303

PROLOG IMPLEMENTATION OF A GRAPHIC TOOL FOR GENERATION OF ADA LANGUAGE SPECIFICATIONS

by

SPENCER SHU-TSU CHENG

B.S., NATIONAL CHENG-KUNG UNIVERSITY, 1980
M.S., KANSAS STATE UNIVERSITY, 1986

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer and Information Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1987

Approved by :

[illegible]

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my major professor, Dr. William J. Hankley, for his guidance and direction in completion of this report.

Thanks are to my parents for their love, encouragement through my education.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Prolog

Prolog is recognized as an important language for A.I. [Bobrow, 1985]. Recently, Prolog has also received attention as a suitable language for rapid development of general software tools [Buhr, 1985] [Tavendale, 1985]. It provides declarative syntax for easy representation of knowledge and a powerful database search facility. The declarative nature of Prolog makes it a useful specification language which is also executable. Its dynamic and modular features facilitate incremental development of programs.

1.2 Purpose

The purpose of this project is to evaluate the suitability of Prolog in the implementation of a graphic tool for design and specification of Ada programs. The implementation utilizes the graphics and window facilities of Turbo Prolog for building diagrams and displaying text information. The global database is used for storing graphic information and building specifications. Evaluation of Prolog vs. Pascal code is made with respect to the implementation of this graphic tool.

1.3 Brief overview of the system

The function of this automated tool for specification is to provide software designers a mechanism to represent a system graphically while the specification of each component can be created and viewed interactively. The design of this tool is based upon a previously developed graphic model called GTGALS (a Graphic Tool for Generating Ada Language Specifications) [Bodle, 1985]. It allows graphic representation of a software system with boxes for components and directed arrows for their relationships. Naming and specifying procedures, inputs and outputs can be done interactively by using a prompt-response sequence. The graph of a software system together with its Ada specification are shown in Figure 1.1 and 1.2. The Ada language specifications are generated automatically from the graphic specifications. Both graphic and text specifications can be saved in the files on disk which can be retrieved for further refinement.

In Figure 1.1, Mod1 is an Ada program module; it is represented by a box in the graph. After the user places the cursor in a small window beside the box and presses "v", the Ada specification for Mod1 is displayed on top of the graph (Figure 1.2).

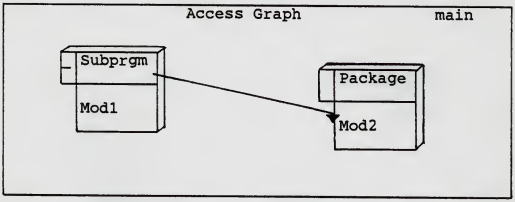
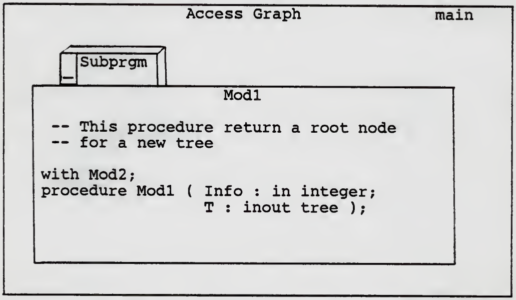


Figure 1.1 - Access graph for an Ada program



```
 -- This procedure return a root node
 -- for a new tree

with Mod2;
procedure Mod1 ( Info : in integer;
                 T : inout tree );
```

Figure 1.2 - Viewing Ada specification

The prototype is written in Turbo Prolog. Knowledge about the format and syntax of Ada specifications is stored in Prolog rules and facts. Text information about data names and types and comments is stored in a list data structure kept in the database section of Turbo Prolog.

1.4 Results

The results for this implementation project are the following:

(1) Code size comparison between Prolog and Pascal

Comparisons will be made between the Prolog version of the prototype implementation for this project and the Pascal version built by Bodle [Bodle, 1985]. The two versions provide very similar functionality. The object modules have roughly the same size, about 64K bytes. In Pascal, the object module is about the same size as the binary image of the program. In Prolog, 37K bytes of system code which provides build-in algorithms are linked to the object modules to make up a total of 101K bytes of binary image.

The analysis of the source codes for the two versions is shown in Table 1.1. The source code in Prolog has about 900 lines (32K bytes) compared to 2100 lines (70K bytes) in Pascal. In the Pascal version, an additional 1500 lines (45K bytes) of source code of Turbo Graphix Toolbox is included in the total space. In Prolog, the graphic and windowing capabilities are provided by the standard built-in precedures embedded in the Turbo Prolog system.

Smaller source code size for Prolog is mainly because of its embedded control flow and easy manipulation of data structures. The program body of Prolog is basically a

	Prolog	Pascal
Body	636	1769
Comments	167	113
Declarations	72	210
	892	2092
Turbo Graphix Toolbox System	0	1500
Total	892	3592

Table 1.1 - Analysis of Prolog vs. Pascal source codes (by number of lines)

series of procedure calls. Implicit control of sequence of statements is embedded in the object modules of Turbo Prolog system. In Pascal, variables for loop and branching structures must be used in specifying flow of control. The constructs for control, such as begin-end, if-then-else, while loop, and repeat-until, account for larger line counts in Pascal. In one program pieces in Pascal for drawing a graphic object, these key words for control account for 30% of the total lines.

Manipulation of data structures is made easy by Prolog's database features. Several variables can be compared at the same time through matching. For example, to search a graphic object in Prolog, we only need to declare a single predicate as the subgoal to retrieve a fact needed from the database; while in Pascal, the parameters of objects have to be compared, one at a time, from the beginning of an object array. We will further illustrate the different style between the two languages with some example code from the implementations in section 3.

(2) Run time information

When the program is executed under Turbo Prolog on a computer with 640K bytes of memory, the average size of the run space is 200K bytes for Turbo Prolog interpreter, 101K bytes for code, 64K bytes for stack, and 270K bytes for heap. The heap space is the remaining memory after the memory for stack, code, and trail areas in Turbo Prolog have been allocated. It is used as the database area for storing information about the objects created. Memory space needed for an object depends upon the complexity of the object, e.g. number of variables in a subprogram, and number of program units in a package. The memory heap space used for a few example objects created are shown in Table 1.2. With 270K bytes of heap space available and an average of 2.7K bytes allocated for a simple object, 100 is the maximum number of objects that can be created.

Type of object	Data space used
Packages	
no unit	1.4 K
no unit	1.7 K
one unit	5.0 K
one unit	7.0 K
two units	10.0 K
Subprograms	
procedure (no variable)	1.7 K
procedure (no variable)	2.0 K
function (no variable)	2.4 K
function (no variable)	2.7 K

Table 1.2 - Heap space used for example objects

(3) Programming style

In this section, we'll briefly compare a few pairs of corresponding pieces of source code taken from the Pascal and the Prolog versions. Each pair performs the same operations.

(i) Displaying help information

The system displays the functions of all the key commands in a window after the user presses the 'h' key.

```
procedure Help;

begin
  Move_cursor_out;
  StoreWindow(1);
  SelectWorld(4);
  SelectWindow(4);
  SetBackground(0);
  DefineHeader(4,'HELP INFORMATION');
  SetHeaderOn;
  DrawBorder;
  gotoxy(10,7);  writeln('DRAW COMMANDS');
  gotoxy(10,8);
  writeln('     a - defines origin and midpoints of',
         ' access arrows');
  gotoxy(10,9);
  writeln('     e - defines end-point of access arrows');
  gotoxy(10,10);
  Writeln('     p - draws package; s - draws subprogram');
  gotoxy(10,11);
  writeln('     gp - draws generic package;',
         '  gs - generic subprogram');
  gotoxy(10,12);
  writeln('     zi- zooms in on object selected by',
         ' cursor position');
  gotoxy(10,13);
  writeln('     zo- zooms out to parent diagram of',
         ' object selected');
  gotoxy(10,14);
  writeln('EDIT COMMANDS');
  gotoxy(10,15);
  writeln('     e - enters component specification',
         ' editing mode');
  gotoxy(10,16);
  writeln('    da - deletes access arrow originating at',
         ' the cursor');
  gotoxy(10,17);
  writeln('    do - deletes object selected by',
         ' cursor position');
  gotoxy(10,18);
  writeln('DISPLAY COMMANDS             ',
         '                       *************');
  gotoxy(10,19);
  writeln('     h - "HELP" describes',
         ' commands                *');
  gotoxy(10,20);
  writeln('     v - displays selected object',
         ' specification    *  \ ends pgm');
  gotoxy(10,24);
  writeln('Press any key to return to access graph');
  repeat until keypressed;
  gotoxy(1,24); writeln(' ':80);
  ClearScreen;
  RestoreWindow(1,0,0);
  Move_cursor_in;
end;  { Help }
```

PROLOG

```
action(104,X,Y,X,Y) :-
   removewindow,
   makewindow(1,7,7,"HELP INFORMATION",0,0,25,80),
   write("DRAW COMMANDS"),nl,
   write("   p -  create package"),nl,
   write("   s -  create subprogram"),nl,
   write("   gp - create generic package"),nl,
   write("   gs - create generic subprogram"),nl,
   write("   a -  make access connection from object"),nl,
   write("   d -  delete the object"),nl,
   write("   zi - zoom in on object"),nl,
   write("   zo - zoom out to parent diagram"),nl,
   write("INPUT OUTPUT COMMANDS"),nl,
   write("   r - input graph from file"),nl,
   write("        (only at the beginning)"),nl,
   write("   t - save text specification and"),nl,
   write("          graph on file and exit"),nl,
   write("DISPLAY COMMAND"),nl,
   write("   h- 'HELP' describes commands"),nl,
   write("   v- view the specification of the object"),
   readchar(_),
   removewindow,
   active(A),
   window(A,Wn,_),
   put_diagram(Wn,X,Y).
```

The code for display help information is similar in the two languages. With the windowing features of Turbo Prolog, the position for each line of text information doesn't have to be specified in the program. Two versions would have about the same size if the effects of graphic primitives are excluded.

