible]
    [illegible]
    [illegible]
    [illegible]
  }

  re[illegible]
}

TPTR [illegible],perr,code,members)
FILE *dptr;
TP[illegible]
[illegible]
int *p[illegible];
[illegible]
[illegible] [illegible]], *cp, p2, *prior();
  int [illegible]
  [illegible]_token(), error(), object_dec();

  [illegible]uct",SP);
  [illegible]
  [illegible]",FALSE);
  [illegible](tp,dptr,FALSE,FALSE))->token,")"))
  [illegible]
  [illegible]
  [illegible],pclass,perr,code,members);
  [illegible]
  [illegible],dptr,FALSE,FALSE))->token,"(") ||
```

```
      [illegible] get_token(tp,dptr,FALSE,TRUE))->line != j)
        { [illegible] = [illegible]_token(tp);
          [illegible]
        }
      [illegible] = prior(tp);
      [illegible],FALSE);
    }
    move[illegible]",[illegible]);
    for [illegible]->line; j [illegible] tp->line;
          [illegible](tp->prior->prior->token,",") ? j : tp->line)
      { move[illegible]tp->token,FALSE);
          [illegible]_token(tp,dptr,FALSE,TRUE);
      }
    [illegible]_token(tp);
    [illegible]",FALSE);
  }
  else
    move[illegible]                  ",FALSE);
}
cp = prior(tp);
if (!strcmp([illegible]))
  { [illegible] = [illegible](CP,[illegible],NONE,dptr,pclass);
    [illegible] = [illegible];
  }
else
  { move[illegible]");
    move[illegible],FALSE);
    move[illegible]);
  }

return(tp);
}

rite(linked_list,cptr)
LLPTR linked_list;
FILE *cptr;
{ int i = -1;

  while (linked_list->array[++i] && i <= LINESIZE)
    puts(linked_list->array[i],cptr);
  if (i > LINESIZE)
    rite(linked_list->next,cptr);
}

LLPTR lalloc()
{ LLPTR linked_array_list;
  int i;
  char *calloc();

  linked_array_list = (LLPTR)calloc(1,sizeof(LLNODE));
  for (i = 0; i < LINESIZE+1; i++)
    linked_array_list->array[i] = '\0';
  linked_array_list->next = NULL;

  return(linked_array_list);
}

move(linked_array_list,string,last_char)
LLPTR linked_array_list;
char *string;
int last_char;
{ int i = -1;
  LLPTR lalloc();
```

```
if (linked_array_list != NULL)
    while (linked_array_list->array[++i] && i < LINESIZE);
    if (linked_array_list->array[i])
        [illegible]_list->next,string,last_char);
    else {
        while ((linked_array_list->array[i++] = *string++) && i <= LINESIZE);
        if ([illegible]string[illegible])
            [illegible]_list->next = lalloc();
            [illegible]_array_list->next,string,last_char);
        }
        else
            if ([illegible]_char)
                [illegible]_array_list->array[i-1] = last_char;
            }
            [illegible]
            if ([illegible] LINESIZE)
                [illegible]_list->next = lalloc();
        }
    }
}

SPTR add_class(root,name,members,lb,ub,pi)
[illegible]
char [illegible]
[illegible]members;
int lb, ub, *pi;
[illegible]alloc();

    *pi = TRUE;
    if (root == NULL)
        { root = [illegible]calloc(1,sizeof(SNODE));
            strcpy(root->[illegible]name);
            [illegible]members;
            root->lb = lb;
            root->ub = ub;
            [illegible]root->right = NULL;
            [illegible]
        }
    else if ([illegible](name,root->name) > 0)
        [illegible](root->right,name,members,lb,ub,pi);
    else [illegible](name,root->name) < 0)
        [illegible](root->left,name,members,lb,ub,pi);

    return [illegible];
}

VP [illegible](root,name,type,pointer,array,pi)
[illegible]
[illegible]
[illegible]
[illegible]

    *pi = [illegible];
    if (root == [illegible])
        { root = [illegible]calloc(1,sizeof(VNODE));
            strcpy(root->[illegible]name);
            [illegible]type);
            [illegible]pointer;
            [illegible]
            [illegible]root->right = NULL;
            [illegible]
        }
    else [illegible](name,root->name) > 0)
```

```
    [illegible] = [illegible]_var(root->right,name,type,pointer,array,pi);
    [illegible] name) < 0)
    [illegible] left,[illegible],type,pointer,arr[illegible]pi);

    [illegible]

FO[illegible]  [illegible](name,intype,outtype,pi)
[illegible]

    [illegible] NULL)
        [illegible] alloc(1,sizeof(FONODE));
        [illegible] name);
        [illegible] intype);
        [illegible] outtype);
        [illegible] root->result = NULL;
        [illegible]
        [illegible] right = NULL;
        [illegible]
    }
    el[illegible] (name,root->name) > 0)
        [illegible](root->right,name,intype,outtype,pi);
    el[illegible] root->name) < 0)
        [illegible](root->left,name,intype,outtype,pi);
    el[illegible](intype,root->intype) > 0)
        [illegible](root->right,name,intype,outtype,pi);
    el[illegible](intype,root->intype) < 0)
        [illegible](root->left,name,intype,outtype,pi);
    el[illegible](outtype,root->outtype) > 0)
        [illegible](root->right,name,intype,outtype,pi);
    el[illegible](outtype,root->outtype) < 0)
        [illegible](root->left,name,intype,outtype,pi);

    re[illegible](root);
}

FP[illegible](name,root)
[illegible]

    [illegible] root;
    if [illegible] NULL)
        ;
    el[illegible](name,root->name) > 0)
        [illegible](name,root->right);
    el[illegible](name,root->name) < 0)
        [illegible](name,root->left);

    re[illegible]
}

BP[illegible](name,root)
[illegible]

    [illegible] root;
    if [illegible] NULL)
        ;
```

```
    else if (strcmp(name,root->name) > 0)
      file = find_[illegible](name,root->right);
    [illegible] if [illegible](name,root->name) < 0)
      [illegible] = [illegible](name,root->left);

    return [illegible];
}

FLPTR find_file(name,root)
char *name;
FLPTR root;
{ FLPTR file;

    file = root;
    if (root == NULL)
      [illegible]
    else if (strcmp(name,root->name) > 0)
      file = find_file(name,root->right);
    else if (strcmp(name,root->name) < 0)
      file = find_file(name,root->left);

    return (file);
}


VPTR [illegible](dptr,tp,ft,sos,perr,code,members)
[illegible]
[illegible]
[illegible]
[illegible]
FLPTR [illegible];
{
    [illegible] get_[illegible](), unget_token(), declaration(), error(), get_class();
    TPTR [illegible]();
    char [illegible][LINESIZE+1], class[LINESIZE+1], *calloc();
    VPTR vptr;

    vptr = (VPTR)calloc(1,sizeof(VNODE));
    [illegible](dptr,tp,ft,sos,perr,code,class,vptr);
    if (![illegible])
      if (!strcmp(class,sos))
        [illegible]("struct",SP);
        [illegible](class,SP);
        tp = [illegible]_colon(tp,dptr,"::");
        for ([illegible](string,","); !strcmp(string,",") && !(*perr);
             strcpy(string,(tp = get_token(tp,dptr,FALSE,TRUE))->token))
          { tp = declaration(dptr,tp,ft,perr,code,members,vptr->pointer,
                             vptr->array);
            [illegible](members,class,SP);
          }
        if (!(*perr))
          [illegible]_colon(tp);
    }
    else
      [illegible];

    return (tp);
}

TPTR [illegible](dptr,tp,ft,perr,code,members,pointer,array)
[illegible]
int *perr, *pointer, *array;
[illegible]
[illegible]
```

