A MODULE DECLARATION GENERATOR

by

JOSEPH LIBRERS

B. S., San Jose` State University, 1976

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

[illegible]

Major Professor

Master's Report on
A Module Declaration Generator

LIST OF FIGURES

Acknowledgment

To my wife, Elizabeth for her patience, understanding, support, and love while I completed the requirements for this Degree.

To my mother and father for instilling in me the desire to learn.

To Mick and Mary Beth Cole for their friendship and efforts which made my five summers in Kansas enjoyable and productive.

To Dr. and Mrs. David A. Gustafson for their guidance and kindness.

To Ms. Clare E. Wherley who gave me the opportunity to attend Kansas State Univerisity.

Chapter 1 - Overview

1.1 Introduction

This project developed a Module Declaration Generator (MDG) from an Entity Relationship Attribute (ERA) Requirements Specification. This project is one of many software tools designed and developed as part of a research project conducted at Kansas State University's Computer Science Department during the summer of 1984. The overall project is intended to address the software development technology of the future.

1.1.1 The Software Crisis

During the 1960's the manpower costs of programming new information systems increased so rapidly the term "Software Crisis" was coined. Despite massive research and development efforts to advance the field of software engineering technology the trend continued into the 1970's. In the United States the software industry still suffers from insufficient reliability and poor maintainability of software products, frequent overruns in development costs, and high rates of personnel turnover <Kim83>. The United States Department of Defense (DoD) initiated the STARS program; the STARS program goal is to improve productivity while achieving greater system reliability and adaptability <Mar83>, <Dru83>.

Japan, having achieved phenomenal technological advances through the 60's and 70's, is trying to extend its success via a national research and development project, the Fifth-generation Computer System project <Tre82>. Their project is intended to represent a unification of four currently separate areas of research, namely knowledge-based expert systems, very-high-level programming languages, decentralized computing, and very-large-scale integration (VLSI) technology <Tre82>. One of the key technologies being given considerable amount of attention is software engineering. The Japanese Government intends to push Japan into the forefront of computer system technology by 1990 <Kim83>.

1.1.2 Automating Programmer Activities

One of the primary thrusts of both the United States and the Japanese efforts is to reduce the cost of producing software through automation <Dou83>. One of the objectives of the DoD STARS program is an automated support environment; this environment comprises tools to provide automatic support of the tasks occurring during the entire system life cycle, providing generic automated tools and automated tools oriented towards specific management practices, methodologies or applications <Dru83>. In Japan, much research and development is also being done in this area. The Yokosuka Electric Communication Laboratory (ELC) is one of many organizations developing a variety of new software tools. One tool ELC is developing is PAL (program automation language), a system design language that automatically generates an executable program from a description of its data structure specifications <Kim83>.

Though radical changes in the way computers work will be essential to break the programming bottle neck, a number of years will be required before this approach bears fruit. The principles of computer-system design for the past three decades has remained largely static, being based upon the von Neumann computer, a sequential control-flow computer. This computer contains a low-level machine language in which the instructions perform simple operations on simple operands; the memory is organized in linear fixed-sized cells. The von Neumann computer incorporates a processor, communications and memory with sequential, centralized control of computations and primitive input/output capability. The Japanese believe this concept has contributed significantly to the "software crisis". Their fifth-generation computers are expected to be available in the 1990's; these computer systems will provide three basic functions: the intelligent interface, knowledge-based management, and problem-solving and interface functions. the intelligent interface will support conversations with a computer and can be described as being analogous to traditional input/output channels and devices; the conversation will be in the form of speech, graphics and natural languages. The knowledge-based management function is to be capable of retrieving information needed for a problem solution not only from mere files of uniform content but also from files containing collections of facts, inferences and procedures <Kan83>, <Tre82>. This function is to be accomplished through an integration of main memory, virtual memory facilities and a file system. The problem-solving and inference function is regarded as equivalent to the traditional computers' central processing unit (CPU) but much faster <Tre82>.

Present efforts are aimed at developing automatic program-generator tools using very-high-level programming languages capable of generating entire high-level language algorithms from a simple statement about the purpose or behavior of a program. Some of these tools use relationships between the data groups to determine the sequence in which procedures are to be performed; these tools depend upon data-flow languages. Data flow languages are collectively termed "nonprocedural"; by allowing the user to merely state what is to be performed rather than as with conventional languages how to perform the program, they provide improved programming methodologies <Ler82>, <Tre82>.

In data-flow programming the compiler program uses the data group relationships to determine the sequence of procedures by analyzing the statements entered; the availability of the input data triggers the execution of the task or operation to be performed <Ler82>. Functional languages such as Pure Lisp and predicate logical languages such as Prolog are two of the best developed classes of very-high-level languages; these languages are well suited for programming knowledge-based expert systems <Tre82>. Data flow programming and very-high-level programming languages are the basis for many new automation program-generation systems.

There are some computer scientists that question whether the problems raised by the software explosion can be solved by automatic program generation with existing computer hardware and object languages; the translation from a specification into a high-level language into machine language code may prove to be too slow and cumbersome. Redesigning computers and languages may be more effective than merely automating existing practices <Ler82>.

1.1.3 Automatic Program Generators

Automatic program generators are taking the mystery and drudgery out of the art of programming computers. The process of automatically generating a program given a formal specification of its variable input and output groupings will probably require much more basic research <Dou83>. A limited form of automatic programming is presently possible but these automated tools are still surrounded by other manual activities; these manual activities constitute bottlenecks by consuming manpower and producing errors. The Model language developed by the automatic program generation project of the Moore School of Electrical Engineering at the University of Pennsylvania is an example <Ler82>. This automatic programming generator creates the program through a series of separate steps. After accepting statements from a user a compiler analyzes and categorizes each statement identifying variables and conditions. A graphical representation of the dependencies of each variable upon the others. The graph shows which variables must be calculated first and which variables are subdependents. The compiler searches for ambiguities in the variable names and incompleteness and inconsistencies in variable definitions. The user will resolve such errors. The compiler creates a flow chart of the program once an error free specification is prepared. The flow chart will detail the sequence of program steps; the compiler attempts to optimize the efficiency of the program by consolidating iterations. Each block of the flow chart is translated into high-level language code <Ler82>. Although automatic programming does relieve the drudgery of programming manual user interfaces are still required.

The purpose of many software tools is to minimize the time and effort required to turn the requirements into executable code. Some of the manufacturers of automatic program generators claim the generators improve productivity of trained programmers by more than ten times, although their users have a more conservative estimates. Program maintenance is much easier because the program structure is consistent from application to application. Much of the disagreement stems from the notion programmers only code; very little of the programmers time is spent in actual coding but in communication with others on the project, system design and testing <Gla83>.

1.1.4 Common Characteristics

Although module generators differ widely in what languages and applications they are used some common characteristics exist. Module generators require the user to list inputs, outputs and local variables. They accept the user request in a formal framework. Unique and consistent definitions of the variables is required. Since some module generators automate existing practices, they use existing hardware.

Some generators accept the user requests through preset frameworks. APG (a program generator) interrogates a user who answers a series of questions in English <Hag75>. An application generator, TAGS, requires a user to describe an application by answering prompts by picking items from menus and filling in blanks <Gla83>. Model, discussed above, accepts sets of equations, matrix and vector expressions and operations such as matrix multiplication <Ler82>. Currently, most module generators accept user requests through a formal preset framework.

Module input/output is handled by requiring a user list inputs, outputs and local variable names. Following the data flow architecture concepts, the module generators use the data group relationships to determine the sequence of procedures; the availability of the input data is required before the task or operation can be performed <Tre82>.

Unique and consistent definition of the data elements is required. As explained above in the Model language discussion, the source specification must be error free; ambiguities, different data assigned the same name, must not exist. All variables must have complete and consistent definitions.

Some generators automatically translate the user input into programming code needed to perform the application. The Model system discussed above generates either PL/1 or COBOL. Another module generator generates FORTRAN programs <Cro83>. MAPSE (minimal Ada programming support environment) a programming support environment within the STARS program, is being expanded to automate the Ada programming process <Dru83>.

1.1.5 The Module Declaration Generator

In this project the user will enter the module requirements through an ERA specification. The ERA will support the formal specification requirement. The ERA as used here is a formal specification consisting of a set of frames (see ERA BNF). A group of frames will compose an activity frame. Each activity frame will be entered singularly.

The entities from the ERA will be matched against a data dictionary file which will contain all entities required by the application. Relationship frames designated as 'input', 'output' or 'uses' will serve as the vehicle for user input. Each entity name will precisely match an entity name being held in the data dictionary (see Appendix - Data Dictionary BNF). The entity interrelationships are described within the data dictionary.

The entities will be mapped through a data dictionary to produce the data declarations and definitions required for the module. The entity definition held within the data dictionary will determine the data declaration or data definition required during the transformation into the modules' declaration code.

A file containing the framework for an executable PASCAL program will be generated. This file will be a UNIX* text file and will contain a program header, comments describing the activity, comments containing the user input entity descriptions, variable declarations, constant definitions, data type definitions and the PASCAL 'begin' and 'end.' statements.

* UNIX is a Trademark of AT&T Bell Laboratories

Chapter 2 - Project Requirements

2.1 Introduction

The module declaration generator should minimize the programming effort required to produce software. The amount of manual effort should be reduced by automating some of the manual functions normally required to produce a module. As with some software tools the purpose of the module declaration generator should be to minimize the time and effort required to turn the requirements into executable code.

The design of small, singular-function modules should be encouraged. A small and singular-function module is less cumbersome to test and debug than multi-function bulky modules. The manual effort for maintenance should be reduced with a large assemblage of small modules rather than one large complex module.

