CIPS/ACI Congress 84, May 1984, pp. 105-112.

GENERAL USER INTERACTIVE DESIGN ENVIRONMENT: AN OVERVIEW

Duane Szafron, John Adria, Brian Wilkerson

Programming Languages Group, Department of Computing Science
University of Alberta
Edmonton, Alberta, CANADA, T6G 2H1

Abstract

This paper presents an overview of the General User Interactive Design Environment (GUIDE). It includes a discussion of the general principles of the system, a simulated example, and an extract from the formal specifications of a task model for the system.

Cet article présente de façon sommaire un système général pour la conception de programmes nommé "General User Interactive Design Environment"(GUIDE). Il comporte une discussion des principes directeurs du système, un exemple simulé, et un extrait des spécifications formelles des taches du système.

INTRODUCTION

Consider a software design environment which allows the user to enter the conceptual specifications for a complex piece of software and produces a valid program that meets these specifications, along with detailed performance evaluations for the program. No design environment exists today which can meet this challenge. The approaches to this problem and its component parts are quite diverse and span the breadth of computing science. The problem can be broken down many ways. One view of the breakdown consists of:

1) Transformation of conceptual specifications to formal specifications.

2) Detailed modular design.

3) Coding of modules.

4) Debugging of modules.

5) Testing and performance evaluation.

Although this list of steps appears to be sequential, it does in fact, involve considerable backtracking. In addition, program verification must be done concurrently instead of being left to the end of the process where it would be too complex.

Although no complete environment exists for the entire design process, there are many software tools which aid in various phases of the development cycle. Certainly, the coding and debugging phases have seen the largest growth in the availability of computer tools. There are many good programming environments which have been designed and implemented in the last five years. These environments include: the Cornell Program Synthesizer (CPS)[1], the "Z" editor[2], the SDS editor[3], CAPS[4], and Emily[5]. They generally provide an editor for program code entry, with complete entry-time syntax checking, and various degrees of entry-time semantic checking, as well as run-time debugging support.

It is quite natural to expect that these programming environments should be extended to design environments which incorporate all of the phases of the design process. It is also natural that the design environments should evolve in an incremental fashion, first incorporating the detailed design phase and then attacking the specification phases.

A good detailed design environment should encourage the user to follow good design practices. It should eliminate both syntactic and semantic errors at code entry-time, by either automatically correcting the errors, or by delineating them for the immediate attention of the user. It should support run-time debugging and program testing. Above all, it should recognize and support the non-sequential nature of the design phases by allowing the modular design to change as coding and debugging uncover logic errors.

This paper describes a task model for a prototype design environment which incorporates phases 2, 3 and 4 as described above. It provides syntax and semantic checks at code entry time using an algorithmic language for the code. Code is stored internally in parse trees and can be interpreted for run-time debugging. When a complete program has been designed, coded and debugged, the environment will generate Pascal code which can be compiled externally for production purposes. Since this is only a prototype system, only a restricted set of data and control structures are used. The main goal of this prototype system is to study the interaction of the design, coding and debugging phases of the development process.

THE ENVIRONMENT

Detailed software design consists of the translation of human software specifications for the solution of a single large complex problem, to human software specifications for the solution of a multitude of smaller simpler problems. In this paper, the object used to represent a single problem, will be called a module. A module which represents a small simple problem which requires no more subdivision will be called a fundamental module. Detailed design can then be described as the process of dividing complex modules into groups of simpler modules. We shall call this process, module refinement. Coding can be described as the the process of transforming the contents of a module from the human specification of a problem to a representation of the problem suitable for machine solution. We shall call this process, module transformation.

During the software development process, detailed design is typically represented as a process which occurs before coding. That is, complete module refinement is supposed to be carried out before any module transformation. Coding however, generally exposes design flaws which result in changes to the design. It is for this reason that detailed design and coding should be performed in a common environment.

In such an environment the user begins by entering the name of the program. The environment responds by creating a module whose only attribute is the module name. The user then refines the module by entering the names of submodules. The environment responds by creating more modules whose attributes are the module names and information which represents the hierarchy of the modular structure. The user proceeds in this way until all modules are fundamental.

At this point, the user begins module transformation, by entering both data flow information and code. The environment has two basic responsibilities during module transformation. One responsibility is to ensure that the module interfaces remain consistent. A change in the data flow requirements for one module must result in changes for the data flow requirements in all modules which invoke it. The second responsibility of the environment is to ensure that the user is notified about syntactic and semantic errors as code is entered. Recall however, that the most important feature of the design environment is that any time a design error is uncovered, the user can modify the design by module refinement operations. The transition from module refinement operations to module transformation operations and back again should require a minimum of effort from the user. For example, consider the case where a user is entering code and discovers that a new submodule is required. The environment must provide a simple command so that the user can generate the new module and make any module transformations to it that are deemed necessary, including data flow and code entry. Following this action, the environment must return the user to the point at which this diversion occurred without requiring the user to find the location of the old module in the module hierarchy.

To accomplish these objectives, we propose a task model in which four distinct sets of tasks are available. Three of these sets of tasks correspond to user views of the environment: the Hierarchy View, the Declaration View and the Code View. The hierarchy view is distinctive in that there is only one view which consists of all modules, while there are separate code and declaration views for each module. The fourth set of tasks, called global tasks, are for changing views and interacting with the world outside of the environment. They can be performed from any view.