```
_token(), expression(), error();

,dptr,FALSE,OPTIONAL))->token,"#"))
FALSE,OPTIONAL);

type != OPERATOR && tp->type != LITERAL)
dptr,"");

get_token(tp,dptr,FALSE,TRUE))->token,"["))

_token(tp,dptr,FALSE,OPTIONAL))->token,"]"))
(tp);
ptr,tp,code);

(tp,dptr,FALSE,OPTIONAL))->token,""))
dptr,"");

is_class(tp->token,"",position,fptr))

tr,"");
,OPTIONAL);
```

```
    [illegible] != [illegible] && tp->type != OPERATOR)
        [illegible] (&[illegible]p,NON[illegible],&tr,str);
        [illegible],OPTIONAL);

  else
    tp = [illegible]
    [illegible](tr,FALSE,TRUE))->token,"="))
      [illegible]
        [illegible],tp,co[illegible]);
        [illegible] [illegible]dptr,"");

  else
    [illegible]
    [illegible]dptr,"");

  return [illegible]

[illegible],class,vptr)
[illegible]

  tp [illegible],FALSE,FALSE);
  [illegible]
  if [illegible]position,fptr) || !strcmp(tp->token,"@") ||
    [illegible]
      [illegible]n);
      [illegible]"))
        [illegible](dptr,tp,lname,code);
        [illegible]lname)
        [illegible],tp,NONE,dptr,"");
        [illegible]
    [illegible]position,fptr) && !(*perr))
      [illegible],tp,[illegible]_LINE,dptr,"");
      [illegible]
  else
    [illegible]

is_[illegible]lass,fptr)
[illegible]

  if [illegible],fptr))
```

```
[illegible] root->left);
[illegible] cmp(intype,""))
[illegible] root->right);
[illegible] (intype,""))
[illegible] root->left);
[illegible] strcmp(intype,""))
[illegible] root->right);
[illegible] strcmp(intype,""))
[illegible] root->left);

[illegible] f[illegible];
}

[illegible] code)
[illegible]

[illegible] logical_name(), error();
[illegible] params();
[illegible]
[illegible] SIZE+1];
[illegible] find_fo[illegible];

[illegible]
[illegible] (tp->line)
[illegible]
[illegible] type == COMMA)
[illegible] (tp,dptr,FALSE,FALSE))->prior->type != LTERM)
[illegible];
[illegible]
[illegible] n(tp);
[illegible] FALSE;
[illegible] '\0';
[illegible] en(tp,dptr,FALSE,OPTIONAL))->type != LITERAL
[illegible] != INTEGER)
[illegible] (tp->token,"\""))
[illegible] (dptr,tp,code);
[illegible] (tp->token,"'"))
[illegible] (dptr,tp,code);
[illegible] (tp->token,"@"))
[illegible] (dptr,tp,lname,code);
[illegible] (tp->token,"("))
[illegible] (dptr,tp,code);
[illegible] (tp->token,"null"))
;
[illegible] (tp->token,"accept"))
[illegible] (tp,dptr,FALSE,OPTIONAL))->type!=NAME)
[illegible] dptr,"");
;
[illegible] (tp->token,"","",position,fptr->fodec))!=NULL)
[illegible] (intype,"null"))
[illegible] (dptr,tp,code);
;
[illegible] (tp->token,"*"))
[illegible] (dptr,tp,code);
[illegible] || !strcmp(tp->token,"#"))
[illegible] n(tp);
[illegible] (dptr,tp,lname,code);
;
```

```
  return(1);
  [illegible] "null") || !strcmp(classname,[illegible]nt") ||
  [illegible] !strcmp(classname,[illegible]pclass))
    return(1);
  else
    return(0);
}

in_file(classname,fptr)
[illegible]
{
  [illegible]
  if [illegible] NULL)
  [illegible]
  else if [illegible]classname,fptr->vdef) != NULL)
    return(1);
  [illegible]
}

SPTR [illegible]class,root)
[illegible]
{
  [illegible] root;
  if [illegible] NULL)
  [illegible]
  else if [illegible](class,root->name) > 0)
    [illegible] class,root->right);
  else if [illegible] root->name) < 0)
    [illegible] root->left);
  [illegible]
}

VPTR find_var([illegible],root)
[illegible]
{
  VPTR [illegible] root;
  if [illegible] NULL)
  [illegible]
  else if [illegible] root->name) > 0)
    [illegible] root->right);
  else if [illegible] root->name) < 0)
    [illegible] root->left);
  [illegible]
}

FOPTR find_[illegible],intype,outtype,root)
[illegible]
{
  [illegible] root;
  if [illegible] NULL)
  [illegible]
  else if [illegible](fo,root->name) > 0)
    [illegible] find_fo(fo,intype,outtype,root->right);
```

```
        else if (tp->type == OPERATOR)
          if (!strcmp(tp->token,"*") || !strcmp(tp->token,"-"))
            tp = [illegible]sion(dptr,tp,code);
          else
            tp = [illegible](11,tp,NONE,dptr,"");
        else
          { tp = error(14,tp,END_LINE,dptr,"");
            [illegible] = T[illegible];
            [illegible];
          }
        i = tp->line;
      } while ([illegible] get_token([illegible],dptr,FALSE,TRUE))->type == OPERATOR);
    if ([illegible])
      tp = [illegible]_token(tp);

  return(tp);
}

TPTR dbl_quote(dptr,tp,code)
FILE *dptr;
TPTR tp;
LLPTR code;
{ int i;

  strcpy(tp->token,"\"");
  for (i = tp->line; i == tp->line && strcmp(tp->token,"\"");
        i = strcmp(tp->prior->prior->token,",") ? i : tp->line)
    tp = get_token(tp,dptr,FALSE,TRUE);
  if (strcmp(tp->token,"\""))
    { tp = error(15,tp,NONE,dptr,"");
      tp = [illegible]_token(tp);
    }

  return(tp);
}

TPTR [illegible]_quote(dptr,tp,code)
FILE *dptr;
TPTR tp;
LLPTR code;
{ if ([illegible](tp = get_token(tp,dptr,FALSE,TRUE))->token) > 1)
    tp = error(16,tp,NONE,dptr,"");
  if ([illegible](tp = get_token(tp,dptr,FALSE,FALSE))->token,"'"))
    { tp = error(17,tp,NONE,dptr,"");
      tp = [illegible]_token(tp);
    }

  return(tp);
}

TPTR [illegible](dptr,tp,code)
FILE *dptr;
TPTR tp;
LLPTR code;
{ [illegible] expression();

  tp = expression(dptr,tp,code);
  if ([illegible](tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,")"))
    { tp = error(18,tp,NONE,dptr,"");
      tp = [illegible]_token(tp);
    }

  return(tp);
}
```

```
TPTR logical_name(dptr,tp,lname,code)
DPTR *dptr;
TPTR tp;
char *lname;
CLPTR code;
{
  TPTR get_token(), unget_token(), expression(), error();