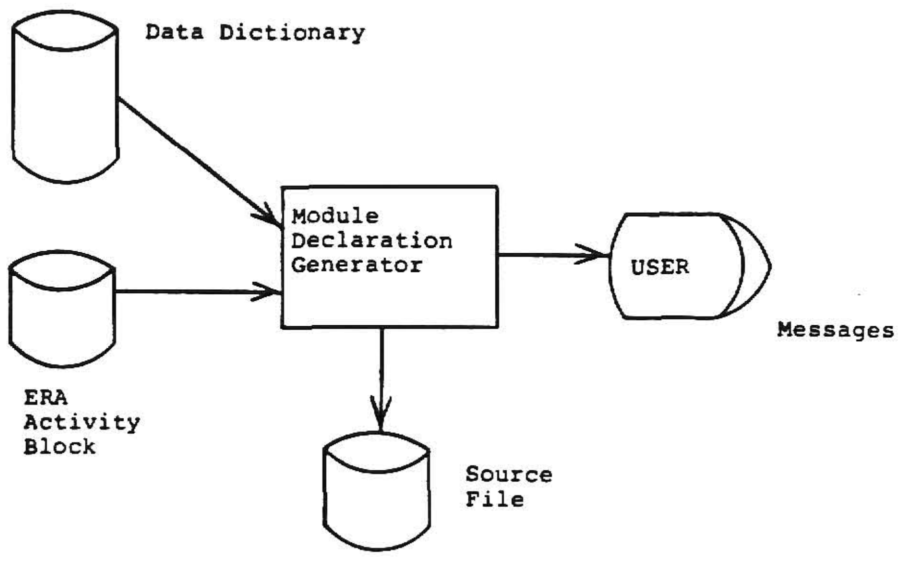


Figure 1. Project Overview

2.1.1 Requirements of Input

A standard user input format should be employed. A user is better able to communicate the module requirements if the input format is consistent rather than in a state of flux. As discussed above one of the bottlenecks in software development is the manual interfaces; care must be taken to keep this interface standard and simple but formal.

A centralized repository of data definitions and declarations should be employed. Using a centralized repository should reduce the ambiguity of entity names and ensure completeness and consistency in the entity interrelationships. The repository should contain a list of entries; each entry should contain a complete specification of the entity including its symbol name as it appears in the requirements specification, a description, a data specification and any informative information such as a constant value if the entity is a constant.

The user should only have to specify the input/output requirements once. By specifying these requirements only once manual effort is reduced and chance for error from the manual interface is also reduced. The consistency of the requirements will also increase.

2.1.2 Requirements on Output

The name of the module generated should be similar to the name used in the requirements specification. This will add continuity for the whole application during the design, testing, implementation and maintenance operations. The use of ambiguous or vague names could be reduced. The effort required to locate a malfunction or a module requiring a change should be reduced.

The format of the module framework should be standardized. Standardization tends to increase communication among programmers. The output module framework code should be uniform, correct, complete and compilable. When produced manually code too often contains time-consuming errors. When code is not uniform, programmers are reluctant to increase the number of modules in a program and instead increase the size of an existing module <Cro83>.

The entity names generated by the module declaration generator should be similar to the symbol name as it appears in the repository and the requirements specification. This will add continuity to the application's operation.

Consistent data structures for the entities should be generated. The interrelationships of the data must be preserved. For interfacing modules to communicate data with minimum errors the data structure must be consistent.

2.1.3 Characteristics

The module declaration generator itself should be maintainable and flexible. The MDG program should be composed of small, singular-function modules; simple algorithms and a top-down structure should be required. The MDG should use standard and generic functions and subroutines when possible. Comments should be included for all global variables and to explain subroutines. Variable names should be meaningful and if used for a similar task have consistently structured names.

The MDG should be portable to other UNIX operating system environments. The portability is a function of the language in which the MDG is written. C language should be used. C is powerful, concise and a machine-independent language. Commands are simple and structured programming methodology is supported with looping (for, while statements), with case selection (switch statements) and with decision-making control (if then else) statements. In a standard UNIX environment a C program can be recompiled and executed under an other operating system (CP/M-86) <Rif83>.

The intent of the module declaration generator should be to minimize the effort to transform the requirements specification into a useful module framework; at the same time the MDG itself should be easy to use, flexible, maintainable, reliable and relatively portable.

Chapter 3 - Project Design

3.1 Introduction

The module declaration generator project`s objective is to develop a method of accepting standardized user information and minimize the effort required to transform this information into a framework for a PASCAL program. This module declaration generator project will have two input files and produce two output files. This module declaration generator project will use as input an ERA Requirements Specification Activity Block and a data dictionary file, an entity data repository. User input and output requirements will be accepted through the Activity Block from the ERA. The entity interrelationships will be supplied through the data dictionary file. The output will consist of a UNIX text file containing the framework for a PASCAL program and user messages to be displayed on the user`s display device. The output framework will be a compilable PASCAL program.

3.2 The Module Declaration Generator Input

The module declaration generator will have two input files, the ERA Requirements Specification file and the data dictionary file. The use of the ERA Requirements Specification will accomplish many of the project requirements. All user input will be standardized with a formal input framework; the input/output requirements need only be specified once. The ERA Activity Block Header, "Activity : activity-name ", will supply the name to be used as the name of the generated module. The input specification line(s), "input : symbol-name", will supply the symbol name(s) of the input variable(s); the output specification line(s), "output : symbol-name", will supply the symbol name(s) of the output variable(s). The symbol names entered through the ERA Activity Block will be individually mapped through the data dictionary file.

The data dictionary is the central repository for data about the entities contained on the ERA Specification. This data will include information necessary to understand the relationship of the entity and its use within the organization. The symbol names of the data dictionary entities will be the same as supplied from the ERA Requirements Specification. The entity interrelationships will be described through a hierarchy of symbol declarations, which are the keywords of the data dictionary. The sequence of these declarations is important to the proper generation of the data type declarations for the module being generated. As show in the hierarchy figures (see I/O Mapping and Type Mapping), the data dictionary keywords "PASCAL", "USAGE", "TYPE", "RANGE", "VALUES", and "COMPOSITION" play an important roll in determining the proper variable data declaration construct and ensuring the compilability of the output framework program. Each entity is to be searched for individually in the data dictionary.

When the keyword "PASCAL" is associated with a entity the data definition is to be used as supplied. This feature will enable the MDG to generate PASCAL declarations more complex then the intended scope of the project at hand.

The keyword "USAGE" is to signify the intended usage of the entity. This keyword will indicate the entity could be used as an array, string or constant. The usage keyword will assist the interpretation of the entity.

Where as, the "TYPE" keyword will describe the data type of the entity. The MDG will be able to discover whether the entity is to be used as a type character, real, integer, or enumerator.

And the keyword "RANGE" will signify the actual range of an array. The actual starting and ending values of the array will be noted in this keyword. If the entity is a double array, both sets of values will be described. The MDG will be supplied with the information required to construct the array with know values.

The data dictionary keyword "VALUES" will describe the actual value of a constant entity. As the MDG forms the PASCAL module declarations the defined constant value is to be established within the PASCAL format.

The "COMPOSITION" keyword will declare which other entities compose the entity being declared. This keyword will enable the MDG to further define those entities. The MDG should be able to search the data dictionary file for each of these entities and locate and format their declarations.

Using these data dictionary keywords, the MDG should be able to precisely define the entities as entered from the ERA Specification in a manner consistent with other users of the data dictionary..

3.3 The Module Declaration Generator Output

Two outputs will be produced from the module declaration generator: a UNIX text file and screen messages returned to the user`s display device. The UNIX text file generated is to contain the framework of an executable PASCAL program. This file is to contain (in this sequence) a PASCAL program header, the ERA Activity Block in comments, the entity descriptions as they were located in the data dictionary, all PASCAL constant ("CONST") definitions, all PASCAL data type ("TYPE") definitions, all the PASCAL variable ("VAR") declarations, the PASCAL "begin" program statement, the PASCAL program body area, and the PASCAL "end." program statement.

The variable names in the generated framework are to be similar to the symbol names found in the ERA Specification Activity Block (and within the data dictionary). This naming convention will aid in debugging, testing and maintenance; a consistent flow of data can then be mapped through the program generated with this module declaration generator.

The user`s messages should display the ERA Requirements Specification Activity Block as entered and messages appropriate for the proper operation of the module declaration generator. These messages should indicate the name of the UNIX text file containing the PASCAL program generated and error messages associated with the non-performance of or completion messages of the MDG module.

3.4 The Module Declaration Generator Program Design

The module declaration generator is to be written in C Language on a UNIX operating system. A Top-down structure and modular functions are to be employed. Algorithms should be designed to be simple and easily understood. The subroutines are to be documented with useful comments to enable an understanding of the routine being performed. All variables used in the MDG program are to be described with useful comments. The names of these variables are to be meaningful; variables with similar uses are to have unique but similarly constructed names. Generic functions are to be employed and must remain consistent throughout the program. Standard input/output routines are to be utilized wherever and whenever appropriate; file and character string manipulation functions should be similar.

Chapter 4 - The Implementation

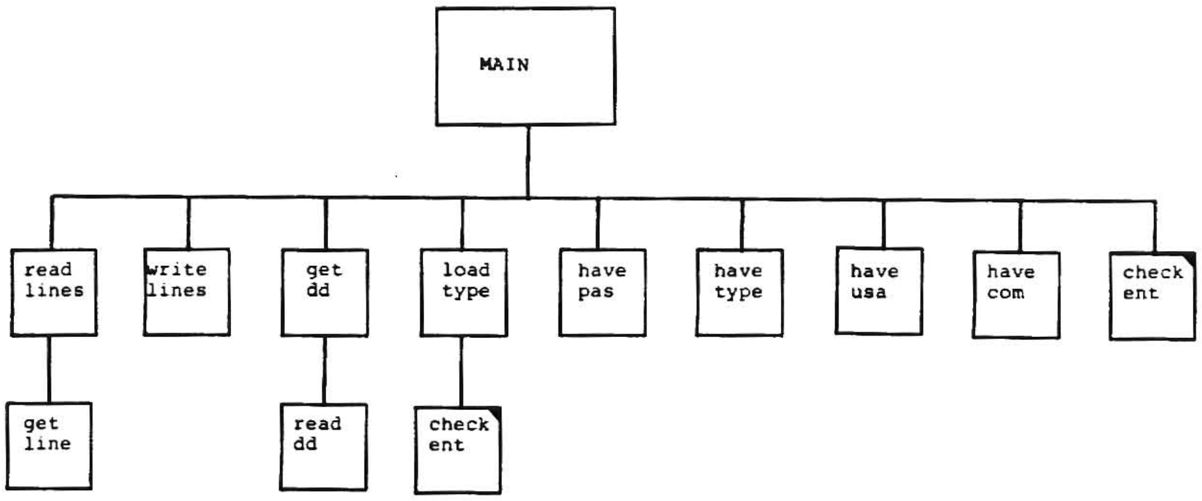
4.1 General Overview

The implementation of the MDG project at Kansas State University occurred during the 1984 summer semester. This MDG project was a part of an overall software tool prototype project intended to automate software development tasks. This Module Declaration Generator is responsible for the generation of a compilable framework for a PASCAL program from a standardized input specification, the ERA Requirements Specification.