(ii) Drawing a graphic object

A box is drawn on the access graph to represent a module in an Ada software system.

PASCAL

```
procedure Draw_object(which : char; x, y : real);

procedure Draw_std_object(x,y : real);
begin
  Move_cursor_out;
  DrawSquare(x-50,y-60,x+50,y+40,false);
  DrawSquare(x-50,y+40,x+50,y+80,false);
  Move_cursor_in;
end;  { Draw Std Object }

procedure Draw_generic(x, y : real);
begin
  Move_cursor_out;
  DrawLine(x-40,y-60,x+60,y-60);
  DrawLine(x+60,y-60,x+40,y+40);
  DrawLine(x+40,y+40,x-60,y+40);
  DrawLine(x-60,y+40,x-40,y-60);
  DrawLine(x-60,y+40,x-65,y+80);
  DrawLine(x-65,y+80,x+35,y+80);
  DrawLine(x+35,y+80,x+40,y+40);
  Move_cursor_in;
end;  { draw generic }

begin { draw object }
  case which of
    'g' : begin                { generic package }
            Draw_generic(x,y);
            Move_cursor_out;
            DrawTextW(x-38,y+53,1,'PACKAGE');
            Move_cursor_in;
          end;
    'h' : begin                { generic subprogram }
            Draw_generic(x,y);
            Move_cursor_out;
            DrawTextW(x-58,y+53,1,'SUBPROGRAM');
            Move_cursor_in;
          end;
    'p' : begin                { package }
            Draw_std_object(x,y);
            Move_cursor_out;
            DrawTextW(x-28,y+53,1,'PACKAGE');
            Move_cursor_in;
          end;
    's' : begin                { subprogram }
            Draw_std_object(x,y);
            Move_cursor_out;
            DrawTextW(x-45,y+53,1,'SUBPROGRAM');
            Move_cursor_in;
          end;
  end; { case }
end; { draw object }
```

PROLOG

```
/*  draws a box for an object labelled
    with  its  object type */

empty(G,X,Y,T) :-  /* G: generic or not */
   square(G,X,Y),  /* T: name of the type of the module */
   X1=X+1,Y1=Y+2,
   makewindow(2,7,0,"",X1,Y1,1,8),
   write(T),
   removewindow.

 square(G,X,Y) :-
   X1=1280*(X-1)+500,Y1=400*Y,
   A1=X1,B1=400*(Y-1),A2=1280*X+430,B2=B1,
   A3=A2,B3=Y1,
   X2=1280*(X+3)+500,Y2=Y1,
   X3=X1,Y3=400*(Y+9),
   X4=X2,Y4=Y3,
   X5=1280*(X+1)+500,Y5=Y1, /*   I1------------------I2   */
   X6=X5,Y6=Y3,             /*  /                  / ¦    */
   line(A1,B1,A2,B2,7),     /* A1----1------------3  ¦    */
   line(A2,B2,A3,B3,7),     /* ¦     ¦  window    ¦  ¦    */
   line(X1,Y1,X2,Y2,7),     /* ¦     5------------6  ¦    */
   line(A1,B1,X3,Y3,7),     /* A2---A3            ¦ I3    */
   line(X2,Y2,X4,Y4,7),     /*       ¦   Name     ¦ /     */
   line(X3,Y3,X4,Y4,7),     /*       2------------4       */
   line(X5,Y5,X6,Y6,7),     /*                            */
   G<>"generic",
   I1=1280*(X-1)+300,J1=Y1,
   I2=I1,J2=400*(Y+10),
   I3=1280*(X+2)+500,J3=J2,
   line(A1,B1,I1,J1,7),
   line(I1,J1,I2,J2,7),
   line(X3,Y3,I2,J2,7),
   line(I2,J2,I3,J3,7),
   line(X4,Y4,I3,J3,7).
square(_,_,_).
```

The styles for drawing graphic objects are different between the two versions. From the structures of the programs, the keywords for the control structures in Pascal, such as "begin", "end" , and "case", account for 30% of the line counts in Pascal code.

(iii) Creating comments

In the specification entry session, the system prompts the user to provide comments about the object created and stores them in a data structure.

PASCAL

```
procedure get_comments(var in_ptr : comment_ptr);

var current_com : comment_ptr;
    comment: comment_ptr;
    command: char;
    in_comment : string[60];

begin
  if line_no > 17 then
  begin
    for i := 11 to 20 do  { blank out information }
    begin
      gotoxy(10,i); writeln(' ':60);
    end;
    line_no := 11;
  end;
  gotoxy(10,line_no); writeln(' ':60);
  gotoxy(10,line_no);
  in_comment := '';
  writeln('Enter up to 58 characters of comment after',
          ' -- (or return)');
  line_no := line_no + 1;
  gotoxy(10,line_no); write('--'); readln(in_comment);
  line_no := line_no + 1;
  if in_comment <> '' then
  begin
    New(comment);
    comment^.line := '--' + in_comment;
    comment^.next := nil;
    current_com := comment;
    in_ptr := comment;
    repeat
      if line_no > 17 then
      begin
        for i := 11 to 20 do  { blank out information }
        begin
          gotoxy(10,i); writeln(' ':60);
        end;
        line_no := 11;
      end;
      gotoxy(10,line_no);
      write('--');
      in_comment := '';
      readln(in_comment);
      line_no := line_no + 1;
      if in_comment <> '' then
      begin
        New(comment);
        current_com^.next := comment;
        comment^.line := '--' + in_comment;
        comment^.next := nil;
        current_com := comment;
      end;
    until (in_comment = '');
  end; { if first comment <> '' }
end;   { get_comments }
```

PROLOG

```
takecomm(L) :-
      write("Enter comments after --"),nl,
      comment([],L).

comment(In,Out) :-
      write("  -- "),
      readln(Line),
      Line <> "",
      fronttoken(A," -- ",Line),
      appendtoken(In,[comm(A)],M),
      !,comment(M,Out).
comment(In,In).
```

In Pascal, each line of comments is stored in a linked list using pointers. Several assignments have to be made when a new line of comment is added. Additional variables are needed to control looping and positions for writing text. In Prolog, the process is considerably simpler. Each recursive call appends one line of comments into a list. The control for the execution is supplied by the compiler. The second clause for the predicate "comment" is used for the boudary condition to terminate the recursion.

(iv) Deleting an object

An object on the graph is deleted after the user places the cursor in an appropriate position and presses a key command.

PASCAL

```
procedure Delete;
  begin
     select(x,y,found,in_object,index);
     if found then
     begin                     { if found }
        gotoxy(1,24); writeln(' ':80);
        SetColorBlack;
        Draw_object(in_object,object[index].point.x,
                    object[index].point.y);
        Draw_name(object[index].point.x,
                  object[index].point.y,
                  object[index].name);
        Erase_arrow(in_object, index);
        SetColorWhite;
          Init_object(index);
        SetColorWhite;
        end; { if found }
  end;   { end delete object }
```

PROLOG

```
 /* "d" key to delete an object */
action(100,X,Y,X,Y) :-
    active(A),
    retract(obdb(A,' ',_,_,X,Y,_,_,_)),
    retract(window(A,Wn,C)),
    retract(total(Tw,To)),
    C1=C-1,To1=To-1,
    asserta(window(A,Wn,C1)),
    assert(total(Tw,To1)),
    put_diagram(Wn,X,Y).
```

In Prolog, it is easy to access the data structure from the database. After the fact which stores all the information about the object is retrieved from the dynamic database of Turbo Prolog, the deletion is made by "retracting" the fact from the database.

(4) Turbo Prolog features

The combination of graphic and window features in Turbo Prolog facilitates the design of this graphic tool. However, there are a few things not fully supported by Turbo Prolog. The graphic mode of Turbo Prolog doesn't provide a blinking cursor. Cursor movement has to be simulated by drawing and erasing a short line. Since the content of a window can not be saved, the graphic objects have to be redrawn whenever the window is overlapped by another window. Windows have to be created in order to write characters for labelling graphic objects.

Recursive procedure calls are frequently used in this implementation. It would be very easy to run out of stack space during run-time if predicates have parameters with large data structures. The problem was alleviated by putting the parameters in the facts of the dynamic database and modifying them successively during recursive calls. This approach is utilized for constructing token list and text specification for each component.

1.5 Organization

Background information on feasibility of Prolog for software system design is provided in Chapter 2. It presents programming features about Prolog and a summary of previous Prolog implementations of graphic projects and design tools for Ada systems.

The focus of Chapter 3 is Ada language constructs to support effective software design. Programming units related to the implemented project are discussed.

Chapter 4 describes the graphic tool called GTGALS (a Graphic Tool for Generation of Ada Language Specifications). Implementation details using Prolog rule predicates and database facilities are described in this chapter.

Chapter 2

Prolog as a system design tool

2.1 History

Prolog was invented in 1972 by Alain Colmerauer and his associates at University of Marseilles. It was intended as a language for specifying programming tasks in logic. It didn't attract widespread interest until David Warren's efficient implementation of Edinburgh Prolog in 1979 on a DEC-10 computer. Since then, Prolog has been applied as a tool for natural-language processing, expert systems, and logical problem solving. Other areas of application include planning, design automation, solving symbolic equations, biochemical structure analysis, drug design, and architectural design.

The announcement of the Japanese fifth generation computer system project has drawn world-wide attention. Their commitment to Prolog as the basis for building intelligent knowledge-based computer programs has stimulated study of logic programming and Prolog. Prolog is expected to have more application in various fields of computer science.