  if ((tp = get_token(tp,dptr,FALSE,OPTIONAL))->type == NAME)
    { check_pname(tp->token);
      strcat(lname,tp->token);
      while(!strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,"["))
        { strcat(lname,"[]");
          if (strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,"]"))
            { tp = unget_token(tp);
              tp = expression(dptr,tp,code);
            }
          else
            tp = unget_token(tp);
          if (strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,"]"))
            { tp = error(17,tp,NONE,dptr,"");
              tp = unget_token(tp);
            }
        }
      if (!strcmp(tp->token,"."))
        { strcat(lname,".");
          tp = logical_name(dptr,tp,lname,code);
        }
      else
        tp = unget_token(tp);
    }
  else if (!strcmp(tp->token,"*"))
    { strcat(lname,"*");
      tp = logical_name(dptr,tp,lname,code);
    }
  else
    tp = error(11,tp,CUR_LINE,dptr,"");

  return(tp);
}

check_pname(name)
char *name;
{
  if (look_name_up(position.bptr->rtree,name))
    return(1);
  else if (position.foptr != NULL && look_name_up(position.foptr->rtree,name))
    return(1);
  else if (position.fptr != NULL && look_name_up(position.fptr->rtree,name))
    return(1);
  else
    return(0);
}

look_name_up(root,name)
RPTR root;
char *name;
{ int i;

  if (root == NULL)
    i = 0;
  else if (strcmp(root->name,name) > 0)
    i = look_name_up(root->right,name);
  else if (strcmp(root->name,name) < 0)
```

```
    i = look_name_up(root->left,name);
    
      strcpy(l  e,root->lname);
      i = 1;
    }

  return(i);
}

determine_class(class,lname)
VPTR class;
char *lname;
{ int pointer, array, point, arr, func;
  char *cp, temp[LINESIZE+1], member[LINESIZE+1], inclass[LINESIZE+1];
  char outclass[LINESIZE+1];
  VPTR vptr = NULL;
  VPTR find_var();
  SPTR sptr = NULL;
  SPTR find_class();
  LLIST member , find_member();

  pointer = array = point = arr = func = Ø;
  cp = temp;
  while (*lname == '*')
        { pointer; ++lname; }
  while ((*cp = *lname) >= 'A' && *cp <= 'Z' || *cp >= 'a' && *cp <= 'z' ||
            *cp >= 'Ø' && *cp <= '9' || *cp == '_')
    { ++cp; ++lname; }
  *cp = '\Ø';
  if (position.sptr->type == ACTOR && position.foptr != NULL)
    if ((vptr = find_var(temp,position.foptr->arg1)) == NULL)
      if ((vptr = find_var(temp,position.foptr->arg2)) == NULL)
        vptr = find_var(temp,position.foptr->result);
  if (vptr == NULL)
    if ((vptr = find_var(temp,position.fptr->vdec)) == NULL)
      return(1);
  strcpy(outclass,vptr->type);
  pointer = vptr->pointer - pointer;
  while (*lname == '[')
    { lname+=2; array++; }
  if (pointer < Ø)
    return(2);
  else if (pointer > Ø && *lname != Ø)
    return(3);
  else if (array > vptr->array)
    return(4);
  else if (*lname == '\Ø')
    { strcpy(class->name,"");
      strcpy(class->type,vptr->type);
      class->pointer = pointer;
      class->array = vptr->array - array;
      class->left = class->right = NULL;
      return(Ø);
    }
  else
    do { array = Ø;
           if ((sptr = find_class(outclass,position.fptr->vdef)) == NULL)
             return(5);
           else
             { lname++;
               cp = temp;
               while (*lname == '*')
                 { ++pointer; ++lname; }
               while ((*cp = *lname)>='A' && *cp<='Z' || *cp>='a' && *cp<='z' ||
```

```
                *cp>='Ø' && *cp<='9' || *cp=='_')
              { ++cp; ++lname; }
            *[illegible] = '\Ø';
            members = [illegible]ptr->members;
            while ([illegible] = find_member(member,inclass,outclass,&point,&arr,
                                             &func,member ))
              if (!strcmp(temp,member))
                break;
            if ([illegible] == NULL)
              return([illegible]);
            else
              { pointer = point - pointer;
                while (*lname == '[')
                  { lname+=2; array++; }
                if (pointer < Ø)
                  return(7);
                else if (pointer > Ø && *lname != '\Ø')
                  return(8);
                else if (array > arr)
                  return(9);
              }
          }
        } while (*lname != '\Ø');
    strcpy(cla[illegible],"");
    strcpy(cla[illegible]outclass);
    [illegible]pointer = pointer;
    [illegible]array;
    [illegible]right = NULL;
    return([illegible]);
}


TPTR [illegible]ctor(dptr,cptr,tp)
FILE *dptr, *[illegible];
TPTR tp;
{
  TPTR get_token(), function_declaration(), operator_declaration(), linkage();
  TPTR unget_token(), error(), block();
  LPTR [illegible] = NULL;
  char fo_in[LINESIZE+1], fo_out[LINESIZE+1];
  int err;
  [illegible] code, members, lalloc();

  while (![illegible]_file((tp = get_token(tp,dptr,FALSE,FALSE))->token) &&
          [illegible](tp->token,"."))
    { strcpy(fo_in,"");
      strcpy(fo_out,"");
      err = FALSE;
      code = [illegible]lloc();
      members = lalloc();
      strcpy([illegible]ition.fo_name,tp->token);
      if ([illegible]type == NAME)
        { tp = function_declaration(dptr,tp,&err,code,fo_in,fo_out);
          [illegible]position.fo_name,fo_in,fo_out,code,cptr);
        }
      else if (tp->type == OPERATOR)
        { tp = [illegible]declaration(dptr,tp,&err,code,fo_in,fo_out);
          [illegible]position.fo_name,fo_in,fo_out,code,cptr);
        }
      else
        { tp = error(7,tp,END_DEF,dptr,"");
          continue;
        }
      tp = [illegible](dptr,tp,5);
```

```
    tp = pl[illegible](gptr,dptr,tp.l.listhead);
  [illegible] unget_token(tp);
  [illegible](tp);
}

TPIR function_declaration(dptr,tp,perr,code,f_in,f_out)
FIL[illegible]
[illegible]
int [illegible];
LIST [illegible];
char [illegible] f_out;
{
  [illegible] get_to[illegible](), [illegible]_colon(), unget_token(), declaration(), error();
  [illegible] [illegible]d_members();
  [illegible] add_[illegible]();
  [illegible]
  [illegible]
  [illegible] [illegible]1], [illegible]ls1[LINESIZE+1], outcls1[LINESIZE+1];
  [illegible] function1;
  [illegible] 1], [illegible]3[LINESIZE+1], outcls3[LINESIZE+1];
  [illegible] function3;