The MDG project was developed on a PLEXUS minicomputer running a UNIX operating system. A demonstration of the MDG project capabilities and functions was displayed upon the Kansas State University Perkin-Elmer 8/32 minicomputer also running a UNIX operating system.

The MDG project consists of one C Language module and one shell in which to execute the program. The program is approximately 950 lines of code and contains 11 subroutines (see Figure 2). A Top-down structured design and implementation was utilized to develop this program. Small singular-function routines were developed. The names of the subroutines and variables were chosen to be meaningful to their function or use; the names of similar subroutines and variables are similarly named but have unique names.

Figure 2. Hierarchy Diagram



This project can be used as a Software tool to generate the framework for a compilable PASCAL program.

4.2 MDG Module Input

The MDG module expects an ERA Requirements Specification Activity Block and a Data Dictionary file as input. The ERA Activity Block will specify the input and output entities for the activity and the data dictionary will serve as a data base for information about each entity.

The ERA Activity Block must in the format described in the ERA Requirements Specification BNF (see Appendix - ERA BNF). The use of a standard specification will minimize development effort by requiring the user to specify the input and output entities only once for each PASCAL program to be generated.

The Data Dictionary supplies the information about the entities entered on the ERA Activity Block (see Appendix Data Dictionary Example). Each input and output entity which can be included on an ERA Activity Block will be described in detail within the Data Dictionary file. The format of the Data Dictionary file is described in the Appendix Data Dictionary BNF.

Each entity is individually located within the Data Dictionary file. If an entity is composed of other entities, these additional entities are in turn located within the dictionary. Each entity is to be declared only once with in the PASCAL framework; a check to prevent duplicate declarations was included.

4.3 MDG Module Output

The MDG module will generate user messages and when correctly executed, generate a UNIX file. These messages will advise the user about the operation of the MDG and the file name of the UNIX text file containing the PASCAL framework. The file name will be accepted through the execution statements or the execution shell (see Appendix - Run Instructions).

The UNIX text file will contain the framework of an executable PASCAL program. The file contains (in this sequence) a PASCAL program header statement, the ERA Activity Block in comments, the entity descriptions as they were located in the data dictionary, all PASCAL constant ("CONST") definitions, all PASCAL data type ("TYPE") definitions, all the PASCAL variable ("VAR") declarations, the PASCAL "begin" program statement, the PASCAL program body area, and the PASCAL "end." program statement. The ERA Activity Block will be included as comments to the framework as are descriptions of the entities located in the Data Dictionary. The framework generated upon the successful completion of this MDG will be compilable. This file can then be detailed with the PASCAL logic to perform the activity required.

User messages will be generated describing the success or failure of the MDG. These messages will keep the user informed as to the operation of the MDG. During the testing of this MDG, these messages were generated instantly. This implies the performance of the MDG is very adequate.

4.4 MDG Operation

After reading the ERA Activity Block, using subroutine "readlines", the MDG will begin to generate the UNIX file. The Activity Block name will be used as the PASCAL procedure name. The ERA Activity Block will be included as comments within the framework generated; this will assist in the documentation of the individual modules. The entity names will be stored in an internal table to be used later for matching the Data Dictionary.

Messages will be relayed to the user. The user will receive messages describing the Activity Block entered; in this way the user can verify the correct input ERA Activity Block was entered, without waiting to review the output framework.

Each entity entered will be located in the data dictionary. The Data Dictionary will be read from the beginning for each entity using subroutines "getdd" and "readdd". This will allow the entities to be entered in any sequence or be composed of any other entity. When there is a match in the "NAME" keyword, the MDG will continue to gather related information about the entity until another "NAME" keyword is discovered. The related information is obtained form other Data Dictionary keywords to be used in the IO Mapping and Type Mapping processes. If an entity is composed of other entities, these entities are added to an entity table using subroutine "loadtype",and will in turn be located within the Data Dictionary.

As the entities are matched to the Data Dictionary, the subroutine "writelines" generates the lines of the framework. After all entities in the entity table have been evaluated and their PASCAL declarations composed and generated, the PASCAL "begin" and "end." statements are generated.

When the MDG's task is completed a message indicating the name of the UNIX text file generated is issued to the user. This will allow the user to immediately enter the framework and begin to compose the PASCAL logic required to perform the specified task.

The user will be sent messages indicating the successful completion of the MDG. Messages indicating error conditions are also generated, if required. If the input is unexpected or data is not matched to the Data Dictionary, meaningful error messages will be returned to the user indicating the problem preventing the successful completion of this MDG (see Appendix - User Messages).

4.5 IO Mapping and Type Mapping

The IO Mapping and Type Mapping functions ("havepas", "havetype", "haveusa", and "havecom") will enable the MDG to compose the PASCAL Language declarations from the Data Dictionary file data. Each entity submitted from the Activity Block must be declared within the PASCAL framework. The IO Mapping and Type Mapping will determine the type of entity.

The IO Mapping function will first locate the entity using the keyword "NAME". The "DESCRIPTION" keyword is next to be located; this description is used in the framework as informational comments. If the "PASCAL" keyword exists, the subroutine "havepas" will use the information found to format the PASCAL type declaration and write to the framework file. The next entity is ,in turn, located in the Data Dictionary. If this "PASCAL" keyword is not present than other keywords will be located.

The MDG will attempt to locate the next keyword "USAGE". If located, the MDG will determine if the entity is a constant or an array using subroutine "haveusa". If the entity is a constant a "VALUE" keyword will be located; this keyword will indicate the actual value of the constant. If the entity is an array a "RANGE" keyword will be located which will indicate the range of the array. At this point in the process an additional situation is present. If the range of the entity is another entity, the newly found entity will be stacked on the entity table for Type Mapping and the declaration is formatted and written to the framework file. If the range is a data type (real, integer, etc) the declaration is formatted and written to the framework file.

If the keyword "USAGE" was not located, a search for the keyword "TYPE" will be initiated. If found, the "TYPE" keyword will inform the MDG of the entity's data type. Subroutine "havetyp" will process this information. The type could be another entity in which case the newly located entity will be entered in the entity table for Type Mapping. If not another entity, the entity could be of real, integer or another defined type. In either case the declaration will be formatted and written to the framework file.

If none of the above keywords have been located, a search for the "COMPOSITION" keyword will be initiated. This keyword will be processed in the subroutine "havecom", and will indicate the entity is a record and contains other entities. These new entities will be added to the entity table for Type Mapping and the declaration will be written to the framework file.

Type Mapping is very similar to the IO Mapping. Other than the "DESCRIPTION" keyword process, all other keyword searches and processes are required; the same subroutines will be executed as in the IO Mapping. This was decomposed into a separate function to enable recursive routines to added at a later time if required.

4.6 Limitations

The MDG program can be easily expanded to operate under a different set of specifications. This project, being a part of a Software Tool Prototype, is flexible enough to allow for modifications to the input files if required; Only modifications to the MDG source code "define" statements should be required. These established limitations allowed the project to be shown feasible and the development to be accomplished within a reasonable time schedule.

The definitions of the MDG input and output specifications can be adjusted to accommodate format or size modifications. Presently, the maximum length of an entity name, including the $ sign, is established at twenty-five (25) characters. Up to twenty (20) lines of thirty-five character text of the ERA Activity Block will be read. One hundred (100) lines of sixty (60) character text will be read from the Data Dictionary. The maximum length of a Data Dictionary keyword was established at thirteen (13) characters. The output PASCAL module name will be no longer than twenty (20) characters and each line of PASCAL code will be no more than 80 characters.

By changing the MDG program "define" statements found in the beginning of the source code, (see Appendix Program Source Code), the MDG can accommodate format and file size modifications.

4.7 Testing

The testing of the MDG project consisted of running numerous ERA Activity Blocks of various types through the MDG program. The PASCAL compiler was applied to the generated framework to verify the correctness of the declarations generated. No cases of this failure have been discovered.

Samples of the PASCAL program framework output generated using this MDG as well as samples of the ERA Activity Block and the Data Dictionary used as input are included in the Appendix of this report.

4.8 Results

The MDG generates a compilable framework for a PASCAL program. With the addition of the PASCAL logic code, a workable program can be developed to accomplish the goals of the Activity Block. The MDG shows flexibility by allowing modifications to the input and output limitations to be easily applied to the "define" statements as required. As a software tool this MDG is useful to the development of PASCAL modules by automating the manual effort required to generate the PASCAL module.

Chapter 5 - Conclusions and Extensions

5.1 Conclusions

All requirements of this project were satisfied. The Module Declaration Generator was developed; the use of which will reduce the manual effort to develop PASCAL modules by requiring the user to input the specification only once. The MDG is written in C Language and runs on a UNIX operating system.

The MDG reads the ERA Activity Block and the Data Dictionary. These standardized input files contain the information about the entities required to complete the declarations for a PASCAL program. Keywords are utilized to perform the declaration function.

The PASCAL program framework is generated. This framework is compilable. A programmer can complete this module by adding the logic with which the module can perform its specified task.

Messages related to the operation of the MDG are sent to the user. The user is advised of the success or failure of the MDG. The ERA Activity Block and the output file containing the PASCAL framework are displayed to the user.

5.2 Advantages

The use of this MDG can reduce the effort required to produce PASCAL software. The amount of manual effort required is reduced by automating the declaration functions normally required to produce a module. The use of a Data Dictionary file will reduce data specification errors, since all entities will be resident in the file.

The design of small, singular-function modules is encouraged. Each ERA Activity Block will represent a small, singular-function module. The MDG will present this information to the user.

The maintenance of modules produced using this MDG should be reduced. Since all modules will use the same names for variables and the names of these variables are the same as referenced with in the specification, the programmer can easily determine which modules will required modifications.

5.3 Disadvantages

There are two disadvantages to using the MDG. All entities must be resident within the Data Dictionary and must be similarly referenced by the ERA Requirements Specification. This is a large clerical task. All known entities must be described with adequate detail for this information to be useful to the MDG. Every interrelationship for the entities must also be established.

The software functions must be specified using the ERA Requirements Specification. This is no small task either. Each process must be decomposed into small, singular-function activities.