2.2 Features

Prolog is fundamentally different from the more conventional languages, such as FORTRAN and Pascal, in which operations are expressed in an imperative way. In Prolog, details of a program is described in a declarative style and the language uses a built-in procedure for resolving goals and instantiating parameters. In Prolog, the program is defined as a set of facts and rules; a problem is represented in a logical structure described by these facts and rules. Execution of the program is carried out by attempting to satisfy goals using the facts and rules. The flow of control is maintained by the Prolog system through its unification and backtracking algorithms.

An inherent feature of Prolog is its deductive database. It stores all the facts and rules (clauses) composing the program body. When a goal is given, Prolog reponds by searching through all the clauses sequentially. When a match is found, any preconditions (subgoals) of the matched clause are treated as new goals, that Prolog has to solve to provide the solution for the matched condition. Once a solution is found, Prolog backtracks and keeps going through the entries until it has exhausted all possibilities of matches.

In a general Prolog interpreter system, facts and rules can be added and deleted from the database dynamically during run time. The Prolog built-in predicates that add and delete facts and rules are "assert" and "retract", respectively. In Turbo Prolog, however, only the facts for the predicates declared in a separate database section of Turbo Prolog can be updated dynamically during execution. No rule can be added or deleted during execution. The clauses for the normal predicates that are not declared in the database section are static after the compilation. In the implementation of this report, the predicates in the dynamic database section are mainly for storing graphic information and the specification for each software component. In the context of this report, the word "database" will refer specifically to the dynamic database section of Turbo Prolog.

2.3 Survey of applications

Prolog has received increasing attention as a powerful language for artificial intelligence research because it is well suited for knowledge representation [Subrahmanyam, 1985] and natural-language processing [Cuadrado, 1985]. Recently, it has also been used for geometry projects [Franklin 1986], CAD applications as well as for system design tools. Comparison between Prolog and Pascal in a graphic project [Gonzalez, 1984] has shown that Prolog is not only more concise, more readable, and clearer than

Pascal, but also more efficient. Furthermore, Prolog programs were developed more quickly and with less errors. In experiments with a Prolog tool for Ada system design environment [Buhr, 1985], its expressive power provided a means of developing a powerful, rule-based tool to enhance the flexibility and extensibility of the design environment. In Buhr's work, Ada code skeletons were generated from the Prolog facts in the design database. The code was generated by attempting to satisfy the language syntax rules. When the execution of the program encountered rules containing goals or subgoals that must be matched against the Prolog facts in the design database, the facts were then used to generate code. These facts are also the representations of lists of components, their properties and their relationships.

Chapter 3

Software development with Ada

3.1 History

Ada is a programming language developed by the U.S. Department of Defense in response to part of the escalating software cost problems faced in many of the large military software systems. Many systems developed by conventional languages were found to be unreliable, hard to maintain and lacking portability. It was believed that the problems were caused by the fact that these languages did not support good software practices [Wiener, 1984]. Ada was designed to reduce software life cycle cost and enforce software engineering principles and methodologies.

3.2 Ada's support for modular design

Software design is also considered a process of abstraction. Solving programming problems involves mapping the entities in problem domain to the structures available in a language. In conventional languages, we are often preoccupied with details of translating problems into predefined data and control structures. Ada allows programmer-defined abstract data types and functional abstractions for the operations on data types so that real world problems can be mapped naturally into programmer-created abstractions.

Ada systems are collection of program units including subprograms, packages and tasks. A subprogram defines a simple operation or action. A package collects a group of logically related entities including data types, data objects, subprograms, tasks and other packages. It encourages the development of reusable software modules for storage in the libraries. A task defines parallel actions.

Separate compilation of specification and information hiding in the body facilitate modular software construction in Ada. The compiler would enforces the interfaces during the development of the system. Each unit has a specification part that is separate from its implementation body. The specification is the visible part that defines how the unit interfaces with the other units. For example, the specification of a package has a description of accessible parts of the package and an indication of how they can be used. The body is the hidden part that contains the implementation details of the resources indicated in the specification. The specification part must be compiled before the implementation part. One can create specifications early in the design process and have them compiled and placed in a library. The bodies of units can be developed later. Other members of a team would then have access to the interface as they develop their units.

Generic program units are a powerful Ada construct. Generic units permit software developers to create subprgrams or packages to operate on data objects of different types. Therefore, an algorithm can be defined as a template of program units and then tailored to particular needs at translation time.

3.3 Graphic tools for Ada system design

The work in this report was based on GTGALS [Bodle, 1985] which was designed for specifying separately compilable Ada language units. Bodle's model is based on the design of Buhr [Buhr, 1984] for graphic development of Ada systems.

In Buhr's design, different shapes of boxes are drawn for different types of components, such as packages and tasks, and an arrow is drawn from the user box to the accessed box. There are sockets in a box to represent the interface parts of a component with other components. In packages, sockets may serve for procedural or non-procedural calls. In tasks, they may serve for entry calls during rendezvous mechanism. In this report and Bodle's version, only packages and subprograms are included as the components in the prototype version. Treatment of concurrency is not included.

Chapter 4

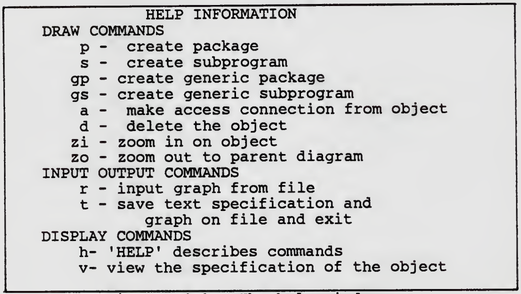
IMPLEMENTATION

This chapter describes the Prolog implementation of the prototype of a graphic tool for generating Ada language specifications. The program was done on a IBM-XT compatible micro-computer. It was written in Turbo Prolog with about 900 lines of source code and compiled to 64K bytes of object code. It runs under the graphic mode of Turbo Prolog with overlayed windows for diagrams and text displays. The output of the program are files with extension .ada and .gph (The filename is supplied by the user at the end). The .ada file is the Ada language specification of the developed graphs and the .gph file contains the facts asserted in the database section.

4.1 Overview of the prototype

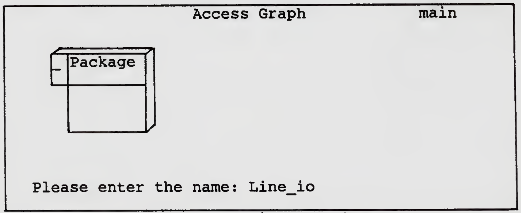
An access graph is used for pictorially describing software systems. The components of a system and the interfaces among the components can be easily represented on the graph to provide the designer a nice overview of the software system. Boxes are drawn as the symbols for the components and an arrow from a point in a box to another box indicates an access connection from one component to another component.

The user first enters a main diagram with a cursor in the middle of the screen. Help information can be retrieved from the diagram by pressing the "h" key. This will bring up a help window containing the commands and a brief description of what each key command does (Figure 4.1). An object is created by moving the cursor to the desired screen location and pressing a key command for the desired object. A box is drawn for the object, then the user is prompted for the object name (Figure 4.2).



```
                HELP INFORMATION
    DRAW COMMANDS
        p -  create package
        s -  create subprogram
       gp - create generic package
       gs - create generic subprogram
        a -  make access connection from object
        d -  delete the object
       zi - zoom in on object
       zo - zoom out to parent diagram
    INPUT OUTPUT COMMANDS
        r - input graph from file
        t - save text specification and
               graph on file and exit
    DISPLAY COMMANDS
        h- 'HELP' describes commands
        v- view the specification of the object
```

Figure 4.1 - The help window



```
Please enter the name: Line_io
```

Figure 4.2 - Creating a package

A specification entry window is then placed on top of the diagram. The interactive prompt-response sequence in the entry window then prompts the user to provide information for each object. For example, for procedures, the user is prompted for procedure names, and input and output variable names. The user can provide comments for an entire component as well as for each procedure or function within the component (Figure 4.3).

```
Specification entry for component Line_io

 Enter comments after -- (return to skip)
 -- This package takes care of a system's
 -- interaction with the physical terminal
 --
```

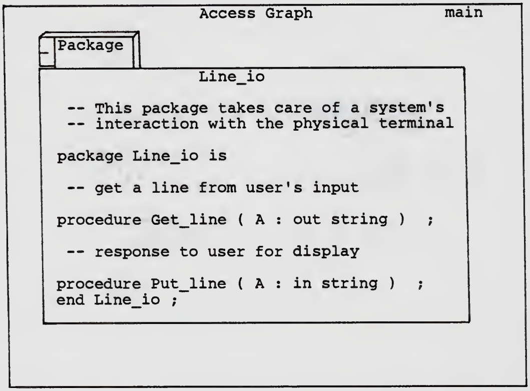
(a) Editing comments

```
Specification entry for component Line_io
Procedure or Function (p or f)? p
  name: Get_line

Enter comment after --
-- get a line from user's input
--

in:

out: A
type: string
```

(b) Specifying a procedure

Figure 4.3 - Specification entry for an object

After completion of the prompt-response sequence, the top window for the entry is removed. The screen goes back to the diagram with a new box representing the object. A small window called an activation window is attached to the box. The cursor has to be placed into this window before the user presses any key command about the object. Direct access to a component's specification is done by moving cursor to the activation window of the component and pressing "v". The system then creates a window and displays the Ada language specification for the component (Figure 4.4).



```
 -- This package takes care of a system's
 -- interaction with the physical terminal

package Line_io is

 -- get a line from user's input

procedure Get_line ( A : out string )  ;

 -- response to user for display

procedure Put_line ( A : in string )  ;
end Line_io ;
```

Figure 4.4 - Viewing text specification

Making access connection between two objects is started by pressing "a" key in the activation window of the first object. The next step is to specify the starting point of the arrow by moving cursor to any point on the first object and pressing "b". Then, if necessary, the user can establish intermediate points on the arrow by pressing "t" at each intermediate point. The arrow is completed by pressing "e" at the end point (Figure 4.5). Another "a" has to be pressed in the activation window of the second object in order to finish the access action. The name of the second object will be included in the with clause of the first object. The user can witness such an access action by viewing the modified text specification of the first object.