  [illegible](tp,dptr,"::");
  if [illegible]_token(tp,dptr,FALSE,OPTIONAL))->token,"in"))
      [illegible]_token(tp,dptr,FALSE,OPTIONAL))->token,
      [illegible]tr))
      tp [illegible],NONE,dptr,"");
      [illegible]();
      [illegible]->token);
      [illegible]1);
      [illegible]_colon(tp,dptr,"::");
      [illegible]dptr,tp,ACTOR,perr,code,members,NULL,NULL);
      [illegible]1,SP);
      [illegible]member1,incls1,outcls1,&pointer1,&array1,
                      &function1,members);
  }
  else
      [illegible]1,"null");
      [illegible](tp);
  }
  if ([illegible]_token(tp,dptr,FALSE,FALSE))->token,"out"))
      [illegible]_token(tp,dptr,FALSE,OPTIONAL))->token,
      [illegible]tr))
      tp [illegible],NONE,dptr,"");
      [illegible]();
      [illegible]->token);
      [illegible]3);
      [illegible]_colon(tp,dptr,"::");
      [illegible]dptr,tp,ACTOR,perr,NULL,members,NULL,NULL);
      [illegible]3,SP);
      [illegible]member3,incls3,outcls3,&pointer3,&array3,
                      &function3,members);
  }
  else
      [illegible]3,"null");
      [illegible](tp);
  }
  p[illegible] = add_fo(position,fptr->fodef,position.fo_name,
                      outcls1,outcl[illegible]);
  if [illegible]
    tp [illegible](17,tp,NONE,dptr,"");
  p[illegible]_fo(position.fo_name,outcls1,outcls3,
```

```
                        position.fptr->fodef);
    if (strcmp(outcls1,"null"))
      position.dptr->varl = add_var(position.foptr->arg1,member1,outcls1,
                                  pointer1,array1,defined);
    if (strcmp(outcls3,"null"))
      position.dptr->varlt = add_var(position.foptr->result,member3,
                                  outcls3,pointer3,array3,defined);

  return(tp);
}

TPTR operator_declaration(dptr,tp,perr,code,o_in,o_out)
FILE *dptr;
TPTR tp;
int *perr;
LLPTR code;
char *o_in, *o_out;
{
  TPTR get_token(), check_colon(), unget_token(), declaration(), error();
  FLPTR members, lalloc(), find_member();
  FOPTR add_fo(), find_fo();
  VPTR add_var();
  int unary_prefix = FALSE;
  int binary = FALSE;
  int defined;
  char member1[LINESIZE+1], incls1[LINESIZE+1], outcls1[LINESIZE+1];
  int pointer1, array1, function1;
  char member2[LINESIZE+1], incls2[LINESIZE+1], outcls2[LINESIZE+1];
  int pointer2, array2, function2;
  char member3[LINESIZE+1], incls3[LINESIZE+1], outcls3[LINESIZE+1];
  int pointer3, array3, function3;

  tp = check_colon(tp,dptr,":");
  if (strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,"in"))
    tp = error(33,tp,END_LINE,dptr,"in");
  else
    { if (!is_class((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,
           "",position.dptr))
        tp = error(28,tp,NONE,dptr,"");
      members = lalloc();
      strcpy(outcls1,tp->token);
      strcpy(o_in,outcls1);
      tp = check_colon(tp,dptr,"::");
      if (strcmp((tp = get_token(tp,dptr,FALSE,TRUE))->token,","))
        { tp = unget_token(tp);
          tp = declaration(dptr,tp,ACTOR,perr,code,members,NULL,NULL);
          move(members,outcls1,SP);
          if (!strcmp((tp = get_token(tp,dptr,FALSE,TRUE))->token,","))
            { tp = declaration(dptr,tp,ACTOR,perr,code,members,NULL,NULL);
              move(members,outcls1,SP);
              binary = TRUE;
            }
          else
            tp = unget_token(tp);
        }
      else
        { tp = declaration(dptr,tp,ACTOR,perr,code,members,NULL,NULL);
          move(members,outcls1,SP);
          unary_prefix = TRUE;
        }
      if (!unary_prefix)
        member1 = find_member(member1,incls1,outcls1,&pointer1,&array1,
                              &function1,members);
      if (binary || unary_prefix)
```

```
        member2 = find_member(member2,incls2,outcls2,&pointer2,&array2,
                              &function2,members);

    if (strcmp((tp = get_token(tp,dptr,FALSE,FALSE))->token, "out"))
      tp = error(2,tp,END_LINE,dptr,"out");
    else
      { if (!is_class(tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,
            "",tp,dfile,dptr))
          tp = error(25,tp,NONE,dptr,"");
        members = 1;
        strcpy(outcls3,tp->token);
        strcpy(fo_out,outcls3);
        tp = find_colon(tp,dptr,"::");
        tp = decl_call(dptr,tp,ACTOR,perr,NULL,members,NULL,NULL);
        move(members,outcls3,SP);
        member3 = find_member(member3,incls3,outcls3,&pointer3,&array3,
                              &function3,members);
      }
    position.fptr->fodef = add_fo(position.rptr->fodef,position.fo_name,
                                  outcls1,outcls3,&defined);
    if (defined)
      tp = error(37,tp,NONE,dptr,"");
    position.feptr = find_fo(position.fo_name,outcls1,outcls3,
                             position.fptr->fodef);
    if (!unary_prefix)
      position.feptr->arg1 = add_var(position.feptr->arg1,member1,outcls1,
                                     pointer1,array1,&defined);
    if (binary || unary_prefix)
      position.feptr->arg2 = add_var(position.feptr->arg2,member2,outcls2,
                                     pointer2,array2,&defined);
    position.feptr->result = add_var(position.feptr->result,member3,
                                     outcls3,pointer3,array3,&defined);

  return(tp);
}

gen_fo_oper(fo,fo_in,fo_out,code,cptr)
char *fo, *fo_in, *fo_out;
CLPTR code;
FILE *cptr;
{ if (*fo_out) fprintf(cptr,"%s ",fo_out);
  fprintf(cptr,"%s(",fo);
  ...(fo_out,cptr);
  fprintf(cptr,")\n");
  if (*fo_in) fprintf(cptr,"%s ",fo_in);
  ...(code,cptr);
  fprintf(cptr,"\n{\n");
  fprintf(cptr,"  /*********************************************************/\n");
  fprintf(cptr,"  /*                                                       */\n");
  fprintf(cptr,"  /*            function or operator body                  */\n");
  fprintf(cptr,"  /*                                                       */\n");
  fprintf(cptr,"  /*********************************************************/\n");
  fprintf(cptr,"\n}\n");
}

TPTR check_colon(tp,dptr,s)
TPTR tp;
FILE *dptr;
char *s;
{
  TPTR get_token(), unget_token(), error();

  if (strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,":") &&
      strcmp(tp->token,"::"))
```

```
    { tp = error(21,tp,NONE,dptr,s);
      tp = [illegible]);
    }
  if ([illegible]->token,s))
    tp = error([illegible],dptr,s);

  return(tp);
}


TPTR block([illegible],dptr,tp,level,listhead)
FILE [illegible];
int [illegible];
TPTR [illegible];
LPTR [illegible]head;
{
  TPTR [illegible](), [illegible]t_token(), expression(), check_colon();
  TPTR [illegible], error[illegible]);
  int [illegible]guard, [illegible], typ;
  LPTR [illegible]();
  char [illegible];
  LLPTR [illegible];