5.4 Extensions

This MDG produces the framework for a PASCAL program. The capability to produce framework for other languages can be added. By expanding the IO Mapping and Type Mapping functions, the declarations can be generated for the other languages.

The MDG has the capability to construct more complex structures. The IO Mapping and Type functions were intentionally limited to constrain the scope of this project. To process more complex structures, these mapping function could be executed recursively, if required.

This MDG accomplishes the first step in producing the PASCAL module.

The programmer must supply the logic to produce a functional module. A study of PASCAL logic generation should be initiated to aid the programmers logic code detailing effort.

The restrictions placed upon the MDG by the Data Dictionary and the ERA Requirements Specification could reduced but not entirely eliminated. By expanding the specification to include constant values for entities uniquely required of the activity would eliminate the search to the Data Dictionary file. Also, by allowing the Data Dictionary to contain specialized usages of some entities would enlarge the declaration capabilities; the specialized use could be marked with a keyword matching the ERA Activity Block name. For example, entities as dates can then be more meaningful; the data could be calendar, or Julian. The date could be specified as month day year or day month year. In this way more than one use of an entity can be stated.

References

<Cro83> Cross, F.E.,
The Module Generator -
A Practical Definition for a Software Module,
Proc CompSac 83, pp. 121-133.

<Dou83> Douglas, John H.,
New Computer Architectures Tackle Bottleneck,
High Technology, June 1983, pp. 71-78.

<Dru83> Druffel, Larry E., Redwine, Samual T. Jr.,
and Riddle, William E.,
The Stars Program: Overview and Rationale,
IEEE Computer, Vol. 16, No. 11, November 1983,
pp. 21-29.

<Gla83> Glazer, Sarah,
Application Generators: Automating the Art of Programming,
Mini-micro Systems, May 1983, pp. 125-132.

<Hag75> Hagamen, W. D., Linden, D. J., Mai,
K. F., Newell, S. M., and Weber, J. C.,
A Program Generator,
IBM System Journal, Volume 14, No. 2, 1975, pp. 102-133.

<Kan83> Kahn, Robert E.,
A New Generation in Computing,
IEEE Spectrum, Vol. 20, No. 11, November 1983, pp. 36-41.

<Kim83> Kim, K. H.,
A Look at Japan's Development of Software
Engineering Technology,
IEEE Computer, Vol. 16, No. 5, May 1983, pp. 26-37.

<Ler82> Lerner, Eric J.,
Computers Software II Automating Programming,
IEEE Spectrum, Vol. 19, No. 8, August 1982, pp. 28-33.

<Mar83> Martin, Edith W.,
The Context of Stars,
IEEE Computer, Vol. 16, No. 11, November 1983, pp. 14-17.

<Rif83> Rifkin, Edward M., and Williams, Steve,
The C Language: Key to Portability,
Computer Design, August 1983, pp. 143-150.

<Tre82> Treleaven, Philip C. and Lima, Isabel Gouveia,
Japan's Fifth-Generation Computer Systems,
IEEE Computer, Vol. 15, No. 8, August 1982, pp. 79-88.

Appendix A

A.1 Input File Specifications and Examples

A.1.1 Data Dictionary BNF

```
<data_dictionary> ::= <definition> |
                      <definition> <data_dictionary>
<definition> ::= <definition_header> <definition_body>
<definition_header> ::= 'NAME       : ' <entity_name>
<entity_name> ::=  $ <word> $
<definition_body> ::= <attribute_desc> |
                      <attribute_desc> <definition_body>
<attribute_desc> ::= <attribute_word> <attribute_text>
<attribute_word> ::= 'DESCRIPTION: ' | 'COMPOSITION: ' |
                     'TYPE       : ' | 'USAGE      : ' |
                     'VALUES     : ' | 'RANGE      : ' |
                     'PASCAL     : ' | <capital_word>
<attribute_text> ::= <word> | 'char' | 'array' | 'string' |
                     'const' | <range> | '$' <word> '$'
<word> ::=  <char> | <char> <word>
<char> ::= <lower_case_char> | <symbol>
<symbol> ::=  '_'   (underscore)
<range> ::= <number> <sp> <sp> <number>
<number> ::= <numeric> | <numeric> <numeric>
<numeric> ::=  0 | ... | 9
<lower_case_char> ::=  a | ... | z | <numeric>
<capital_word> ::= <capital_letter> |
                   <capital_letter> <capital_word>
<capital_letter> ::= A | ... | Z
<sp> ::= ' '
```

A.1.2 Data Dictionary File Example

```
NAME        : $name_of_game$
DESCRIPTION: this is the name of the game to be continued
TYPE        : char
USAGE       : array
RANGE       : 01  20
NAME        : $store_message$
DESCRIPTION: the message returned from a *store* command
TYPE        : char
USAGE       : string
RANGE       : 01  14
PASCAL      : array [01..14] of char
NAME        : $retrieve_message$
DESCRIPTION: the message returned from a *retrieve* command
TYPE        : char
USAGE       : string
RANGE       : 01  20
PASCAL      : array [01..20] of char
NAME        : $area$
DESCRIPTION: a total outside surface, measured in sq units
TYPE        : real
NAME        : $pi$
DESCRIPTION: the ratio of the circumference of a circle
DESCRIPTION: to its diameter
USAGE       : const
VALUES      : 3.14159
NAME        : $radius$
DESCRIPTION: any straight line from the center to the
DESCRIPTION: periphery of a circle or sphere
TYPE        : real
NAME        : $move$
DESCRIPTION: the chess move to and from the computer
COMPOSITION: $position$ $comma$ $position$
NAME        : $move_message$
DESCRIPTION: the message returned from after the *move* command
TYPE        : char
USAGE       : string
RANGE       : 01  12
```

```
PASCAL      : array [01..12] of char
NAME        : $computer_move_message$
DESCRIPTION:  the computer move message
TYPE        : char
NAME        : $board_display$
DESCRIPTION: the view of the game board
TYPE        : char
USAGE       : array
RANGE       : 01  24 01  79
PASCAL      : array [01..24] of array [01..79] of char
NAME        : $comma$
DESCRIPTION: this is the character ',' (comma)
DESCRIPTION: this can be used as a word separator or in a record
USAGE       : const
VALUES      : ','
NAME        : $allwhite$
DESCRIPTION: the placement of the white pieces on the board
TYPE        : $white$
NAME        : $white$
DESCRIPTION: the placement of the white pieces on the board
TYPE        : char
NAME        : $black$
DESCRIPTION: the placement of the black pieces on the board
TYPE        : $move_message$
NAME        : $position$
DESCRIPTION: the value (rank) and piece designation of each piece
COMPOSITION: $piece$ $rank$
NAME        : $piece_position$
DESCRIPTION: the board position of each piece
COMPOSITION: $piece$ $space$ $position$
NAME        : $space$
DESCRIPTION: the value ' 40'
USAGE       : const
VALUES      : ' '
NAME        : $piece$
DESCRIPTION: the designation of each piece
TYPE        : enumerator
VALUES      : kr,kk,kb,k,q,qb,qk,qr,p
PASCAL      : (kr,kk,kb,k,q,qb,qk,qr,p)
NAME        : $rank$
```

```
DESCRIPTION:
TYPE        : enumerator
VALUES      : A,B,C,D,E,F,G,H
PASCAL      : (A,B,C,D,E,F,G,H)
NAME        : $board_matrix$
DESCRIPTION: this is the matrix of the board
TYPE        : $piece$
USAGE       : array
RANGE       : 01  08 01  08
PASCAL      : array [01..08] of array [01..08] of char
NAME        : $chess_board$
DESCRIPTION: this is the board lay-out
TYPE        : $board_matrix$
NAME        : $board_description$
DESCRIPTION: placement of the white and black pieces
COMPOSITION: $allwhite$ $black$
NAME        : $status$
DESCRIPTION: the game status
TYPE        : char
USAGE       : string
RANGE       : 01  20
PASCAL      : array [01..20] of char
```

A.1.3 ERA Requirements Specification BNF

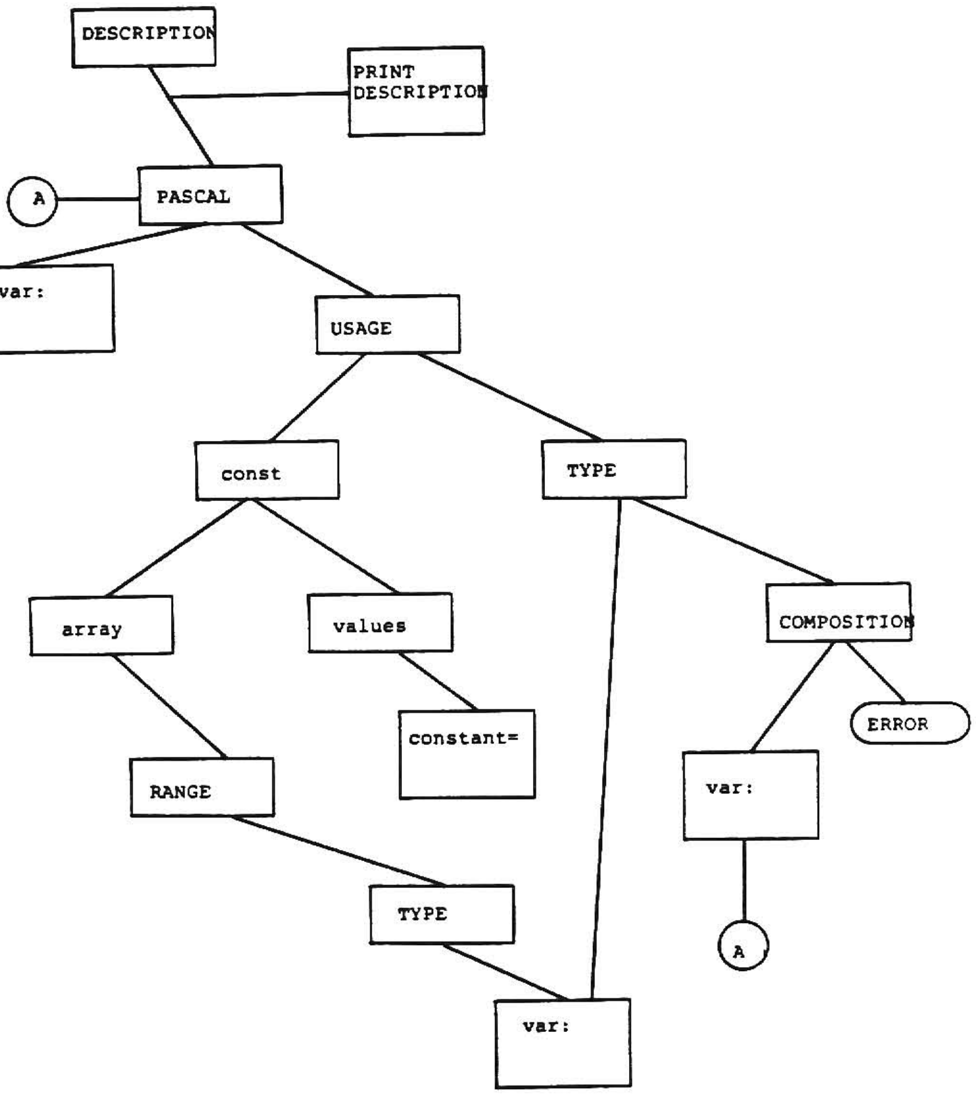
```
<frame> ::= <frame_header> <frame_body>

<frame_header> ::= <function_header> <word> <NL>

<function_header> ::= 'Activity    : '

<frame_body> ::= <relation> | <relation> <frame_body>

<relation> ::= <relation_type> <relation_value> <NL>

<relation_type> ::= '   input    : ' |
                    '   output   : ' |
                    '   uses     : , |
                    <word>

<relation_value> ::= <i_o_data_name> | <text_lines>

<i_o_data_name> ::= '$' <word> '$'

<word> ::= <char> | <char> <word>

<char> ::= <lower_case_char> | <symbol>

<NL> ::= '\n'

<symbol> ::= '_'

<lower_case_char> ::= a | b | ... | z | 0 | 1 | ... | 9

<text_lines> ::= <word> | <word> <text>
```