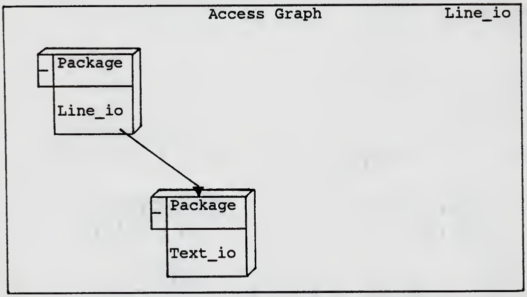


Figure 4.5 - Access connection between components

4.2 Logic and data structure

The implementation of this model utilizes the graphic and windowing features offered by Turbo Prolog. Under the graphic mode, lines can be drawn for making up different boxes and arrows. Windows can be created for writing text, getting help information, editing specification and shifting among different diagrams. A visual cursor is provided for moving around the graph to the desired location. Cursor movement is simulated by repeatedly drawing a short line in each new position and erasing the line in the old position. Erasure is done by drawing the line in the color of background. The side effect of such a cursor design is that part of the graph or characters may be erased when cursor goes through them. The design has been taken to avoid cursor passing through parallel lines. The user can press the "n" key to redraw the current diagram. The cursor moves as a text cursor, e.g. row number from 1 to 25 and column number from 1 to 80 for specifying the cursor position.

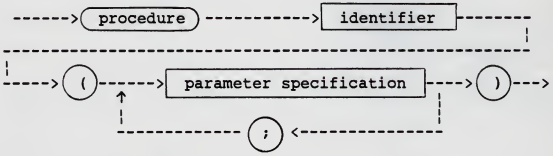
Different shapes of boxes are drawn to distinguish generic from non-generic components. Non-generic packages and subprograms are represented by projections of three dimensional boxes. Generic packages and subprograms are represented by rectangles. A small activation window is attached to the upper left edge of each box and rectangle. When a command is to be selected, the cursor has to be placed into the small window of the corresponding object.

Top-down and stepwise refinement is the recommended method of development using this graphic tool. A typical design starts with a main diagram for the user to lay out the major components of a software system. After an object is drawn, a specification entry window is created on top of the current window and the user is prompted for information needed for composing the specification for the object. The prompt-response sequence in the specification entry window follows the syntax of the specification for the object.

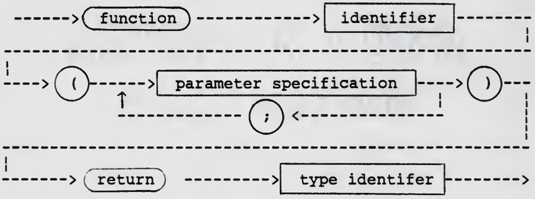
The syntax diagrams for Ada package and subprogram specifications are shown below. In the syntax diagrams, items in rounded boxes or circles are terminal symbols; items in rectangles are non-terminals that need to be further defined.

(1) Subprogram specification

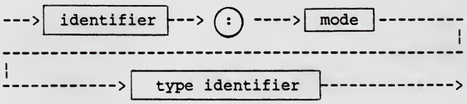
(i) for procedure



(ii) for function

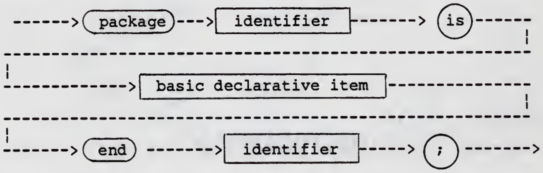


where the parameter specification is



mode is one of in, out or inout.

(2) Package specification



where basic declarative item is subprogram specification that may be repeated zero or more times.

When the information is provided by the user, it is stored as a list of tokens with the following Prolog structures. (note: each name is a functor and each single letter is a token list)

(1) for procedure

```
[comments(C),access(A),generic(G),title("procedure"),
 name(N),parameter(P)]
```

(2) for function

```
[comments(C),access(A),generic(G),title("function"),
 name(N),parameter(P),return(R)]
```

(3) for package

```
[comments(C),access(A),generic(G),title("package"),
 name(N),unit(U)]
```

C : a list of comm(string)

A : a list of access object names

G : either "generic" or "" (empty string)

N : object name

P : a list of var(M,V,T) where M is one of "in", "out" or "inout", V is parameter name, and T is parameter type

U : a list of token lists for subprograms

For example, the text specifications and the corresponding token list for an Ada's procedure are shown below :

<u>Text Specifications</u>

```
 -- This is the main program for control
with Process ,
     Counter ,
     Line_io ;
procedure Main1 ( in_text : in code ;
                 out_text : out code )  ;
```

<u>Token List</u>

```
[comms([comm(" -- This is the main program for control")]),
 access([with("Process"),with("Counter"),with("Line_io)]),
 g(""),title("procedure "),name("Main"),
 para([var("in","in_text","code"),
       var("out","out_text","code")])]
```

After the prompt-response sequence for an object is completed, the corresponding token list is then used for building text specifications. The specification entry window is removed and the screen goes back to the diagram with the new object. Both token list and text specification of the object are stored in the database for final output or further refinement.

The main control relies on a predicate called "readkey". The single clause for this predicate is :

```
/* X and Y are input row and column numbers for cursor
   position, respectively. They are initialized  by
   the system at X=10 and Y=20. X1 and Y1 are for the
   new cursor position after action   */

 readkey(X,Y) :- readchar(C), char_int(C,I),
                 action(I,X,Y,X1,Y1),
                 !, readkey(X1,Y1).
```

X and Y are the current vertical and horizontal coordinates for the visual cursor, respectively. The cursor is placed at the center of the screen initially. When the user presses a key command, the key pressed is then converted to its corresponding ASCII integer value by the predicate "char_int". The predicate "action" does action corresponding to the key pressed. They will be explained in the following section. The new coordinates X1 and Y1 are returned after the action clause is satisfied. Finally, a recursive call is made by having "readkey" as the last subgoal with the new coordinates as the new input parameters.

Here is a brief look of how each key command is implemented :

(1) Creating objects ("p", "s", "gp", or "gs" keys)

After a box is drawn for the object created, the information about the object provided by the user is stored in a database predicate with the following structure :

```
obdb(A,M,G,S,X,Y,N,Tok,Spec)
```

The arguments for the predicate are :

A: window number (diagram number)

M: a flag to indicate whether the object is in a zoomed-in diagram

G: generic or not - a control for the shape of the box

S: type of object (subprogram, package, generic subprogram or generic package)

X,Y: coordinates of the box in the diagram

N: name of the object

Tok: token list (a list of items needed for building the specification)

Spec: text specification of the object

(2) Viewing specifications ("v" key)

The cursor must be in the activation window beside the box before the "v" key is pressed for viewing the specification. Otherwise, no action can be done on the the object. The location of the cursor (X and Y) and currently active window number are used as the indexes for searching facts in the database. The clause for viewing looks like :

```
/* Ascii value for "v" = 118 */
   action(118,X,Y) :-
        active(A),  /* currently active window number */
        obdb(A,_,_,_,X,Y,_,_,Spec), ............
```

By using the unification algorithm, the fact in the database is selected with matching values for the known parameters A, X and Y. The parameter "Spec" is instantiated from the matched fact; it is used to display text specification for the object.

(3) Deleting objects ("d" key)

The algorithm for deleting objects is very similar to that for viewing specification. The database fact for the object to be deleted is removed by using the standard predicate "retract".

(4) Zooming in on objects and out to a parent diagram
("zi" or "zo" keys)

Any component in a diagram can be decomposed by zooming in on the component. This will cause the object to be represented in two different windows, one in the original window and one in the expanded window. In this case, two facts for the same object will be present in the database with different window numbers. After the fact for the object is found in the database, the second fact for the object is created if the object has never been zoomed in. A mark '*' is placed on the second fact to indicate that the fact is for a lower level diagram. The user can then specify a new diagram for the component. If an object to be expanded has previously been expanded, then the window number is retrieved from the existing second fact for the object. A new diagram is then shown by drawing every object with the new window number. Zooming-out is done by simply selecting the fact for the object in the parent diagram and redrawing the parent diagram. One fact for the predicate "active" is present in the database to keep track of the currently active window number.

(5) Making access connection ("a" key)

The clause for making access connection looks like :

```
    /* Ascii value for "a" = 97 */
  action(97,X,Y,X,Y) :-
    active(A), .............
    retract(obdb(A,M,G,S,X,Y,N,[C,access(W)|R],_)),
    start_connect(X,Y),
    retract(access_name(An)),
    assert(obdb(A,M,G,S,X,Y,N,[C,access(W,with(An)|R],_)),
    ...........................
```

After an "a" key is pressed with the cursor inside the activation window of an object, the token list in the database fact for the object is retrieved for modification. The predicate "start_connect" does the arrow drawing between the accessing object and the accessed object, and stores the name of the accessed object in a database predicate called "access_name". The name in the predicate "access_name" is then appended to the element "access" in the token list of the accessing object. The text specification is then reconstructed according to the new token list.

4.3 Sample design and output

This section shows an example of software design using this graphic tool and its output Ada language specifications. The access graph (Figure 4.6) for a text analyzer system is shown below as the example design. Process, Counter and Line_io are the components under control of a main program.