  if ([illegible]_token(tp,dptr,FALSE,FALSE))->type == NAME)
    { listhead = [illegible](tp->token,level,listhead,&in_list);
      if (in_list)
        tp = [illegible],NONE,dptr,"");
      tp = [illegible]_colon(tp,dptr,"::");
      if ([illegible](tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,"?"))
        tp = [illegible](tp,dptr,FALSE,OPTIONAL);
      if ([illegible](tp->token,"("))
        { tp = [illegible]_token(tp);
          tp = [illegible](dptr,tp,code);
          tp = get_tok[illegible](tp,dptr,FALSE,OPTIONAL);
        }
    }
  if ([illegible](tp->token,"{"))
    { [illegible](26,tp,NONE,dptr,"");
      tp = [illegible]_token(tp);
    }
  tp = [illegible](dptr,tp,0);
  while ([illegible](tp = get_token(tp,dptr,FALSE,FALSE))->token,"}"))
    { [illegible];
      typ = tp->type;
      [illegible](str,tp->token);
      if(((![illegible](tp = get_token(tp,dptr,FALSE,OPTIONAL))->token,":")
          || ![illegible](tp->token,"::")) && typ == NAME) || !strcmp(str,"(")
          || ![illegible](str,"new") || !strcmp(str,"return"))
        { tp = [illegible]_token(tp);
          tp = [illegible]_token(tp);
          [illegible] FALSE;
        }
      else
        { tp = unget_token(tp);
          tp = unget_token(tp);
          tp = [illegible](dptr,tp,code);
        }
      tp = get_tok[illegible],dptr,FALSE,OPTIONAL);
        ([illegible](tp->token,"%") || !guard)
        { [illegible](tp->token,"%"))
            tp = unget_token(tp);
          if ([illegible](tp = get_token(tp,dptr,FALSE,FALSE))->
              token,"[illegible]turn"))
            if ([illegible]((tp = get_token(tp,dptr,FALSE,OPTIONAL))->
```

```
                token,level,listhead) && !is_contn(tp->token) &&
                strcmp(tp->token,")"))
              { tp = unget_token(tp);
                tp = [illegible](dptr,tp,code);
              }
            else
              tp = unget_token(tp);
          else if (!strcmp(tp->token,"new"))
            { if ((tp = get_token(tp,dptr,FALSE,OPTIONAL))->type!= NAME)
                tp = [illegible]
              [illegible](tp->token);
              if (!strcmp((tp = get_token(tp,dptr,FALSE,FALSE))->
                token,"="))
                tp = [illegible]pression(dptr,tp,code);
              else
                tp = unget_token(tp);
            }
          else
            { typ = tp->type;
              strcpy(str,tp->token);
              if (!strcmp((tp = get_token(tp,dptr,FALSE,OPTIONAL))->
                  token,":") || !strcmp(tp->token,"::")) && typ == NAME)
                  || !strcmp(str,"("))
              { tp = unget_token(tp);
                tp = unget_token(tp);
                tp = [illegible](dptr,dptr,tp,level+1,listhead);
              }
              else
              { tp = unget_token(tp);
                strcpy(str,tp->token);
                tp = unget_token(tp);
                if ([illegible],level,listhead) && !is_contn(str))
                  tp = [illegible](dptr,tp,code);
              }
            }
        }
      else
        [illegible]
          [illegible](tp,dptr,FALSE,OPTIONAL))->token,
          [illegible]
          [illegible](tp,dptr,FALSE,OPTIONAL))->token))
          [illegible]
            [illegible]_token(tp);
          [illegible]
            [illegible]token[illegible])
          [illegible]
            [illegible]_token(tp);
        }
    return [illegible];
}

LPTR [illegible],level,listhead,pi)
char [illegible];
int [illegible];
[illegible]
{
  LPTR [illegible];
  char [illegible];

  *pi = FALSE;
  if ([illegible] NULL)
```

```
    if (listhead = ptr = (LPTR)calloc(1,sizeof(LNODE));
    else
        { for (ptr = listhead; ptr->next != NULL && strcmp(ptr->label,label);
              ptr = ptr->next);
          if (!strcmp(ptr->label,label))
            *p[illegible]
          else
            { ptr->next = (LPTR)calloc(1,sizeof(LNODE));
              ptr = ptr->next;
            }
        }
    if (!(*p[illegible]))
        { strcpy(ptr->label,label);
          [illegible]
          ptr->next = NULL;
        }

    return(listhead);
}

is_l[illegible](label,level,listhead)
char [illegible];
int [illegible];
LPTR listhead;
{
    LPTR ptr;

    if (listhead == NULL)
        return(1);
    else
        { for (ptr = listhead;
               ptr != NULL && (strcmp(ptr->label,label) || ptr->level > level);
               ptr = ptr->next);
          if (ptr == NULL)
            return(0);
          else
            return(1);
        }
}

is_com[illegible](token)
char [illegible];
{
    if (!strcmp(token,":") || !strcmp(token,"..") || !strcmp(token,"\\"))
        return(1);
    else
        return(0);
}

TP[illegible] [illegible]_te[illegible](tp,fp,end,comma)
[illegible]
FILE [illegible];
int [illegible];
[illegible] int line = 1;
[illegible] int [illegible] = 0;
int [illegible];
in[illegible]
TP[illegible] [illegible]();

if [illegible] NULL)
    [illegible]
    [illegible]prior;
    [illegible] (tp->type == COMMA || tp->type == LTERM);
```

```
      == COMMA && comma != TRUE || tp->type == LTERM);

                                              == CP);

                      || tp->tp...  == COMMA)
            (id,tp,NONE,fp,  );

    if (comma != TRUE)
                 (...end,comma);
                                   OPTIONAL && tp->type!=LTERM)
            (id,tp,NONE,fp,"");

                                 == FF)

                  (...end,comma);
                         || c == '_')

                         'a' && c <= 'z')
                         p,fp);

             (id,tp,"");
             (...end,comma);

                              ? punc(i,tp,fp) : op(i,tp,fp);
  }

  return
}

punc(i,c,tp,fp)
int
FPTR
FILE *fp;
{
  if (c != '!' && c != '%' && c != '&' && c != '*' && c != '+' && c != '-' &&
                     != '<' && c != '>' && c != '^' && c != '|')
                   getc(fp)) != ' ')
         (c, fp);
```

```
        ... >= '0' && c <= '9' || c == '_')
        ... || c >= 'a' && c <= 'z')
        ... >= '0' && c <= '9' || c == '_' ||
        ... && c <= 'z')

    un_get(c, fp);

}

int i;

        ... >= '0' && c <= '9' || c == '_' ||
        ... 'a' && c <= 'z')

    ... (tp->token))

        ... && strcmp(s,"actor")   && strcmp(s,"bundle")  &&
        ...  ,"in")         && strcmp(s,"new")      &&
        ...  ,"out")        && strcmp(s,"rename")   &&
        ...                 && strcmp(s,"state")    &&