A.1.4 ERA Requirements Specification Activity Block File Example

```
Activity    : Initialize_board
   keywords : standard board,initialize,place_pieces
   input    : $area$
   output   : $store_message$
   required-mode : *START*
   necessary-condition : last user command is $start$
   assertion : The output board is a correct representation
               of the standard starting configuration for
               chess
```

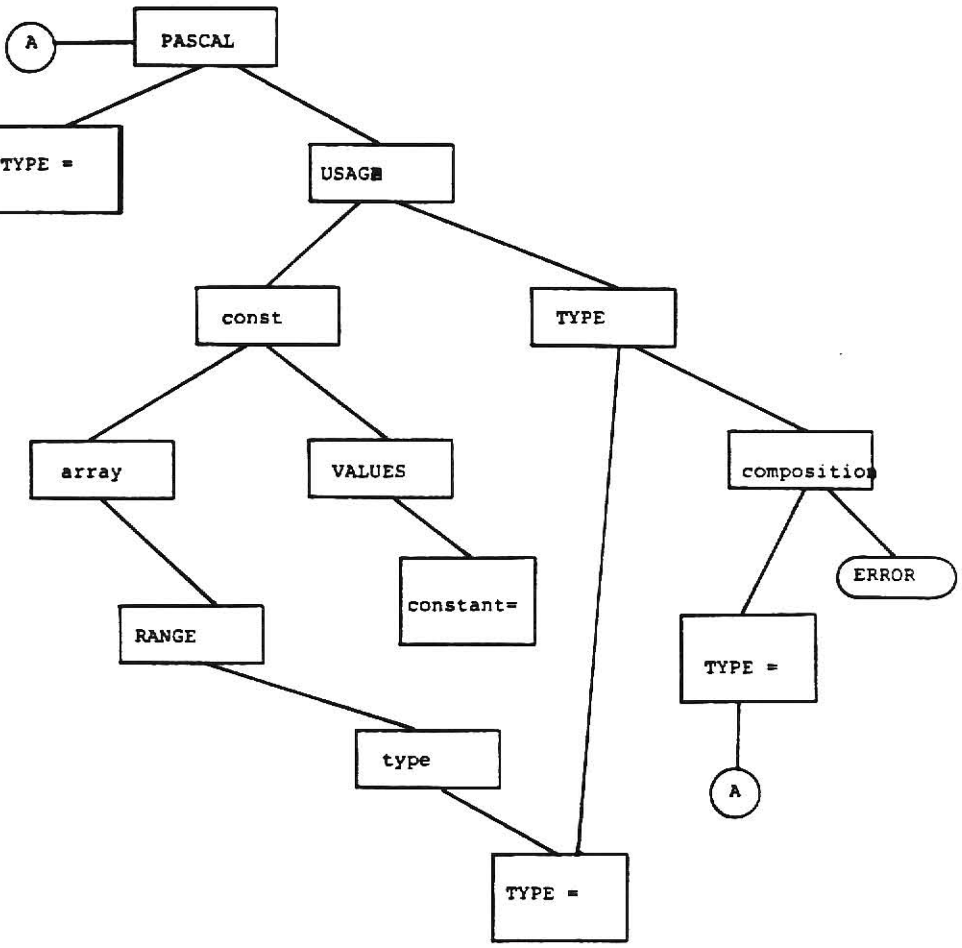
Appendix B

B.1 I/O Mapping



Appendix C

C.1 Type Mapping



Appendix D

D.1 User Instructions

D.1.1 Run Instructions

The Module Declaration Generator is a C Language program which will execute in a UNIX environment. To execute the Module Declaration Generator the following is required.

1. The name of the MDG program.
2. The name of the ERA Activity Block file. This file will contain one (1) ERA Activity Block for which a PASCAL program is to be generated.
3. The name of the output file in which the PASCAL program framework is to be placed.
4. The name of the Data Dictionary file.

Example :

```
a.out erain pasout.p ddfile
```

To simplify the execution of the MDG, the use of a shell could be employed.

Example :

```
        a.out <$1 pasout.p ddfile
```

In this example of a shell "a.out" is the MDG program name. The MDG program will accept "ddfile" as input as well as a file to be named when executed ,"$1". The "pasout.p" is the output file which will contain the PASCAL program framework. This output file can be renamed to something more meaningful by the user.

Now to execute this shell, the user will key the shell name (in this case the shell is named bonsai), a blank and the name of the ERA Activity Block.

Example :

```
        bonsai erain
```

The Module Declaration Generator will execute, generate the PASCAL program framework and place it into the output file "pasout.p". The user can rename the output file to something more meaningful. Note the MDG will generate to the user a message indicating the name of this output file (see Appendix - User Messages).

D.1.2 User Messages

During the execution of the Module Declaration Generator messages are relayed to the user. These messages indicate the success or failure of the MDG.

The following are examples of user messages returned from a successful operation of the MDG:

```
Activity    : Initialize_board
   keywords : standard board,initialize,place_pieces
   input    : $area$
   output   : $store_message$
   required-mode : *START*
   necessary-condition : last user command is $start$
   assertion : The output board is a correct representation
               of the standard starting configuration for
               chess

will attempt to generate a pascal module

module file is here.p
```

The first nine (9) lines are the ERA Activity Block as read by the MDG. These lines will also appear in the PASCAL program framework as comments. The user can verify the completeness of the Activity Block.

The tenth line indicates the MDG will attempt to generate the PASCAL module as requested. The ERA Activity Block has been determined to be with in the the physical constraints of the MDG.

The eleventh line displays the output file name. This output file contains the PASCAL program framework successfully generated by the MDG. The name is known to the MDG, since it is one of the file names required for execution (see Run Instructions).

Other messages can be relayed to the user; these messages indicate either a system failure or a limitation of the MDG has been exceeded.

The following messages can be returned to the user:

Message	Reason Issued
"cant open file" file_name	The designated file could not be opened.
"input too big to print"	The number of lines of the Activity Block exceed the MDG limits.
"attempted to find the era activity information it was not found in the expected location the pascal module will not be generated as requested"	The MDG could not locate the ERA Activity Block.
"type format unknown"	The entity specified in the ERA Activity Block was not correctly specified in the Data Dictionary.
"malloc error"	The MDG executed the malloc function and was not successful.

D.1.3 Sample Output

```
program Initialize_board (input,output);
{ Activity    : Initialize_board }
{    keywords : standard board,initialize,place_pieces }
{    input    : $area$ }
{    output   : $store_message$ }
{    required-mode : *START* }
{    necessary-condition : last user command is $start$ }
{    assertion : The output board is a correct
                               representation of the    }
{               standard starting configuration for
                                               chess    }
{ $area$ DESCRIPTION: a total outside surface, measured
                                           in sq units }
{ $store_message$ DESCRIPTION: the message
                        returned from a *store* command }
var area : real;
    store_message : array [01..14] of char;
begin

{    put  the program logic here    }

end.
```

Appendix E

E.1 Program Specification

Procedure name : Main

Input :

1. The ERA Activity Specification File.
2. The Data Dictionary File.

Output :

1. The PASCAL module framework file.
2. User messages.

Description :

The main procedure controls the processing flow of the MDG. This code will execute the other subroutines which will read the ERA Activity Block and the Data Dictionary. The main will also, execute the subroutines formatting the PASCAL module framework and generating the user messages.

Subroutine name : readlines

Input :

1. The MDG maximum limit of ERA Activity Block lines.

Output :

1. lineptr - The pointers to the storage area of the ERA Activity Block lines.

2. nlines - The number of ERA Activity Block lines read.

Description :

The readlines subroutine reads and stores the ERA Activity Block lines. These lines will be used to generate user messages and comments in the PASCAL module framework generated.

Subroutine name : writelines

Input :

1. lineptr - The pointers to the storage area of the ERA Activity Block lines.

2. nlines - The number of ERA Activity Block lines read.

Output :

1. The user messages displaying the ERA Activity Block lines.

Description :

The writelines subroutine writes the ERA Activity Block lines to the user's terminal.

Subroutine name : getline

Input :

1. s - A string of characters.
2. lim - The maximum number of characters per line of the ERA Activity Block to be read.

Output :

1. i - The number of characters of the ERA Activity Block line read.

Description :

The getline subroutine reads the ERA Activity Block line one character at a time and returns the number of characters read.

Subroutine name : getdd

Input :

1. es - The entity to be located in the Data Dictionary.
2. lim - The maximum number of characters per line of the Data Dictionary to be read.
3. ddlred - The number of lines read and stored from the Data Dictionary.