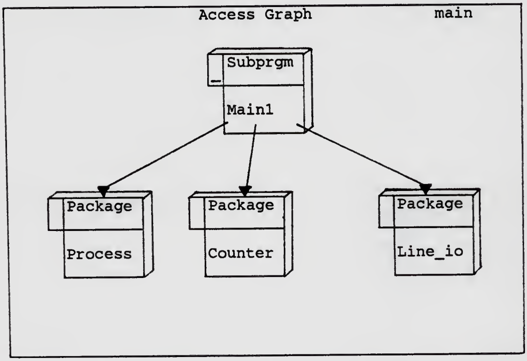


Figure 4.6 - Access graph for a software system

After zooming in on the component Line_io, a new graph for the decomposition of Line_io is drawn (Figure 4.7). The output Ada language specifications (Figure 4.8) are created based on the data entered during the design session for each component.

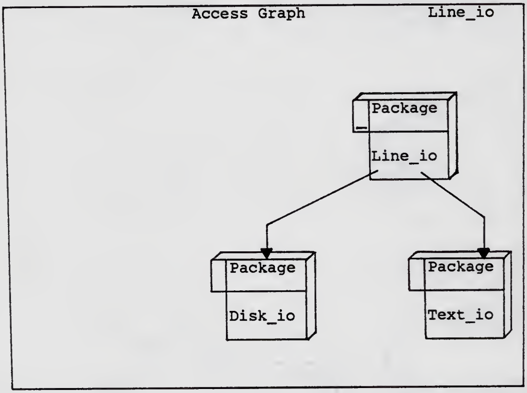


Figure 4.7 Decomposition of Line_io from Main1

```
 -- This is the main program for control
with Process ,
     Counter ,
     Line_io ;
procedure Main1 ( in_text : in code ;
                  out_text : out code )  ;

 -- This package is for analyzing text information
package Process is
 -- This procedure breaks the text into
 -- different components
procedure Break_text ( in_text : in code ;
                  out_char : out character ;
                  out_int : out integer ;
                  out_symbol : out symbol ;
                  out_string : out string )  ;
 -- This function returns the ascii value
 -- of a input character
function Ascii ( in_char : in character )  return integer ;
end Process;


 -- This package is to update counters
package Counter is
 -- Used to increment a counter by one
procedure Increment ( C : inout counter )  ;
end Counter ;

 -- This package takes care of a system's
 -- interaction with the physical terminal
with Disk_io ,
     Text_io ;
package Line_io is
 -- get a line from user's input
procedure Get_line ( A : out string )  ;
 -- respond to user for display
procedure Put_line ( A : in string )  ;
end Line_io ;


 -- This package handles disk i/o
package Disk_io is
end Disk_io ;

 -- This is a predefined library program
 -- for textual information i/o
package Text_io is
end Text_io ;
```

Figure 4.8 - Ada language specification for Main1

Chapter 5

Conclusions

The main feature of the graphic tools for specification are their user interfaces. The software designers can graphically lay out the components of a system with diagrams, and at the same time, manipulate the text specifications of each component. The prototype implementation in this project is a simple demonstration of the desired features and usefulness of such graphic tools. Expansion of the project can be done by additions of Prolog rules.

The source code of the Prolog version for this prototype implementation is relatively shorter than that of the Pascal version. This is due to embedded control flow in Prolog and also different types of graphic and window primitives between the two languages. The unification and pattern matching in Prolog make it easy to access and operate on the data structures.

The extensions of this project would be the additions of Ada contructs, such as tasks, type and private type declarations, and nesting of packages, to the created objects. The specification editor can also be refined to provide greated flexibility for the modification of each component's data structure. It is also desirable to transport such a system to a workstation with more memory space and graphic capabilities.

REFERENCES

Bobrow, D.G. (1985). "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November, 1985, pp. 1401-1408.

Bodle, D. (1985). "A Graphic Tool for Generating Ada Language Specifications," A Master's Thesis, Kansas State University, 1985.

Buhr, R.J.A., Karam, G.M., Woodside, C.M. (1985). "An Overview and Example of Application of CAEDE: A New, Experimental Design Environment for Ada," ADA Letters, September, 1985, pp. 173-184.

Buhr, R.J.A. et al. (1985). "Experiments with Prolog Design Descriptions and Tools in CAEDE: An Iconic Design Environment for Multitasking Embedded Systems," in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington D.C., 1985, pp.62-67.

Buhr, R.J.A. (1984). System Design with Ada, Englewood Cliffs, N.J.: Prentice-Hall Inc., 1984.

Cuadrado, C.Y., Cuadrado, J.L. (1985). "Prolog goes to work", Byte, Vol. 10, No. 9, August, 1985, pp. 151-158.

Franklin, W.R., et al. (1986). "Prolog and Geometry Projects", IEEE CG&A, Vol. 6, No. 11, November, 1986, pp. 46-55.

Gonzalez, J.C., Williams, M.H., Aitchison, I.E. (1984). "Evaluation of the Effectiveness of Prolog for a CAD Application", IEEE CG&A, Vol. 4, No. 3, March, 1984, pp. 67-75.

Subrahmanyam, P.A. (1985) "The Software Engineering of Expert Systems: Is Prolog Appropriate?", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November, 1985, pp. 1391-1400.

Tavendale, R.D. (1985). "A Technique for Prototyping Directly from a Specification", in Proceeding 8th International Conference on Software Engineering, Computer Society Press, Washington D.C., pp.224-229.

Wiener, R., Sincovec, R. (1984). Software Engineering with Modula-2 and Ada, John Wiley & Sons, Inc., New York, 1984.