    if ( ... tp->token[i-1] == ':' || tp->token[i-1] == '.')
        ... == tp->token[i-1])
```

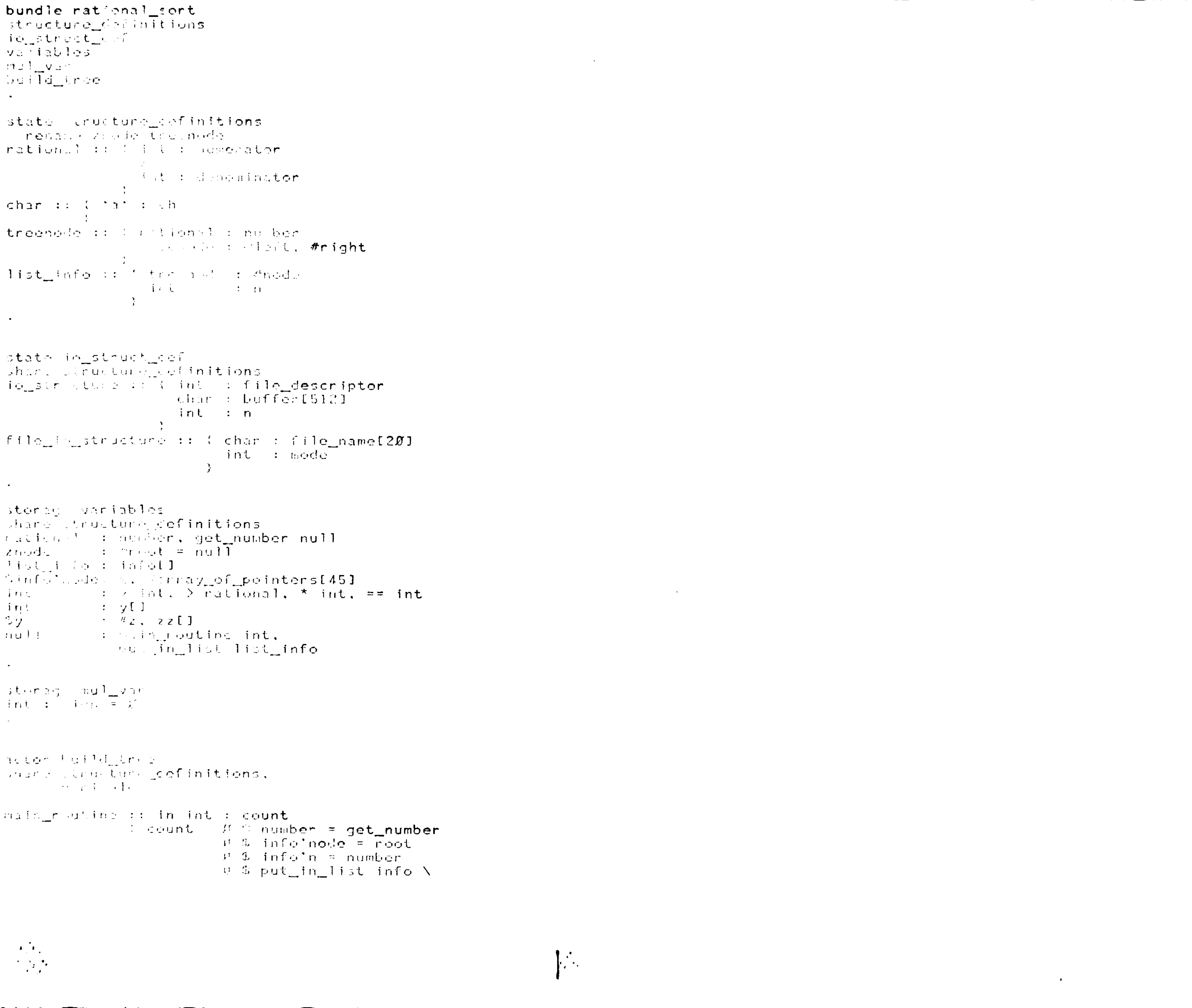
```
[illegible]

    [illegible]_NAME_SIZE-1; j++)
    [illegible] c != '[illegible]' && c != '&' && c != '*' &&
    [illegible]
    [illegible])
    [illegible]
        tp[illegible][i++] = c;

  [illegible] '\0';

  [illegible]
}

TPTR [illegible] (tp)
TPTR [illegible]
{
  [illegible](), NULL, NULL, NULL);

  [illegible]
}

TPTR [illegible](i,tp,rstrt,dptr,string)
int i, rstrt;
TPTR [illegible]
FILE [illegible]ptr;
char *string;
[illegible]

  if ([illegible] i != [illegible])
    [illegible]: ",tp->line);
  [illegible]
  [illegible],rstrt,dptr,string);

  [illegible]
}

print[illegible](tp,string)
[illegible]

    [illegible]
      [illegible]
        [illegible]intf("premature Unix file termination\n"); break;
```

```
case 2  : printf("line %d: line terminator not valid at end\n",tp->line-1);
          break;
[illegible] : [illegible] "[illegible] illegal character, ACSII value %d. ",tp->token[Ø]);
          [illegible] encountered\n"); break;
[illegible] 4 : [illegible] line terminator required before '%s'\n",tp->token); break;
[illegible] 5 : [illegible] file type expected '%s' received\n",tp->token); break;
[illegible] 6 : [illegible] duplicate file name '%s'\n",tp->token); break;
[illegible] 7 : [illegible] label name '%s'\n",tp->token); break;
[illegible] 8 : [illegible] illegal keyword '%s' in linkage encountered\n",tp->token);
          [illegible]
[illegible] 9 : [illegible] file name expected '%s' received\n",tp->token); break;
[illegible] 10 : [illegible] comm[illegible] permitted\n"); break;
[illegible] 11 : [illegible] file name or object class expected '%s' ",tp->token);
          [illegible] received\n"); break;
[illegible] 12 : [illegible] terminating period not detected\n",string); break;
[illegible] 13 : [illegible] '' after %s\n",cp); break;
[illegible] 14 : [illegible] precision: token received was %s\n",tp->token);
          [illegible]
[illegible] 15 : [illegible] '\"' after %s\n",cp); break;
[illegible] 16 : [illegible] never [illegible] quote must be a ",tp->token);
          [illegible] character\n"); break;
[illegible] 17 : [illegible] '' after %s\n",cp); break;
[illegible] 18 : [illegible] '' after %s\n",cp); break;
[illegible] 19 : [illegible] not generated when '%s' received\n",tp->token);
          break;
[illegible] 20 : [illegible] illegal use of ");
          switch (tp->type) {
            case LITERAL     : printf("literal "); break;
            case INTEGER     : printf("integer "); break;
            case KEYWORD     : printf("keyword "); break;
            case PUNCTUATION : printf("punctuation "); break;
            case OPERATOR    : printf("operator "); break;
          }
          printf("[illegible] file [illegible]\n",[illegible] token); break;
case 21 : [illegible]
            printf([illegible] ':' [illegible] %s\n",cp);
          [illegible]
            printf([illegible] ':' after %s\n",cp);
          [illegible]
          [illegible]
            printf([illegible] ':' [illegible] '::'\n");
          [illegible]
            printf([illegible] ':' should be '::'\n");
          break;
[illegible] : [illegible] '.' after %s\n",cp); break;
[illegible] : [illegible] '%s' received\n",tp->token); break;
[illegible] : [illegible] definition may not begin with ");
          printf([illegible]\n",tp->token); break;
[illegible] : [illegible] '' after %s\n",cp); break;
[illegible] : [illegible] duplicate label name '%s'\n",tp->token); break;
[illegible] : [illegible] '%s' received\n",tp->token); break;
[illegible] : [illegible] before '%s'\n",tp->token); break;
[illegible] : [illegible] received\n",tp->token); break;
[illegible] : [illegible] operator '%s'\n",tp->token); break;
[illegible] : [illegible] not permitted in the linkage ",tp->token);
            switch [illegible] {
              [illegible] : printf("preceding "); break;
              [illegible] : printf("inside "); break;
            }
          printf("[illegible]\n"); break;
[illegible] : [illegible] keyword '%s' received %s\n",string,tp->token);
          break;
[illegible] : printf("[illegible] may not be defined empty (i.e. { })\n",string); break;
[illegible] : printf([illegible] operator or literal needed '%s' ",tp->token);
```