Output :

1. ddptr - The pointers to the Data Dictionary lines read and stored.

Description :

The getdd subroutine reads the Data Dictionary. Starting from the beginning each time it is executed, this subroutine will match the entity to the entity of the Data Dictionary keyword "NAME". When matched the keywords "DESCRIPTION", "TYPE", "RANGE", "COMPOSITION", "USAGE", "VALUES" and/or "PASCAL" belonging to this entity will be stored for processing in another subroutine.

Subroutine name : readdd

Input :

1. lim - The MDG maximum limit of Data Dictionary lines.
2. rs - The line of characters from the Data Dictionary file.
3. pt2 - The pointer to the Data Dictionary file.

Output :

1. rs - The line of characters from the Data Dictionary file.

Description :

The readdd subroutine reads the Data Dictionary file line by line character by character.

Subroutine name : havepas

Input :

1. ddptr - The pointers to the Data Dictionary lines.
2. ddlred - The number of lines read and stored from the Data Dictionary.

Output :

1. w - A switch used for loading variable or constant data type entities.
2. rol - An array for storing the information found in the Data Dictionary.

Description :

This subroutine searches the Data Dictionary lines stored for the line with the keyword "PASCAL". When this keyword is located, the information contained on this line is formatted for PASCAL declaration generation.

Subroutine name : havetyp

Input :

1. ddptr - The pointers to the Data Dictionary lines.

2. ddlred - The number of lines read and stored from the Data Dictionary.

Output :

1. for-type - An array used for returning entities used to define the entity being processed.

2. rol - An array for storing the information found in the Data Dictionary.

Description :

this subroutine searches the Data Dictionary lines stored for the line with the keyword "TYPE". When this keyword is located, the information contained on this line is formatted for PASCAL declaration generation. If the entity is defined by other entities, those entities are returned to be processed later.

Subroutine name : haveusa

Input :

1. ddptr - The pointers to the Data Dictionary lines.

2. ddlred - The number of lines read and stored from the Data Dictionary.

Output :

1. for-type - An array used for returning entities used to define the entity being processed.

2. rol - An array for storing the information found in the Data Dictionary.

3. w - A switch used for loading variable or constant data type entities.

Description :

This subroutine searches the Data Dictionary lines stored for the line with the keyword "USAGE". When this keyword is located, the information contained on this line is formatted for PASCAL declaration generation. If the entity is defined by other entities, those entities are returned to be processed later.

Subroutine name : havecom

Input :

1. ddptr - The pointers to the Data Dictionary lines.
2. ddlred - The number of lines read and stored from the Data Dictionary.

Output :

1. comptr - Pointers to entities used to define the entity being processed.

Description :

This subroutine searches the Data Dictionary lines stored for the line with the keyword "COMPOSITION". When this keyword is located, the information contained on this line is formatted for PASCAL declaration generation. If the entity is defined by other entities, pointers to those entities are returned to be processed later.

Subroutine name : loadtype

Input :

1. for-type - Entities to be added to the array of entities.
2. entypno - The number of entities in the array of entities.
3. entyp - The pointer to the array of entities.
4. entline - The pointer to the ERA entity lines.
5. entno - The number of entities in the array of entities from the ERA Activity Block.

Output :

1. Returns the number of entities in the array of entities.

Description :

This subroutine loads the entities into the array of entities. This array is processed sequentially to locate entity information in the Data Dictionary.

Subroutine name : checkent

Input :

1. for-type - Entities to be added to the array of entities.

2. entline - The pointer to the ERA entity lines.

3. entno - The number of entities in the array of entities.

Output :

1. entypno - The number of entities in the array of entities.

Description :

This subroutine checks the presents of an entity in the array of entities to prevent duplicate entries.