Appendix 1

Prolog code for cursor control and key commands

```
Code = 3072
include "main2.pro"

predicates
/* following predicates are for cursor and key action */
  /* coordinates (X --> row number 1 -> 20,
                  Y --> column number 1 -> 69) */

         readkey(integer,integer)

  /* corresponding action after key command is given,
      (ascii value for key, input X, Y coordinates, and
       return X,Y coordinates after action )  */

         action(integer;integer,integer,integer,integer)

  /* action after 'g' or 'z' key is depressed,
   char follows 'g'or 'z'; X,  Y are row and column # */

         g_key(char,integer,integer)
         z_key(char,integer,integer)

/* restore cursor at row and column # */

         recursor(integer,integer)

/* redraw a diagram */

         put_diagram(string,integer,integer)

/*  draw all the objects in an active window  */

         redraw

/* draw one box at a time until the value of the
    argument reaches 0 */

         draw_object(integer)

/* draw all the lines for an active window */

         draw_all_lines
```

```
/* draw a list of line until empty, each element
    in the list has the coordinates for endpoints
    of a line */

         drawline(tokenlist)

/* write spec of all objects to file, argument is
   the number of objects that has been taken care of,
   stops when it reaches zero */

         write_all_spec(integer)

/* write spec (a list of string) to output */

         writelist(spec)

/* following predicates are for access connection */
  /* read key for making connection, arguments are row and
      column number of the cursor */

         start_connect(integer,integer)

  /* moving cursor for making access connection and
      drawing lines, arguments are key read in
      and X,Y coordinates*/

         move(integer,integer,integer)

  /*   for   reconstructing  spec  after   making
       access connection argument is the token list
       of the object */

         re_spec(tokenlist)

  /* makeline with coordinates of two points */

         makeline(integer,integer,integer,integer)

  /*  draw  arrowhead at row and column number X and Y  */

         arrowhead(integer,integer)
goal
    graphics(2,1,7),
    makewindow(1,7,7,"Access Graph         main",
               0,0,25,80),
    line(12800,7600,12800,8000,7),
    assert(active(1)),
    asserta(window(1,"main",0)),
    asserta(total(1,0)),
    asserta(linedb(1,[])),
    readkey(10,20).
```

```
clauses

/* write a list of string (specification)
     to output file or screen */
     writelist([]).
     writelist([Head¦Tail]) :-
     write(Head),nl,writelist(Tail).

/* draw all the objects in the active window */
     redraw :-
       active(A),
       window(A,_,C),
       draw_object(C),
       draw_all_lines.

/* draw one object and bump down counter */
     draw_object(C) :-
       active(A),
       C>0,
       retract(obdb(A,M,G,S,X,Y,N,T,L)),
       draw(G,S,X,Y,N),
       assertz(obdb(A,M,G,S,X,Y,N,T,L)),
       C1=C-1,
       !,draw_object(C1).
       draw_object(0).

/* get lines for the active window and draw them */
     draw_all_lines :-
       active(A),
       linedb(A,L),
       drawline(L).

/* draw a list of lines */
     drawline([]).
       drawline([li(X1,Y1,X2,Y2)¦T]) :-
       makeline(X1,Y1,X2,Y2),
       drawline(T).

/* write text spec of all the components into file */
     write_all_spec(C) :-
       C > 0,
       retract(obdb(A,' ',G,S,X,Y,N,Tok,Spec)),
       writelist(Spec),
       assertz(obdb(A,' ',G,S,X,Y,N,Tok,Spec)),
       C1=C-1,
       !,write_all_spec(C1).
     write_all_spec(0).

/* read character for corresponding action */
     readkey(X,Y) :-
       readchar(C),
       char_int(C,Val),
       action(Val,X,Y,X1,Y1),
       !,readkey(X1,Y1).
```

```
/* restore cursor */
     recursor(X,Y) :-
       X1=1280*X,
       Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,7).


     put_diagram(Wn,X,Y) :-
       fronttoken(Header,"Access   Graph                    ",Wn),
       makewindow(1,7,7,Header,0,0,25,80),
       redraw,
       recursor(X,Y).

/***** CURSOR MOVEMENT AND KEY ACTIONS *******/

/* '#' key to exit from program */
     action(35,X,Y,X,Y) :- exit.

/* "p" key for a package */
     action(112,X,Y,X,Y) :-
       S="package",
       package("",S,X,Y),
       active(A),
       retract(window(A,Wn,C)),
       retract(total(Tw,To)),
       C1=C+1,To1=To+1,
       asserta(window(A,Wn,C1)),
       assert(total(Tw,To1)),
       put_diagram(Wn,X,Y).

/* "s" key for a subprogram */
     action(115,X,Y,X,Y) :-
       S="subprgm",
       subprgm("",S,X,Y),
       active(A),
       retract(window(A,Wn,C)),
       retract(total(Tw,To)),
       C1=C+1,To1=To+1,
       asserta(window(A,Wn,C1)),
       assert(total(Tw,To1)),
       put_diagram(Wn,X,Y).


/* "d" key to delete object */
     action(100,X,Y,X,Y) :-
       active(A),
       retract(obdb(A,' ',_,_,X,Y,_,_,_)),
       retract(window(A,Wn,C)),
       retract(total(Tw,To)),
       C1=C-1,To1=To-1,
       asserta(window(A,Wn,C1)),
       assert(total(Tw,To1)),
       removewindow,
       put_diagram(Wn,X,Y).
```

```
/*  "v"  key  to  view  specification  of  the  object  */
     action(118,X,Y,X,Y) :-
       active(A),
       window(A,Wn,_),
       obdb(A,_,_,_,X,Y,N,_,S),
       removewindow,
       makewindow(2,7,7,N,3,10,18,60),
       writelist(S),
       readchar(_),
       removewindow,
       put_diagram(Wn,X,Y).


/* "h" key for help information */
     action(104,X,Y,X,Y) :-
       removewindow,
       makewindow(1,7,7,"HELP INFORMATION",0,0,25,80),
       write("DRAW COMMANDS"),nl,
       write("   p -  create package"),nl,
       write("   s -  create subprogram"),nl,
       write("   gp - create generic package"),nl,
       write("   gs - create generic subprogram"),nl,
       write("    a - make access connection from
                    object"),nl,
       write("   d -  delete the object"),nl,
       write("   zi - zoom in on object"),nl,
       write("   zo - zoom out to parent diagram"),nl,
       write("INPUT OUTPUT COMMANDS"),nl,
       write("   r - input graph from file"),nl,
       write("        (only at the beginning)"),nl,
       write("   t - save text specification and"),nl,
       write("          graph on file and exit"),nl,
       write("DISPLAY COMMAND"),nl,
       write("   h- 'HELP' describes commands"),nl,
       write("   v- view the specification of the object"),
       readchar(_),
       removewindow,
       active(A),
       window(A,Wn,_),
       put_diagram(Wn,X,Y).

/* "r" key, to read from a file */
     action(114,X,Y,X,Y) :-
       makewindow(2,7,7,"",21,0,3,80),
       write("Please enter the name of the file : "),
       readln(F1),
       fronttoken(F,F1,".gph"),
       removewindow,
       retract(window(1,"main",0)),
       retract(total(1,0)),
       retract(linedb(1,[])),
       consult(F),
       redraw,
       recursor(X,Y).
```

```
/* "t" to exit and save text specification on files */
     action(116,X,Y,X,Y) :-
       makewindow(1,7,7,"",21,0,3,80),
       write("Please enter the name of the file : "),
       readln(F),
       removewindow,
       fronttoken(F1,F,".ada"),
       openwrite(outfile,F1),
       writedevice(outfile),
       total(_,T),
       write_all_spec(T),
       closefile(outfile),
       fronttoken(F2,F,".gph"),
       retract(active(_)),
       save(F2),
       exit.

/* Arrow key */
     action(0,X,Y,X1,Y1) :-
       readchar(T),
       char_int(T,Val),
       action(Val,X,Y,X1,Y1).

/* Down */
     action(80,20,Y,20,Y) :- !.
     action(80,X,Y,A,Y) :-
       !,X1=1280*X,
       Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       X2=1280*(X+1),
       line(X2,Y1,X2,Y2,7),
       A=X+1.

/* Up */
     action(72,1,Y,1,Y) :- !.
     action(72,X,Y,A,Y) :-
       !,X1=1280*X,
       Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       X2=1280*(X-1),
       line(X2,Y1,X2,Y2,7),
       A=X-1.

/* Right */
     action(77,X,69,X,69) :- !.
     action(77,X,Y,X,A) :-
       !,X1=1280*X,
       Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       Y3=400*Y,Y4=400*(Y+1),
       line(X1,Y3,X1,Y4,7),
       A=Y+1.
```

```
/* Left */
     action(75,X,1,X,1) :- !.
     action(75,X,Y,X,A) :-
       !,X1=1280*X,
       Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       Y3=400*(Y-2),Y4=400*(Y-1),
       line(X1,Y3,X1,Y4,7),
       A=Y-1.

/* 'g' for generic object */
     action(103,X,Y,X,Y) :-
       readchar(C),g_key(C,X,Y).

/* 'z' for zooming in and out */
     action(122,X,Y,X,Y) :-
       readchar(C),z_key(C,X,Y).

/* 'a' for accessing another object */
     action(97,X,Y,X,Y) :-
       active(A),
       retract(obdb(A,M,G,S,X,Y,N,[C,access(W)|R],_)),
       makewindow(2,7,7,"",19,0,5,80),
       write("Press 'b'- starting point 't'- midpoint"),nl,
       write("       'e'-end point   then "),nl,
       write(" 'a'- in the window of accessed object"),
       shiftwindow(1),
       start_connect(X,Y),
       retract(access_name(An)),
       appendtoken(W,[with(An)],New),
       assert(tokendb([C,access(New)|R])),
       assert(specdb([])),
       re_spec(R),
       retract(tokendb(Tok)),
       retract(specdb(Spec)),
       assertz(obdb(A,M,G,S,X,Y,N,Tok,Spec)),
       retract(obdb(B,Sec,G,S,X,Y,N,_,_)),
       assertz(obdb(B,Sec,G,S,X,Y,N,Tok,Spec)),
       shiftwindow(2),clearwindow,
       removewindow,
       redraw,
       recursor(X,Y).

/* 'n' to redraw the screen */
     action(110,X,Y,X,Y) :-
       redraw.

/* other keys */
     action(_,X,Y,X,Y) :- !.

/* cases for reconstructing specification */
     re_spec([_,_,_,_,ret(_)]) :-
       func_spec.
     re_spec([_,_,_,para(_)]) :-
       proc_spec.
     re_spec([_,_,_,unit(_)]) :-
       pack_spec.

/* cases after "g" has been pressed */
/* 'p' for creating a generic package */
     g_key('p',X,Y) :-
       S="g.pack",
       package("generic",S,X,Y),
       active(A),
       retract(window(A,Wn,C)),
       retract(total(Tw,To)),
       C1=C+1,To1=To+1,
       asserta(window(A,Wn,C1)),
       assert(total(Tw,To1)),
       put_diagram(Wn,X,Y).

/* 's' for creating a subprogram */
     g_key('s',X,Y) :-
       S="g.subpm",
       subprgm("generic",S,X,Y),
       active(A),
       retract(window(A,Wn,C)),
       retract(total(Tw,To)),
       C1=C+1,To1=To+1,
       asserta(window(A,Wn,C1)),
       assert(total(Tw,To1)),
       put_diagram(Wn,X,Y).



/* keep waiting for next char */
     g_key('g',X,Y) :-
       readchar(C),
       g_key(C,X,Y).


/* neglect first 'g' */
     g_key(_,_,_).


/* on object that can be zoomed in */
     z_key('i',X,Y) :-
       obdb(A,' ',G,S,X,Y,N,Tok,Spec),
       obdb(B,'*',G,S,X,Y,N,Tok,Spec),
       retract(active(A)),
       assert(active(B)),window(B,Wn,_),
       fronttoken(Header,"Access      Graph                    ",Wn),
       removewindow,
       makewindow(1,7,7,Header,0,0,25,80),
       redraw,
       recursor(X,Y).
```

```
/* creating new zoom-in diagram */
     z_key('i',X,Y) :-
       obdb(A,' ',G,S,X,Y,N,Tok,Spec),
       retract(active(A)),
       retract(total(W,O)),
       W1=W+1,
       assert(total(W1,O)),
       assert(active(W1)),
       assertz(obdb(W1,'*',G,S,X,Y,N,Tok,Spec)),
       assertz(window(W1,N,1)),
       fronttoken(Header,"Access      Graph                    ",N),
       removewindow,
       makewindow(1,7,7,Header,0,0,25,80),
       asserta(linedb(W1,[])),
       redraw,
       recursor(X,Y).

/* object not found */
     z_key('i',_,_).

/* zoom out to parent diagram */
     z_key('o',X,Y) :-
       obdb(A,'*',G,S,X,Y,N,Tok,Spec),
 /*  fact  for object at higher level diagram is marked ' '  */
       obdb(B,' ',G,S,X,Y,N,Tok,Spec),
       retract(active(A)),
       assert(active(B)),
       window(B,Wn,_),
       removewindow,
       put_diagram(Wn,X,Y).

/* can't be zoomed out */
     z_key('o',_,_).

/* move cursor or give command for drawing lines */
     start_connect(X,Y) :-
       readchar(C),
       char_int(C,Val),
       move(Val,X,Y).

/* draw one line */
     makeline(X1,Y1,X2,Y2) :-
       A1=1280*X1,
       B1=400*Y1-200,
       A2=1280*X2,
       B2=400*Y2-200,
       line(A1,B1,A2,B2,7).

/* 'b' to mark starting point of the connection line */
     move(98,X,Y) :-
       assert(bline(X,Y)),
       start_connect(X,Y).
```

```
/*  't'  for making intermediate point for connection  line  */
     move(116,X2,Y2) :-
       retract(bline(X1,Y1)),
       makeline(X1,Y1,X2,Y2),
       active(A),
       retract(linedb(A,In)),
       appendtoken(In,[li(X1,Y1,X2,Y2)],Out),
       asserta(linedb(A,Out)),
       assert(bline(X2,Y2)),
       start_connect(X2,Y2).

/* 'e' to mark end point of the connection line */
     move(101,X2,Y2) :-
       retract(bline(X1,Y1)),
       makeline(X1,Y1,X2,Y2),
       arrowhead(X2,Y2),
       active(A),
       retract(linedb(A,In)),
       appendtoken(In,[li(X1,Y1,X2,Y2)],Out),
       asserta(linedb(A,Out)),
       start_connect(X2,Y2).

/* 'a' to get the name of the accessed object */
     move(97,X,Y) :-
       active(A),
       obdb(A,_,_,_,X,Y,N,_,_),
       assert(access_name(N)).


/* Arrow key for cursor movement */
     move(0,X,Y) :-
       readchar(T),char_int(T,Val),
       move(Val,X,Y).


/* Down */
     move(80,20,Y) :- start_connect(20,Y).
     move(80,X,Y) :-
       X1=1280*X,Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       X2=1280*(X+1),line(X2,Y1,X2,Y2,7),
       A=X+1,start_connect(A,Y).


/* Up */
     move(72,1,Y) :- start_connect(1,Y).
     move(72,X,Y) :-
       X1=1280*X,Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       X2=1280*(X-1),line(X2,Y1,X2,Y2,7),
       A=X-1,start_connect(A,Y).
```

```
/* Right */
     move(77,X,69) :- start_connect(X,69).
     move(77,X,Y) :-
       X1=1280*X,Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       Y3=400*Y,Y4=400*(Y+1),line(X1,Y3,X1,Y4,7),
       A=Y+1,start_connect(X,A).

/* Left */
     move(75,X,1) :- start_connect(X,1).
     move(75,X,Y) :-
       X1=1280*X,Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X1,Y2,0),
       Y3=400*(Y-2),Y4=400*(Y-1),line(X1,Y3,X1,Y4,7),
       A=Y-1,start_connect(X,A).

/* rest of the keys */
     move(_,X,Y) :- start_connect(X,Y).

/* make a arrowhead at the end of line */
     arrowhead(X,Y) :-
       X1=1280*X,X2=X1,Y1=400*(Y-1),Y2=400*Y,
       line(X1,Y1,X2,Y2,7),
       X3=1280*(X+1),Y3=400*Y-200,
       line(X1,Y1,X3,Y3,7),
       line(X2,Y2,X3,Y3,7),
       active(A),
       retract(linedb(A,In)),
       A1=X,B1=Y+1,A2=X+1,B2=Y,
       appendtoken(In,[li(A1,B1,X,Y),
       li(A2,B2,A1,B1),li(A2,B2,X,Y)],Out),
       asserta(linedb(A,Out)).
```