```
                printf("received\n"); break;
    case 36 : printf("'%s' is an invalid function or operator name",tp->token);
              printf("\n"); break;
    case 37 : printf("%s is already defined\n",tp->token); break;
    case 38 : printf("%s may not be defined as both a pointer and ",tp->token);
              printf("a function\n"); break;
    case 39 : printf("initialization is not permitted during structure ");
              printf("definition\n"); break;
    case 40 : printf("literals must be initialized\n"); break;
    case 41 : printf("%s is already defined\n",tp->token); break;
    case 42 : printf("%s is not defined\n",tp->token); break;
    case 43 : printf("%s is not a member of [illegible]\n",tp->token,string);
              break;
    case 44 : printf("%s is already used in a [illegible] ",tp->token);
              printf("[illegible]\n"); break;
    case 45 : printf("[illegible] was not declared in its proper [illegible]\n",string);
              printf("[illegible] file [illegible]\n",position,unix_file_name);break;
    case 51 : printf("the logical name has not been declared\n"); break;
    case 52 : printf("the logical name contains too many pointer ");
              printf("references\n"); break;
    case 53 : printf("the logical name is a pointer and therefore cannot ");
              printf("[illegible]\n"); break;
    case 54 : printf("the logical name contains too many array references\n");
              break;
    case 55 : printf("[illegible] for the logical name's members is ");
              printf("[illegible]\n"); break;
    case 56 : printf("the logical name contains an illegal member\n"); break;
    case 57 : printf("one of the logical name's members has too many ");
              printf("pointer references\n"); break;
    case 58 : printf("the logical name's member is a pointer and therefore ");
              printf("cannot contain members\n"); break;
    case 59 : printf("one of the logical name's members has too many array ");
              printf("references\n"); break;
    }
}

TPTR restart(tp,rstrt,dptr,string)
int rstrt;
TPTR tp;
FILE *dptr;
char *string;
{ int j;
  TPTR get_token(), [illegible]_token(), end_link();

  switch(rstrt) {
      [illegible]    : [illegible];
      [illegible]    : [illegible];
      [illegible]    : if (strcmp(tp->token,".") && tp->prior->type != LTERM)
                         [illegible]_token(tp);
                       [illegible]line;
                       [illegible]line;
                       [illegible]strcmp(tp->prior->prior->token,".") ? j:tp->line)
                       [illegible](tp,dptr,FALSE,TRUE);
                       [illegible]_token(tp);
                       [illegible];
      [illegible]    : end_link(tp,dptr,string);
                       [illegible];
      [illegible]    : [illegible];
      [illegible]    : [illegible]_token(tp);
                       [illegible](get_token(tp,dptr,TRUE,TRUE))->token)
                       [illegible] != EOF);
                       [illegible] EOF)
                       printf("[illegible] of current Unix file skipped\n");
                       [illegible];
```

```
              }
              tp = unget_token(tp);
              break;

   return(tp);
}

TPTR [illegible](tp,dptr,string)
TP[illegible]
FILE [illegible]tr;
{ [illegible] get_tok[illegible](), un[illegible]t_token();
   int [illegible];

   tp = [illegible](tp);
   while ([illegible](tp,dptr,FALSE,TRUE)))
      { [illegible]
         [illegible]         : if (strcmp(string,"4"))
                                 continue;
         [illegible]         : if (!strcmp(string,"6"))
                                 { tp = unget_token(tp);
                                   break;
                                 }
                              if (strcmp(string,"1"))
                                 { if ((strcmp((tp = get_token(tp,dptr,FALSE,
                                          OPTIONAL))->token,"::") || strcmp(string,
                                          "4") && strcmp("2",string)) && (strcmp(tp->
                                          token,":") || strcmp(string,"3") &&
                                          strcmp(string,"5")&&strcmp("6",string)))
                                       { tp = unget_token(tp);
                                         continue;
                                       }
                                 }
                              else
                                 { j = tp->line;
                                   k = tp->prior->line;
                                   l = (tp=get_token(tp,dptr,FALSE,TRUE))->line;
                                   if (j ==k || j == l)
                                      { tp = unget_token(tp);
                                        continue;
                                      }
                                 }
                              tp = unget_token(tp);
                              tp = unget_token(tp);
                              break;
         case PUNCTUATION : if ((!strcmp(tp->token,"\"")||!strcmp(tp->token,"'")
                                    ||!strcmp(tp->token,"@")||!strcmp(tp->token,"#")
                                    ||!strcmp(tp->token,"("))&&!strcmp(string,"6"));
                              else if (strcmp(tp->token,".") && (strcmp(tp->token,
                                    "(")||(strcmp(string,"5")&&strcmp(string,"6"))))
                                 continue;
                              tp = unget_token(tp);
                              break;
         case KEYWORD      : if (is_file(tp->token))
                                 { tp = unget_token(tp);
                                   strcpy(tp->token,".");
                                   tp = unget_token(tp);
                                 }
                              else if (!strcmp(string,"6") && (!strcmp(tp->token,
                                    "new")||!strcmp(tp->token,"null")||!strcmp(tp->
                                    token,"return")))
                                 tp = unget_token(tp);
                              else if (!((!strcmp(tp->token,"copy") ||
                                    !strcmp(tp->token,"share"))&&strcmp(string,"5"))
```

```
                              && !(!strcmp(tp->token,"rename") &&
                               strcmp(string,"s")))
                             continue;
                           else
                             tp = get_token(tp);
                           break;
        default          : continue ;
        }
      break;
    }

  return(tp);
}
```



```
bundle rational_sort
structure_definitions
io_struct_def
variables
mul_var
build_tree
.

state structure_definitions
  rename znode treenode
rational :: { int : numerator
              int : denominator
            }
char :: { 'a' : ch
        }
treenode :: { rational : number
              #treenode : #left, #right
            }
list_info :: { treenode : #node
               int : n
             }
.

state io_struct_def
share structure_definitions
io_str_struct :: { int : file_descriptor
                   char : buffer[512]
                   int : n
                 }
file_in_structure :: { char : file_name[20]
                       int : mode
                     }
.

storage variables
share structure_definitions
rational : number, get_number null
znode : #root = null
list_info : info[]
#info'node : array_of_pointers[45]
int : > int, > rational, * int, == int
int : y[]
#y : #z, zz[]
null : main_routine int,
       put_in_list list_info
.

storage mul_var
int : i = 0
.

actor build_tree
share structure_definitions,

main_routine :: in int : count
  ! count  # $ number = get_number
           # $ info'node = root
           # $ info'n = number
           # $ put_in_list info \
```

```
        }

> :: in   int : irtl, int2
     out int : true
   { int1-int2 $ return 1
                 return Ø
   }

put_in_list :: in list_info : info
               rename info'pnode pnode
               rename info'node  node
               rename info'n     n
               block1 : null
               { pnode ? $ new node;
                        pnode'number = n;
                        pnode'left = pnode'right = null
                      } ..
                      { n > pnode'number $ { node = pnode'right
                                               put_in_list info
                                             } ..
                                             node,
                                               = pnode'left
                                             put_in_list info
                      }
               }

> :: in  rational : rat1, rat2
     out int : true
  rename rat1'numerator    n1
  rename rat1'denominator  d1
  rename rat2'numerator    n2
  rename rat2'denominator  d2
  { n1*d2-n2*d1 $ return 1
                  return Ø
  }