Appendix F

F.1 Program Source Code

The Main procedure :

```
/*    pgm name mdg.c                                    */
/*                                                      */
/*    author   Joseph Libiers                           */
/*                                                      */
/*    this program will read an era specification      */
/*    and generate a framework for a pascal module.*/
/*    the era specification will appear as comments*/
/*    in the generated module.                          */
/*    this pgm will also read a data dictionary         */


#include <stdio.h>
#define eof -1
#define null 0
#define lines  20      /* max lines to be read from the era specification*/
#define ddlmax 100     /* max nbr of lines read from data dictionary*/
#define modname 20     /* max nbr characters to be used for a module name*/
#define inlp 14        /* offset of era  line to keyword name */
#define ddlsiz 60      /* max nbr of char in each line of data dictionary*/
#define ddkey  13      /* length of data dict keywords */
#define paslmax 50     /* length of a pascal code line in the data dict */
#define entl 26        /* max length +1 of an entity name (including $) */
#define pascode 80     /* max length of pascal code line generated */
#define nl  "\n"       /*  new line character */
#define bufsiz 512
#define actline "Activity      : "
#define inpline "    input     : "
#define outline "    output    : "
#define useline "    uses      : "
#define namline "NAME        : "
#define desline "DESCRIPTION: "
#define typline "TYPE        : "
#define ranline "RANGE       : "
#define usaline "USAGE       : "
#define valline "VALUES      : "
#define pasline "PASCAL      : "
#define comline "COMPOSITION: "
```

```
#define const    "const"
#define array    "array"
#define maxcom  3
main (argc, argv)
int argc;
char *argv[];
{
        FILE *pt1, *pt2;
        char *lineptr[lines];   /* pointer to era spec lines */
        char *entline[lines];   /* pointer to era entity line */
        char *ddptr[ddlmax];    /* pointer to dd line */
        char phd2[modname];     /* array for module name */
        char pera[entl];        /* array for loading entity names */
        char ent_name[entl];    /* the entity name for pascal code use*/
        char rol[50];           /* array for loading dd code info */
        char for_type[20];      /* for return of entity from type */
        char codel[pascode];    /* the code line */
        char *var[lines];       /* array of var lines */
        char *v;                /* pointer for var */
        int  varno;             /* the number of var code lines */
        char *con[lines];       /* array of const lines */
        char *ca;               /* pointer for con */
        int  conno;             /* the number of const code lines */
        char *typ[lines];       /* array of type lines */
        char *ta;               /* pointer for typ */
        int  typno;             /* the number of type code lines */
        char *entyp[30];        /* array of entities for typ */
        char *enta;             /* pointer for entyp */
        int  entypno;           /* number of entities in entyp */
        char dumb[50];          /* will be loaded will the activity line info*/
        char dumbl[70];         /* will be used as a dummy array */
        char pas[paslmax];      /* string for pascal declarations */
        int  nlines;            /* number of era spec lines read */
        int  ddlred;            /* nbr of lines read from data dictionary */
        int  llent;             /* line length of the data dict line returned*/
        int  i, m, i2, m2, j, j2; /* subscripts */
        int  slen;              /* length of a string */
        int  entno;             /* the number of entities from the era    */
        int  doit;              /* for hierachy of dd keywords */
        char *pent;             /* pointer to the entity names  */
char *malloc();          /* required for dynamic allocation of pent */
char *s1[inlp];          /* contains activity line format to match */
char w[2];               /* used as a switch in loading var type const*/
char *comptr[maxcom];    /* pointer to composite entities */
int  comno;              /* number of composite entities */


/* open for pt1 output */
if ((pt1 = fopen(argv[1], "w")) == null)
{
        printf("cant open file %s\n", argv[1]);
        perror();
        return;
}


/* reads and stores the era specification */
/* returns pointers to spec lines and the */
/* number of lines read                   */
if (( nlines = readlines (lineptr, lines )) >= 0 )
{
        /* write of the era specification */
        writelines (lineptr, nlines );
}
else
        printf("input too big to print\n");

/* loads s1 with activity line format */
strcpy(s1, actline);
if ((strncmp(s1, lineptr[0], inlp)) == 0)
        /* compares s1 to first era spec line */
        printf("will attempt to generate a pascal module\n");
else
{
printf("attempted to find the era activity information\n");
printf("it was not found in the expected location\n");
printf("the pascal module will not be generated as requested\n");
        return;
}

strncpy(dumb,lineptr[0],31);
i2 = 0;
for (i=inlp; i2<modname-1; i++)
        phd2[i2++]=dumb[i];


/* PASCAL program header line is printed */
fprintf(pt1,"%s %s %s\n","program",phd2, "(input,output);");

for (i=0; i<nlines; i++)
        fprintf(pt1,"%s %s %s\n","(", lineptr[i], ")");

entno=0;

strcpy(s1, inpline);
i=0;
while(i<nlines-1)

{
        while ((i<nlines)&&(strncmp(s1,lineptr[i],inlp)!=0))
                i++;
        if (i<nlines)
        {
         strncpy(dumb, lineptr[i], 39);
         i2=0;
         for (m=inlp; i2<entl-2 ; m++)
           pera[i2++]=dumb[m];
           pera[i2]='\0';
           if((checkent(pera,entline,entno)==0)&&
             ((pent = malloc(entl))!= null))
                {
                        strcpy(pent, pera);
                        entline[entno++] = pent;
                }
                ++i;
        }
}
```

```
strcpy(s1, outline);
i=0;
while (i<nlines-1)
{
        while ((i<nlines)&&(strncmp(s1,lineptr[i],inlp)!=0))
                i++;

        if (i<nlines)
        {
         strncpy(dumb, lineptr[i], 39);
         i2=0;
         for (m=inlp; i2<entl-2 ; m++)
           pera[i2++]=dumb[m];
           pera[i2]='\0';
           if((checkent(pera,entline,entno)==0)&&
             ((pent = malloc(entl))!= null))
                {
                        strcpy(pent, pera);
                        entline[entno++] = pent;
                }
                ++i;
        }
}
/* removes the entity name from each input line */
/* and stores it  */
strcpy(s1, useline);
i=0;
while (i<nlines-1)
{
        while ((i<nlines)&&(strncmp(s1,lineptr[i],inlp)!=0))
        i++;

        if (i<nlines)
        {
         strncpy(dumb, lineptr[i], 39);
         i2=0;
         for (m=inlp; i2<entl-2 ; m++)
           pera[i2++]=dumb[m];
         pera[i2]='\0';
```

```
          if((checkent (pera,entline,entno)==0)
             &&((pent = malloc(entl)) != null))
          {
           strcpy(pent, pera);
           entline[entno++] = pent;
          }
          ++i;
         }
}
varno=0;
conno=0;
typno=0;
entypno=0;
/* the data dictionary is read for each entity */
for (m=0; m< entno; m++)
{
 ddlred=(getdd(ddptr, ddlsiz, entline[m], argv, pt2, ddlred));
 if (ddlred>0)
 {
   for (i=0; i <ddlred; i++)
           {
             if ((strncmp(ddptr[i], desline, ddkey)==0))
                    {
                          strcpy(dumbl, ddptr[i]);
                          fprintf(pt1,"{ %s %s }\n",
                          entline[m], ddptr[i]);
                    }
            }
            i2=0;
            strcpy(dumb, entline[m]);
            slen=(strlen(dumb));
            for (m2=0; m2<slen-1; m2++)
            {
                    if ((dumb[m2]!= '$'))
                          ent_name[i2++]=dumb[m2];
                    if ((dumb[m2] == '\n'))
                          ent_name[i2] = '\0';
            }
            ent_name[i2++]= '\0';
            strcpy(codel, ent_name);
```

```
doit=0;
w[0]='v';
/* PASCAL keyword process check */
if((havepas(ddlred, ddptr, rol, w)) > 0)
{
        doit=10;
}
if(doit <= 0)
   /* USAGE keyword process check */
   if((haveusa(ddlred, ddptr, rol, w, for_type))>0)
   {
     doit=9;
     slen=(strlen(for_type));
     if(slen=(strlen(for_type))>0)
     entypno=(loadtype(for_type, entypno,
                entyp, entline, entno));
    }


if(doit<=0)
    /* TYPE keyword process check */
    if((havetyp(ddlred, ddptr, rol, for_type)) > 0)
    {
      doit=8;
      slen=(strlen(for_type));
      if(slen=(strlen(for_type))>0)
      entypno=(loadtype(for_type, entypno,
                entyp,entline, entno));
    }

if(doit<=0)
    /* COMPOSITION keyword process check */
    if((comno=(havecom(ddlred, ddptr, comptr))) > 0)
    {
     if(comno>0)
     {
      strcat(codel, ": record ");
      for(j=0; j<comno; j++)
       {
        ddlred=(getdd(ddptr, ddlsiz, comptr[j],
         argv, pt2, ddlred));
if (ddlred>0)
{
 i2=0;
 strcpy(dumb, comptr[j]);
 slen=(strlen(dumb));
 for (m2=0; m2<slen-1; m2++)
  {
   if ((dumb[m2]!= '$'))
    ent_name[i2++]=dumb[m2];
   if ((dumb[m2] == '\n'))
    ent_name[i2] =  '\0';
  }
 ent_name[i2++]= '\0';
 strcat(codel, ent_name);
 doit=0;
  /* PASCAL keyword process check */
 if((havepas(ddlred, ddptr, rol, w)) > 0)
 {
  doit=10;
 }
 if(doit <= 0)
    /* USAGE keyword process check */
    if((haveusa(ddlred, ddptr, rol,
         w, for_type))>0)
    {
     doit=9;
     slen=(strlen(for_type));
     if(slen=(strlen(for_type))>0)
      entypno=(loadtype(for_type,
        entypno, entyp, entline, entno));
    }


 if(doit<=0)
    /* TYPE keyword process check */
    if((havetyp(ddlred, ddptr, rol,
         for_type)) > 0)
    {
     doit=8;
```

```
          slen=(strlen(for_type));
          if(slen=(strlen(for_type))>0)
           entypno=(loadtype(for_type,
             entypno, entyp, entline, entno));
         }
      if(doit<=0)
       {
        strcpy(rol, "type format unknown\0");
       }
      strcat(codel, " : ");
      strcat(codel, rol);
      strcat(codel, ";");
      slen=(strlen(codel));

      if((v = malloc(slen))==null)
       {
        printf("malloc error");
       }
       else
       {
        codel[slen]='\0';
        strcpy(v, codel);
        var[varno++] = v;
        strcpy(codel, "      ");
        }
    }
  }
  doit=7;
  strcpy(codel, "             end;");
  if((v = malloc(slen))==null)
   {
    printf("malloc error");
   }
   else
   {
    codel[slen]='\0';
    strcpy(v, codel);
    var[varno++] = v;
    }
}
```

```
    }
if(doit<=0)
{
        strcpy(rol, "type format unknown\0");
}

/* stores constant lines */
if((w[0]=='c')&&(doit!=7))
{
        strcat(codel, " = ");
        strcat(codel, rol);
        strcat(codel, ";");
        slen=(strlen(codel));
        if ((ca = malloc(slen))== null)
        {
                printf("malloc error");
        }
        else
        {
                codel[slen] = '\0';
                strcpy(ca, codel);
                con[conno++] = ca;
        }
}
/* stores var lines */
if((w[0]=='v')&&(doit!=7))
{
        strcat(codel, " : ");
        strcat(codel, rol);
        strcat(codel, ";");
        slen=(strlen(codel));
        if ((v = malloc(slen))== null)
        {
                printf("malloc error");
        }
        else
        {
                codel[slen] = '\0';
                strcpy(v, codel);
```

```
                                        var[varno++] = v;
                                }
                        }
                }
}
m=0;
while(m<entypno)
{
        ddlred=(getdd(ddptr, ddlsiz, entyp[m], argv, pt2, ddlred));
        i2=0;
        strcpy(dumb, entyp[m]);
        slen=(strlen(dumb));
        for (m2=0; m2<slen-1; m2++)
        {
                if ((dumb[m2]!= '$'))
                        ent_name[i2++]=dumb[m2];
                if ((dumb[m2] == '\n'))
                        ent_name[i2] = '\0';
        }
        ent_name[i2++]= '\0';
        strcpy(code1, ent_name);
        doit=0;
        if (ddlred>0)
        {
         if((havepas(ddlred, ddptr, rol, w)) > 0)
          {
           doit=10;
          }
         if(doit <= 0)
          if((haveusa(ddlred, ddptr, rol, w, for_type))>0)
           {
            doit=9;
            slen=(strlen(for_type));
            if(slen=(strlen(for_type))>0)
             entypno=(loadtype(for_type, entypno,
                      entyp, entline, entno));
           }

         if(doit<=0)
          if((havetyp(ddlred, ddptr, rol, for_type)) > 0)
```

```
 {
  doit=8;
  slen=(strlen(for_type));
  if(slen=(strlen(for_type))>0)
   entypno=(loadtype(for_type, entypno,
            entyp, entline, entno));
 }

if(doit<=0)
 if((comno=(havecom(ddlred, ddptr, comptr))) > 0)
 {
  if(comno>0)
   {
    strcat(codel, "= record ");
    for(j=0; j<comno; j++)
     {
      ddlred=(getdd(ddptr, ddlsiz, comptr[j],
               argv, pt2, ddlred));
      if (ddlred>0)
       {
        i2=0;
        strcpy(dumb, comptr[j]);
        slen=(strlen(dumb));
        for (m2=0; m2<slen-1; m2++)
         {
          if ((dumb[m2]!= '$'))
            ent_name[i2++]=dumb[m2];
          if ((dumb[m2] == '\n'))
            ent_name[i2] = '\0';
         }
        ent_name[i2++]= '\0';
        strcat(codel, ent_name);
        doit=0;
        if((havepas(ddlred, ddptr, rol, w)) > 0)
         {
          doit=10;
         }
        if(doit <= 0)
         if((haveusa(ddlred, ddptr, rol, w, for_type))>0)
          {
```

```
      doit=9;
      slen=(strlen(for_type));
      if(slen=(strlen(for_type))>0)
       entypno=(loadtype(for_type, entypno,
                 entyp, entline, entno));
     }

    if(doit<=0)
     if((havetyp(ddlred, ddptr, rol, for_type)) > 0)
      {
       doit=8;
       if(slen=(strlen(for_type))>0)
        entypno=(loadtype(for_type, entypno,
                  entyp, entline, entno));
      }
    if(doit<=0)
     {
      strcpy(rol, "type format unknown\0");
     }
    strcat(codel, " : ");
    strcat(codel, rol);
    strcat(codel, ";");
    slen=(strlen(codel));

    if((ta = malloc(slen))==null)
     {
      printf("malloc error");
     }
     else
     {
      codel|slen|='\0';
      strcpy(ta, codel);
      typ|typno++| = ta;
      strcpy(codel, "     ");
     }
   }
 }
 doit=7;
 strcpy(codel, "               end;");
```

```
  if((ta = malloc(slen))==null)
   {
    printf("malloc error");
   }
   else
   {
    codel[slen]='\0';
    strcpy(ta, codel);
    typ[typno++] = ta;
   }
 }

}

if(doit<=0)
 {
  strcpy(rol, "type format unknown\0");
 }
   if((strlen(rol)>0)&&(doit!=7))
        if (w[0]=='c')
        {
             strcat(codel, " = ");
             strcat(codel, rol);
             strcat(codel, ";");
             slen=(strlen(codel));
             if ((ca = malloc(slen))== null)
             {
                  printf("malloc error");
             }
             else
             {
                  codel[slen] = '\0';
                  strcpy(ca, codel);
                  con[conno++] = ca;
             }
        }
        else
        {
             strcat(codel, " = ");
             strcat(codel, rol);
```

```
                                strcat(code1, ";");
                                slen=(strlen(code1));
                                if ((ta = malloc(slen))== null)
                                {
                                        printf("malloc error");
                                }
                                else
                                {
                                        code1[slen] = '\0';
                                        strcpy(ta, code1);
                                        typ[typno++] = ta;
                                }
                        }

        }
        m++;
}
if(conno>0)
{
        fprintf(pt1,"const %s\n", con[0]);
        for (i=1; i<conno; i++)
                fprintf(pt1,"      %s\n", con[i]);
}
if(typno>0)
{
        fprintf(pt1,"type %s\n", typ[typno-1]);
        for(i=typno-2; i>=0; i--)
                fprintf(pt1,"     %s\n", typ[i]);
}
if(varno>0)
{
        fprintf(pt1,"var %s\n", var[0]);
        for (i=1; i<varno; i++)
                fprintf(pt1,"    %s\n", var[i]);
}
fprintf(pt1,"%s\n", "begin");
fprintf(pt1, "\n");
fprintf(pt1, "%s\n", "{    put  the program logic here   }");
fprintf(pt1, "\n");
fprintf(pt1,"%s\n", "end.");
```

```
        printf("module file is %s\n", argv|1|);
        fclose (pt1);
}
```