Appendix 2

Prolog code for specification entry and construction

```
domains

  file = outfile
  spec = string*
  token = g(string) ; title(string) ; part(tokenlist) ;
          name(string) ; var(string,string,string) ;
          comm(string) ; ret(string) ; unit(tokenlist) ;
          comms(tokenlist) ; para(tokenlist) ;
          access(tokenlist) ; with(string) ;
          li(integer,integer,integer,integer)
  tokenlist = token*

database

/* information about the component is stored in this structure
   1. window number
   2. a mark to indicate whether the fact in for higher
      or lower diagram
   3. generic ( "generic" or "" (non-generic),
      for control of box drawing and text spec
   4. type (packagem, subprogram or
           generic package and subprograms)
   5. X, row number
   6. Y, column number
   7. object name
   8. token list of the object
   9. text spec of the object                                */

        obdb(integer,char,string,string,integer,
              integer,string,tokenlist,spec)


/* current active window number */

        active(integer)

/* window number, name and total number of objects */

        window(integer,string,integer)

/* total number of windows and objects */

        total(integer,integer)

/* for building text spec (list of string,
    each element stands for one line of code) */

        specdb(spec)
```

```
/* for store list of token for an component */

        tokendb(tokenlist)

/* a program unit in a package, to be appended to
     the token list of the package*/

        unitdb(tokenlist)

/* to store row and column # for the
    starting point of a line */

        bline(integer,integer)

/* store a list of lines for the window number */

        linedb(integer,tokenlist)

/* store the name of the accessed object */

        access_name(string)

predicates

/* append a new token to tokenlist */

     appendtoken(tokenlist,tokenlist,tokenlist)

/* append a new line of code to spec */

     append(spec,spec,spec)

/* combine six strings into one line of code
                  --> single element of list */

     construct(string,string,string,
               string,string,string,string,spec)

/* following predicates are for drawing object
   1. generic or not ("generic" or "")
   2. title on the box drawn
   3,4. row and column number
   5. name of the object                            */

     draw(string,string,integer,integer,string)

/* draw box and ask for name
      (generic or not, object type, X, Y, object name )  */

       draw_name(string,string,integer,integer,string)

/* (generic or not (g or c), X, Y, object name) */

       empty(string,integer,integer,string)
```

```
/* (generic or not -> control the shape of boxes ,
      X, Y) */

       square(string,integer,integer)

/*----------------------------------------------------------*/
/* following predicates are for specification editing */

/* (generic or not, object type, row #, column #) */

       package(string,string,integer,integer)
       subprgm(string,string,integer,integer)

/* for determining procedure or function
   (generic or not, 'p' or 'f', object name) */

       p_or_f(string,char,string)

/* for getting input infor from user about procedure
   (generic or not, object name)                       */

       procedure(string,string)

/* for getting input infor from user about function
   (generic or not, title, object name)                 */

       function(string,string,string)


/* for getting input infor from user about package
   (generic or not, title, object name)                 */

       build_pack(string,string)

/* construct procedure, function or package text
    specification  from  the  token list in  the  database   */

       proc_spec
       func_spec
       pack_spec

/* get program unit infor for package */

       get_unit

/* get procedure or function information
    depending upon the parameter      */

       unit(char)

/* decompose a list of units for package
     to build text spec for each unit part    */

       decomp(tokenlist)
```

```
/* return a list of string after user input comments */

       takecomm(tokenlist)

/*  input  list for appending and return
       final string  list  */

       comment(tokenlist,tokenlist)

/* control in out mode for procedure or function,
    and return final var list                        */

       takepara(symbol,tokenlist)

/* control in out mode, input list for appending,
    in out identifier and final list to be returned */

       getpara(symbol,tokenlist,string,tokenlist)

/* convert a list of comment to spec and
    append it to specdb predicate in database   */

       writecomm(tokenlist)

/* convert a list of accessed object names to spec and
    append it to specdb predicate in database   */

       writewith(tokenlist)

/*  for  with  clause when more than one  accessed  objects  */

       morewith(tokenlist)

/* generic or empty string, variable list, title and
    end string for building text spec and append it to
     specdb   in  the  database                            */

       writepara(string,tokenlist,string,string)

/* when more than one variable,
    variable list and end string for text spec */

       morethanone(tokenlist,string)

clauses

       append([],L,L).
       append([X¦L1],L2,[X¦L3]) :-
             append(L1,L2,L3).

       appendtoken([],L,L).
       appendtoken([X¦L1],L2,[X¦L3]) :-
             appendtoken(L1,L2,L3).
```

```
/* for combining seven strings into one line of code */

     construct(S1,S2,S3,S4,S5,S6,S7,L) :-
       fronttoken(Sb1,S1,S2),fronttoken(Sb2,Sb1,S3),
       fronttoken(Sb3,Sb2,S4),fronttoken(Sb4,Sb3,S5),
       fronttoken(Sb5,Sb4,S6),fronttoken(Sb6,Sb5,S7),
       append([],[Sb6],L).


/************** SPECIFICATION EDITING *****************/

/* edit specification for the subprogram and assert it */

     subprgm(G,S,X,Y) :-
       draw_name(G,S,X,Y,N),
       fronttoken(Title,"Specification entry for
                         component ",N),
       makewindow(1,7,7,"",0,0,25,80),
       removewindow,
       makewindow(2,7,7,Title,3,10,18,60),nl,
       assert(specdb([])),
       write("Procedure or Function ? (p or f): "),
       readchar(P),write(P),nl,
       p_or_f(G,P,N),
       removewindow,
       retract(specdb(Spec)),
       retract(tokendb(Tok)),
       active(A),
       assertz(obdb(A,' ',G,S,X,Y,N,Tok,Spec)).

     p_or_f(G,'p',N) :-
       procedure(G,N),
       proc_spec.

     p_or_f(G,'f',N)  :-
       function(G,"procedure  ",N),
       func_spec.

     p_or_f(G,_,N) :-
       clearwindow,nl,
       write("Procedure or Function ? (p or f): "),
       readchar(T),write(T),nl,
       p_or_f(G,T,N).
```

```
/* edit specification for the package and assert it */

     package(G,S,X,Y) :-
       draw_name(G,S,X,Y,N),
       fronttoken(Title,"Specification entry for
                          component ",N),
       makewindow(1,7,7,"",0,0,25,80),
       removewindow,
       makewindow(2,7,7,Title,3,10,18,60),nl,
       assert(specdb([])),
       build_pack(G,N),
      removewindow,
      retract(specdb(Spec)),
      retract(tokendb(Tok)),
      active(A),
      assertz(obdb(A,' ',G,S,X,Y,N,Tok,Spec)).

    build_pack(G,N) :-
      takecomm(C),
      assert(unitdb([])),
      get_unit,
      retract(unitdb(U)),
      assert(tokendb([comms(C),access([]),g(G),
             title("package "),name(N),unit(U)])),
      pack_spec.


    get_unit :-
      clearwindow,
      write("Procedure or Function ? (p or f): "),
      readchar(T),write(T),T<>'\13',nl,
      unit(T),
      !,get_unit.

    get_unit.


    unit('p') :-
      write(" name : "),readln(N),nl,
      procedure("",N),
      retract(tokendb(Tok)),
      retract(unitdb(In)),
      appendtoken(In,[part(Tok)],Out),
      assert(unitdb(Out)).

    unit('f') :-
      write(" name : "),readln(N),
      function("","function ",N),
      retract(tokendb(Tok)),