* :: in  int : irt1, int2
     out int : product = Ø
  { copy mul_var
    {   int1 == Ø $ ..
        int2 == Ø $ ..
                    product = product + int1;
        int2 > Ø  $ int2 = int2 - 1 \
                    int2 = int2 + 1;
                    sign = 1 \
    }
    sign $ product = Ø - product
  }

== :: in  int : int1, int2
      out int : true
   { int1 - int2 $ return Ø
     int2 - int1 $ return Ø
                   return 1
   }
```

```
typedef struct rational {
                int numerator;
                int denominator;
                } rational;

typedef struct char {
                } char;

typedef struct znode {
                rational number;
                struct znode *left, *right;
                } znode;

typedef struct list_info {
                znode *node;
                int n;
                } list_info;

typedef struct io_structure {
                int file_descriptor;
                char buffer[];
                int n;
                } io_structure;

typedef struct file_io_structure {
                char file_name[];
                int mode;
                } file_io_structure;

rational number, get_number();
znode *root = ;
list_info info[];
znode (*x), *(*array_of_pointers)[];
int >(), >(), *(), ==();
int y[];
int *(z[]), (z[])[];
null main_routine(), put_in_list();

int sign = ;

main_routine(count)
int count
{
  /****************************************************************/
  /*                                                              */
  /*            function or operator body                         */
  /*                                                              */
  /****************************************************************/



int >(int1, int2)
int int1, int2
{
  /****************************************************************/
  /*                                                              */
  /*            function or operator body                         */
  /*                                                              */
  /****************************************************************/



put_in_list(info)
list_info info
```

```
{
  /*****************************************************/
  /*                                                   */
  /*              function or operator body            */
  /*                                                   */
  /*****************************************************/


int >(rat1, rat2)
rational rat1, rat2
{
  /*****************************************************/
  /*                                                   */
  /*              function or operator body            */
  /*                                                   */
  /*****************************************************/


int *(int1, int2)
int int1, int2

  /*****************************************************/
  /*                                                   */
  /*              function or operator body            */
  /*                                                   */
  /*****************************************************/


int ==(int1, int2)
int int1, int2
{
  /*****************************************************/
  /*                                                   */
  /*              function or operator body            */
  /*                                                   */
  /*****************************************************/


main()
{
  /*******************************************************/
  /*                                                     */
  /*                 D program driver code               */
  /*                                                     */
  /*******************************************************/
}
```

```
bundle test
    share defs
  defs
   vars
   actor1
  actor2

state defs
  struct_x :: { int  : number1, number2
                int  : number3[10]
                char : value
              }

  floating_point_number :: { int : number
                             .
                             int : decimal
                           }
.

storage vars
    rename floating_point_number float
    rename y z
    rename x y
  float : a, b, c, do_nothing null
  char  : z
  int   : y
.

actor actor1
  share vars

  do_nothing :: out float :  flt
  { [illegible]
.
```

```
line 8: 's terminating period not detected
li[illegible]2: class expected; 'float' received

typedef struct struct_x {
                int number1, number2,
                int *number3[];
                char value;
                } struct_x;

typedef struct floating_point_number {
                int number;
                int decimal;
                } floating_point_number;

floating_point_number a, b, c, do_nothing();
char y;
int x;

float do_nothing()
{
  /*************************************************************/
  /*                                                           */
  /*             function or operator body                     */
  /*                                                           */
  /*************************************************************/



main()
{
  /***************************************************************/
  /*                                                             */
  /*                D program driver code                        */
  /*                                                             */
  /***************************************************************/
}
```

```
bundle rational_sort
share getnum
copy afile
share [illegible] plus
share
  [illegible]
copy, another report
share, not sure
action
.

static structure_definitions
rational :: ( int : numerator
              ,
              int :: denominator
             ;
char [illegible]
         int [illegible] in some garbage : ch
        ;
treenode :: ( rational : number
             treenode *left, *right
             ;
list_info :: ( treenode : *node
               int       : n
              ;
sort :: 5 .. 9
.
```

```
line 4: linkage error generated when 'plus' received
line 7: line terminator required before 'another'
line [illegible]7: linkage error generated when 'report' received
line [illegible]: line terminator required before 'not'
[illegible] linkage error generated when 'sure' received
line [illegible]: file structure definition was not declared in its preceeding
        [illegible] file '[illegible]_part'
[illegible] of [illegible] this file skipped
```

Bibliography

1. Booch, Grady. Software Engineering with Ada. Menlo Park, CA: The Benjamin/Cummings Publishing Company, May 1983.

2. Brown, P. J. "Error Messages: The Neglected Area of the Man/Machine Interface?," Communications of the ACM, Vol 26 No 4: pg 246-249 (April I1983).

3. Earley, Jay. "An Efficient Context-Free Parsing Algorithm," Communications of the ACM, Vol 26 No 1: pg 57-61 (January 1983).

4. Graham, Susan L. and Steven P. Rhodes. "Pratical Syntactic Error Recovery," Communications of the ACM, Vol 18 No 11: pg 639-649 (November 1975).

5. Jennings, Capt Richard K. A Candidate Programming Language, MS Thesis GA/EE/83D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1983.

6. Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1978.

7. Peters, Lawrence J. Software Design: Method & Techniques. New York, NY: Yourdon Press, 1981.

8. Sippu, Seppo and Eljas Soisalon-Soininen. "A Syntax-Error-Handling Technique and its Experimental Analysis," ACM Transactions on Programming Languages, Vol 5 No 4: pg 656-679 (October 1983).

9. Waite, Mitchell et al. Unix Primer Plus. Indianapolis, IN: Howard W. Sams & Co., Inc. 1983.

10. Wirth, Nicklaus. Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1976.

VITA

Lieutenant Kevin J. Shomper was born on 4 December 1961 in Harrisburg, Pennsylvania. He graduated from High School in Littleton, Colorado in 1979 where he subsequently entered the University of Northern Colorado. He received a three and one-half year ROTC scholarship, and graduated in June 1983 with a B.A. in Mathematics. Three days following graduation and commissioning Lt Shomper was adjusting to his new duties as a student at the Air Force Institute of Technology.

Permanent address: 6452 W Alder Ave
Littleton, CO 80123

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. 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/84D-27	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) See box 19	

12. PERSONAL AUTHOR(S)
Kevin J. Shomper, B.A., 2Lt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1984 December	166

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Programming Languages, Interpreters, Translators Error Recovery, Parsing,
9	2		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title : A D TO C INTERPRETER

Thesis Chairman: Harold W. Carter, Lt. Col., USAF

Approved for public release: IAW AFR 190-17.
LYNN ...
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB ...

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Harold W. Carter, Lt. Col., USAF	513-255-6913	AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

In a continuing effort to define and analyze the D language a partial interpreter has been built. This interpreter stresses both syntactic and semantic error reporting in order to function as a learning aid to the inexperianced D programmer. The language definition is completed, with the exception of the accept function and the ? block control. All error checking has been accomplished, except expression type checking.

D contains weaknesses; most notable in its class construction. Although, this is a perceived problem and not a functional one. It is reccommended that the reader be familiar with both Ada and C in order to full grasp the ideas. Follow on work should complete the code generation, but in Ada not in C. Only then will the full potential of D be realized.

END

FILMED

10-85

DTIC