Subroutine readlines :

```
#define maxlen 75
 /* max num of characters in the era specification line */

readlines(lineptr, maxlines)  /*  read era specification lines */
char  *lineptr[];            /*  era spec storage area      */
int   maxlines;
{
        int   len, nlines;
        char *p, *malloc(), line [maxlen];

        nlines = 0;
        while (( len = getline( line, maxlen )) > 0)

                if (nlines >= maxlines )
                {
                        return (-1);
                }
                else
                        /* allocate space for array p */
                        if (( p = malloc(len) ) == null )
                        {
                          return(-1);
                        }
                        else
                        {
                          line[len-1] = '\0'; /* zap newline character*/
                          strcpy (p, line);    /* copies line to p */
                          lineptr[nlines++] = p;   /* stores pointer to p */
                        }
        return(nlines);
}
```

Subroutine writelines :

```
/* writes the ERA Spec to the user */

writelines(lineptr, nlines)
char  *lineptr[];
int   nlines;
{
        int i;

        for  (i=0; i < nlines; i++)
                printf("%s\n", lineptr[i]);
}
```

Subroutine getline :

```
getline (s, lim)      /* get line into s, return length */
char  s[];
int   lim;
{
        int c, i;
        i = 0;
        while ((--lim >0)&&((c=getchar())!=eof)&&(c!='\n'))
                s[i++] = c;
        if (c == '\n')
        {
                s[i++]=c;
        }
        s[i] = '\0';
        return(i);
}
```

Subroutine getdd :

```
getdd(ddptr, lim, es, argv, pt2, ddlred)

/*   will read the data dictionary   */

FILE *pt2;
char *argv[];
char *ddptr[];
char es[];
int  lim;
int  ddlred;

{
        char *dd;
        char *malloc();
        char s[75];
        char ls[75];
        int i;
        int  slen;
        int  len;
        int m;
        ddlred = 0;
        if ((pt2 = fopen (argv[2], "r")) == null)
        {
                printf("cant open file %s\n", argv[2]);
                perror();
                return(-1);
        }

        strcpy(s, namline);
        strcat(s, es);
        slen=(strlen(s));
        i=0;

        while((len=readdd(ls,lim,pt2)) > 0)
                if ((strncmp(ls, s, slen))==0)
                        break;
        strcpy(s, namline);
        if (len > 0)
                while(((len=readdd(ls, lim, pt2)) >0)&&(strncmp(ls, s, 4)!=0))
                {
                        if ((strncmp(ls, desline, ddkey)==0) ||
                            (strncmp(ls, typline, ddkey)==0) ||
                            (strncmp(ls, comline, ddkey)==0) ||
                            (strncmp(ls, ranline, ddkey)==0) ||
                            (strncmp(ls, usaline, ddkey)==0) ||
                            (strncmp(ls, valline, ddkey)==0) ||
                            (strncmp(ls, pasline, ddkey)==0))
                                if (( dd = malloc(lim) ) == null)
                                {
                                        return(-1);
                                }
                                else
                                {
                                        ls[lim] = '\0';
                                        strcpy(dd, ls);
                                        ddptr[ddlred++] = dd;
                                }
                }
        fclose (pt2);
        return(ddlred);
}
```

Subroutine readdd :

```
readdd(rs,lim,pt2)

/*  will read data dictionary */

FILE *pt2;
char rs[];
int lim;

{
        int c,i;

        i=0;

        while ((--lim >0)&&((c=getc(pt2))!=eof)&&(c!='\n'))
                rs[i++]=c;

        if(c='\n')
                rs[i++]='\0';

        rs[i]='\0';
        return(i);
}
```

Subroutine havepas :

```
havepas(ddlred, ddptr, rol, w)

char rol[];
char *ddptr[];
int ddlred;
char w[];
{
        int i,i2,m;
        int slen;
        char dumbl[80];
        w[0]='v';
        i=0;
        i2=0;
        m=0;
        for (i=0; i< ddlred; i++)
        {
                if ((strncmp(ddptr[i], pasline, ddkey)==0))
                {
                        strcpy(dumbl, ddptr[i]);
                        slen=(strlen(dumbl));
                        for (m=ddkey, i2=0; m<slen; m++)
                                rol[i2++]=dumbl[m];
                        rol[i2]='\0';
                }
        }
        return(i2);
}
```

Subroutine havetyp :

```
havetyp(ddlred, ddptr, rol, for_type)

char rol[];
char *ddptr[];
int ddlred;
char for_type[];
{
        int i, i2, m, m1, m2, m3;
        int slen;
        char dumbl[50];
        char arol[50];
        i=0;
        i2=0;
        m=0;
        m1=0;
        m2=0;
        m3=0;
        for (i=0; i< ddlred; i++)
        {
                if ((strncmp(ddptr[i], typline, ddkey)==0))
                {
                        strcpy(dumbl, ddptr[i]);
                        slen=(strlen(dumbl));
                        for (m=ddkey, i2=0; m<slen; m++)
                                if(dumbl[m] == '$')
                                {
                                        arol[i2++] = '\040';
                                        m2++;
                                }
                                else
                                        arol[i2++]= dumbl[m];

                        arol[i2]='\0';
                        strcpy(rol, arol);
                        m3=(slen-ddkey);
                        if (m2>0)
                        {
```

```
                                                  for(m1=0, m2=ddkey; m1< m3; m2++)
                                                          for_type[m1++] = dumb1[m2];
                                                  for_type[m1] = '\0';
                                          }
                                  }
                          }
                          return(i2);
                  }
```

Subroutine haveusa :

```
haveusa(ddlred, ddptr, rol, w, for_type)

char rol[];
char *ddptr[];
int ddlred;
char w[];
char for_type[];
{
        int i,i2,m;
        int slen;
        char tol[20];
        char dumbl[50];
        char cora[50];
        i=0;
        i2=0;
        m=0;
        for (i=0; i< ddlred; i++)
        {
                if ((strncmp(ddptr[i], usaline, ddkey)==0))
                {
                        strcpy(dumbl, ddptr[i]);
                        slen=(strlen(dumbl));
                        for (m=ddkey, i2=0; i2<slen; m++)
                                cora[i2++]=dumbl[m];
                }
        }
        if(strncmp(cora, const, 5)==0)
        {
                w[0]='c';
                for (i=0; i< ddlred; i++)
                {
                        if ((strncmp(ddptr[i], valline, ddkey)==0))
                        {
                                strcpy(dumbl, ddptr[i]);
                                slen=(strlen(dumbl));
                                for (m=ddkey, i2=0; i2< slen; m++)
                                        rol[i2++]=dumbl[m];
                                }
                        }
                }
                if(strncmp(cora, array, 5)==0)
                {
                        for (i=0; i< ddlred; i++)
                        {
                                if ((strncmp(ddptr[i], ranline, ddkey)==0))
                                {
                                        strcpy(rol, "array [");
                                        strcpy(dumbl, ddptr[i]);
                                        slen=(strlen(dumbl));
                                        for (m=ddkey, i2=7; m< slen; m++)
                                        {
                                                if (dumbl[m] == ' ')
                                                        rol[i2++]= '.';
                                                else
                                                        rol[i2++]=dumbl[m];
                                        }
                                        strcat(rol, "] of ");
                                        if (havetyp(ddlred, ddptr, tol, for_type) >0)
                                                strcat(rol, tol);
                                        else
                                                i2=0;
                                }
                        }
                }
                return(i2);
}
```

Subroutine havecom :

```
havecom(ddlred, ddptr, comptr)

char *comptr[];
char *ddptr[];
int ddlred;
{
        int i,i2,m,m2;
        int slen,slen2;
        char dumbl[50];
        char arol[50];
        char *malloc();
        char *com;
        i=0;
        i2=0;
        m=0;
        m2=0;
        for (i=0; i< ddlred; i++)
        {
                if ((strncmp(ddptr[i], comline, ddkey)==0))
                {
                        strcpy(dumbl, ddptr[i]);
                        slen=(strlen(dumbl));
                        while(m<slen)
                        {
                                m++;
                                if(dumbl[m]=='$')
                                {
                                        m++;
                                        arol[i2++] = '$';
                                        while(dumbl[m]!='$')
                                        {
                                                arol[i2++] = dumbl[m++];
                                        }
                                        arol[i2++] = '$';
                                        arol[i2]='\0';
                                        slen2=(strlen(arol));
                                        if((com = malloc(slen2))==null)
```

```
                    {
                        printf("malloc error");
                        return(0);
                    }
                    else
                    {
                        strcpy(com, arol);
                        comptr[m2++] = com;
                        i2=0;
                    }
                }
            }
        }
    }
    return(m2);
}
```

Subroutine loadtype :

```
loadtype(for_type, entypno, entyp, entline, entno)

char for_type[];
char *entyp[];
int  entypno;
char *entline;
int  entno;

{
        char *enta;
        int  i;
        int slen;
        char *malloc();

        if (entypno>0)
                for(i=0;  i< entypno; i++)
                {
                        if(strcmp(for_type, entyp[i])==0)
                                return(entypno);
                }
        if(checkent(for_type, entline, entno)!=0)
                return(entypno);
        slen=(strlen(for_type));
        if ((enta = malloc(slen))== null)
                printf("malloc error");
        else
        {
                for_type[slen]='\0';
                strcpy(enta, for_type);
                entyp[entypno++]= enta;
        }
        return(entypno);
}
```

Subroutine checkent :

```
checkent(for_type, entline, entno)

char for_type[];
char *entline[];
int  entno;

{
        int  j, j2;

        j2=0;
        if (entno>0)
        {
                for(j=0; j<entno ; j++)
                {
                        if(strcmp(for_type, entline[j])==0)
                                j2++;
                }
        }
        return(j2);
}
```

A
MODULE DECLARATION
GENERATOR

by

JOSEPH LIBRERS

B. S., San Jose` State University, 1976

ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

ABSTRACT

The Module Declaration Generator is a software engineering tool for implementing PASCAL modules. It inputs an Entity Relationship Attribute (ERA) Requirements Specification Activity Block and a data dictionary repository file. It outputs a framework of an executable PASCAL module with the data declarations required to handle the modules' inputs and outputs. In addition, it includes the ERA Activity Block and the input and output variable descriptions in a PASCAL comment construct.

This Module Declaration Generator is written in C Language. It will execute in a UNIX* environment.

* UNIX is a trademark of AT&T Bell laboratories