      retract(unitdb(In)),
      appendtoken(In,[part(Tok)],Out),
      assert(unitdb(Out)).

    unit(_).
    pack_spec :-
      retract(tokendb([comms(C),access(A),g(G),
              title(T),name(N),unit(U)])),
      writecomm(C),
      writewith(A),
      retract(specdb(In)),
      append(In,[G],M1),
      fronttoken(H1,T,N),
      fronttoken(H2,H1," is"),
      append(M1,[H2],M2),
      assert(specdb(M2)),
      decomp(U),
      retract(specdb(En)),
      fronttoken(E1,"end ",N),
      fronttoken(E2,E1," ;"),
      append(En,[E2],Out),
      assert(specdb(Out)),
      assert(tokendb([comms(C),access(A),g(G),
             title(T),name(N),unit(U)])).

    decomp([]).

    decomp([part([C,A,Z,title("function ")|R])|L]) :-
      assert(tokendb([C,A,Z,title("function ")|R])),
      func_spec,
      retract(tokendb(_)),
      !,decomp(L).

    decomp([part([C,A,Z,title("procedure ")|R])|L]) :-
      assert(tokendb([C,A,Z,title("procedure ")|R])),
      proc_spec,
      retract(tokendb(_)),
      !,decomp(L).


/* draw a box for the object and ask for name */
    draw_name(G,S,X,Y,N) :-

      empty(G,X,Y,S),
      removewindow,
      makewindow(1,7,7,"",21,0,3,80),
      write("Please enter the name:  "),
      readln(N),
      removewindow.


    function(G,T,N) :-
      write("   return ? "),readln(R),nl,
      takecomm(C),
      takepara(function,P),
      assert(tokendb([comms(C),access([]),g(G),
            title(T),name(N),para(P),ret(R)])).
```

```
func_spec :-
  tokendb([comms(C),access(A),g(G),title(T),
           name(N),para(P),ret(R)]),
  writecomm(C),
  writewith(A),
  fronttoken(H,T,N),
  fronttoken(E," return ",R),
  writepara(G,P,H,E).


procedure(G,N) :-
  takecomm(C),
  takepara(procedure,P),
  assert(tokendb([comms(C),access([]),g(G),
          title("procedure "),name(N),para(P)])).

proc_spec :-
  tokendb([comms(C),access(A),g(G),
           title(T),name(N),para(P)]),
  writecomm(C),
  writewith(A),
  fronttoken(H,T,N),
  writepara(G,P,H,"").


takecomm(L) :-
  write("Enter comments after --"),nl,
  comment([],L).


comment(In,Out) :-
  write("  -- "),
  readln(Line),
  Line <> "",
  fronttoken(A," -- ",Line),
  appendtoken(In,[comm(A)],M),
  !,comment(M,Out).

comment(In,In).


writecomm([]).

writecomm([comm(Head)|Tail]) :-
  retract(specdb(In)),
  append(In,[Head],Out),
  assert(specdb(Out)),
  !,writecomm(Tail).
```

```
writewith([]).

writewith([with(N)|[]]) :-
  retract(specdb(In)),
  fronttoken(H1,"with ",N),
  fronttoken(H2,H1," ;"),
  append(In,[H2],Out),
  assert(specdb(Out)).

writewith([with(N)|R]) :-
  retract(specdb(In)),
  fronttoken(H1,"with ",N),
  fronttoken(H2,H1," ,"),
  append(In,[H2],Out),
  assert(specdb(Out)),
  morewith(R).


morewith([with(N)|[]]) :-
  retract(specdb(In)),
  fronttoken(H1,"     ",N),
  fronttoken(H2,H1," ;"),
  append(In,[H2],Out),
  assert(specdb(Out)).


morewith([with(N)|R]) :-
  retract(specdb(In)),
  fronttoken(H1,"     ",N),
  fronttoken(H2,H1," ,"),
  append(In,[H2],Out),
  assert(specdb(Out)),
  morewith(R).


writepara(G,[],Head,End) :-
  retract(specdb(In)),
  append(In,[G],M),
  fronttoken(E2,End," ;"),
  fronttoken(H2,Head,E2),
  append(M,[H2],Out),
  assert(specdb(Out)).


writepara(G,[var(Io,Name,Type)|[]],Head,End) :-
  retract(specdb(In)),
  append(In,[G],M),
  fronttoken(H2,Head," ( "),
  fronttoken(E2," ) ",End),
  fronttoken(E3,E2," ;"),
  construct(H2,Name," : ",Io," ",Type,E3,L),
  append(M,L,Out),
  assert(specdb(Out)).
```

```
writepara(G,[var(Io,Name,Type)¦Rest],Head,End) :-
  retract(specdb(In)),
  append(In,[G],M),
  fronttoken(H2,Head," ( "),
  construct(H2,Name," : ",Io," ",Type," ; ",L),
  append(M,L,Out),
  assert(specdb(Out)),
  morethanone(Rest,End).


morethanone([var(Io,Name,Type)¦[]],End) :-
  retract(specdb(In)),
  fronttoken(E2," ) ",End),
  fronttoken(E3,E2," ;"),
  construct("                ",Name,
            " : ",Io," ",Type,E3,L),
  append(In,L,Out),
  assert(specdb(Out)).


morethanone([var(Io,Name,Type)¦Rest],End) :-
  retract(specdb(In)),
  construct("                ",Name,
           " : ",Io," ",Type," ; ",L),
  append(In,L,Out),
  assert(specdb(Out)),
  !,morethanone(Rest,End).


takepara(Pf,P) :-
  nl,
  write("Please enter the name and type of the
         variables:"),
  nl,nl,getpara(Pf,[],"in",P).


getpara(Pf,Blist,"in",P) :-
  write(" in : "),readln(N),N<>"",
  write(" type : "),readln(T),
  appendtoken(Blist,[var("in",N,T)],Newlist),
  !,getpara(Pf,Newlist,"in",P).


getpara(procedure,Blist,"in",P) :-
  nl,getpara(procedure,Blist,"out",P).

getpara(function,Blist,"in",Blist).
```

```
    getpara(procedure,Blist,"out",P) :-
      write(" out : "),readln(N),N<>"",
      write(" type : "),readln(T),
      appendtoken(Blist,[var("out",N,T)],Newlist),
      !,getpara(procedure,Newlist,"out",P).


    getpara(procedure,Blist,"out",P) :-
      nl,getpara(procedure,Blist,"inout",P).


    getpara(procedure,Blist,"inout",P) :-
      write(" inout : "),readln(N),N<>"",
      write(" type : "),readln(T),
      appendtoken(Blist,[var("inout",N,T)],Newlist),
      !,getpara(procedure,Newlist,"inout",P).

    getpara(procedure,Blist,"inout",Blist).


/************** DRAWING BOXES *******************************/

/* draw a object with its name */

     draw(G,S,X,Y,N) :-
       empty(G,X,Y,S),
       X2=X+3,Y2=Y+2,
       makewindow(2,7,0,"",X2,Y2,1,8),
       write(N),removewindow.

/* draw an empty package */

     empty(G,X,Y,S) :-
       square(G,X,Y),
       X1=X+1,Y2=Y+2,
       makewindow(2,7,0,"",X1,Y2,1,8),
       write(S),
       removewindow.
```

```
/* draw a square as an object */
    square(G,X,Y) :-
      X1=1280*(X-1)+500,Y1=400*Y,
      A1=X1,B1=400*(Y-1),A2=1280*X+430,B2=B1,
      A3=A2,B3=Y1,
      X2=1280*(X+3)+500,Y2=Y1,
      X3=X1,Y3=400*(Y+9),
      X4=X2,Y4=Y3,
      X5=1280*(X+1)+500,Y5=Y1,/*     I1-----------------I2   */
      X6=X5,Y6=Y3,           /*    /                / |   */
      line(A1,B1,A2,B2,7),   /*  A1----1-----------3  |   */
      line(A2,B2,A3,B3,7),   /*  |     |  window   |  |   */
      line(X1,Y1,X2,Y2,7),   /*  |     5-----------6  |   */
      line(A1,B1,X3,Y3,7)    /*  A2---A3           | I3   */
      line(X2,Y2,X4,Y4,7),   /*        |   Name    | /    */
      line(X3,Y3,X4,Y4,7),   /*        2-----------4      */
      line(X5,Y5,X6,Y6,7),   /*                           */
      G<>"generic",
      I1=1280*(X-1)+300,J1=Y1,
      I2=I1,J2=400*(Y+10),
      I3=1280*(X+2)+500,J3=J2,
      line(A1,B1,I1,J1,7),
      line(I1,J1,I2,J2,7),
      line(X3,Y3,I2,J2,7),
      line(I2,J2,I3,J3,7),
      line(X4,Y4,I3,J3,7).
    square(_,_,_).
```

PROLOG IMPLEMENTATION OF A GRAPHIC TOOL FOR

GENERATION OF ADA LANGUAGE SPECIFICATIONS

by

SPENCER SHU-TSU CHENG

B.S., NATIONAL CHENG-KUNG UNIVERSITY, 1980
M.S., KANSAS STATE UNIVERSITY, 1986

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer and Information Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1987

ABSTRACT

This report presents the design and implementation of a graphic tool for the design and specification of the modular structure of Ada programs. The function of the software tool is based upon a similar tool that was presented in a previous report and implemented in Pascal. The current version is implemented in Turbo Prolog, a variant of Prolog that executes on a microcomputer and supports graphics and windows for the user interface.

The purpose of this project was to explore the suitability of Prolog for graphics and system design tools, particularly to evaluate the size and readability of the Prolog vs. Pascal code. We find that the source code in Prolog is relatively shorter than equivalent code in Pascal. We feel that Prolog code is more concise and readable than Pascal. The graphic mode of Turbo Prolog does not support mixing of text with graphic objects; however, the use of overlayed windows provides a mechanism to create diagrams with integrated text information.