There is a consistency in the user's conceptual model of GUIDE which exists throughout the views. In each view, the user manipulates ordered trees of structures. In the code view, these structures are statements and components of statements, whereas in the declaration view, the components are declarations and components of declarations.

Four guiding principals were used in designing these views. All information known by the environment must be available to the user. Information is classified by view and each piece of information can be changed in only one view. All views represent information as structures, not text. Changing views must be simple and fast.

The Hierarchy View

This view encompasses the hierarchical structure of the software being designed. It is represented by an ordered tree in which the nodes are named boxes representing program modules. A program module may, of course, appear in several places in the hierarchy. A formal specification for the tasks in this view are described in appendix A. A grammar for the formal task definition language which is used, and a discussion of its features appears in [6]. Essentially, the user is allowed to move a cursor from node to node in the hierarchy and to insert, delete and move subtrees. The following figure is a sample screen from the hierarchical view of a simulation of the GUIDE prototype.

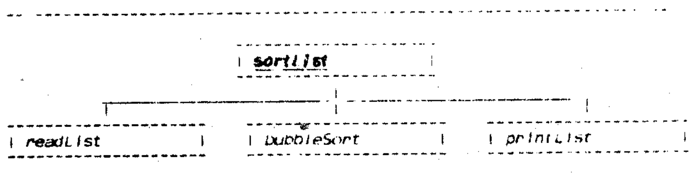
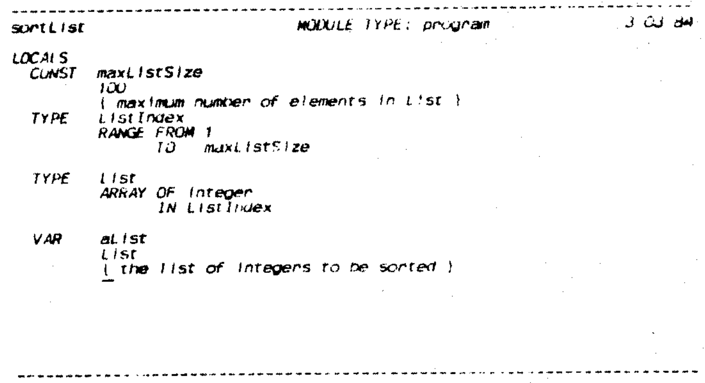


Figure 1: An example of the hierarchy view.

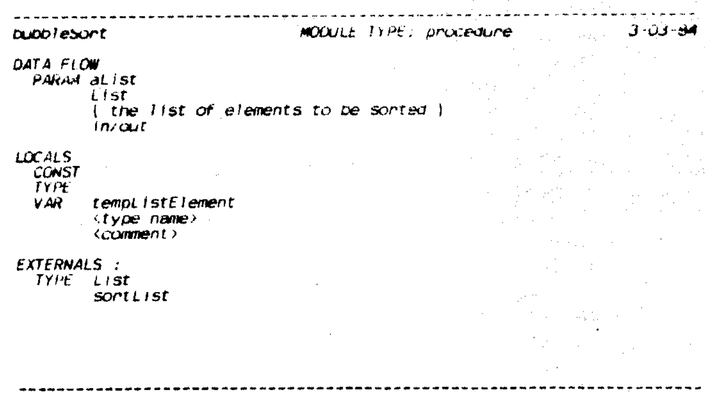
The Declaration View

The user can display the declaration view of any module, and edit the declarations for all identifiers appearing in that module. These declarations are displayed in a format so that the user can use a conceptual model of an ordered tree to edit the information. This is accomplished by moving a cursor to the various components of a declaration and replacing the information in the chosen component. New declarations are created by commands which generate declaration templates. The formal task model for this view appears in [6], and is similar to the task model for the hierarchical view which appears in appendix A. It is important to note that the information is displayed in a form which is independent of the target language (Pascal in this prototype). The following two figures are sample screens from declaration views of a simulation of the GUIDE prototype.



```
sortList                          MODULE TYPE: program            3 03 84

LOCALS
  CONST   maxListSize
          100
          { maximum number of elements in List }
  TYPE    ListIndex
          RANGE FROM 1
                TO   maxListSize

  TYPE    List
          ARRAY OF Integer
                IN ListIndex

  VAR     aList
          List
          { the list of integers to be sorted }
```

Figure 2: The declaration view for module sortList



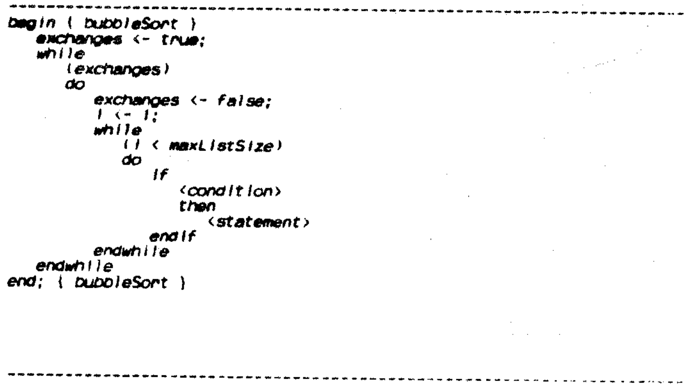
```
bubbleSort                        MODULE TYPE: procedure          3-03-84

DATA FLOW
  PARAM aList
        List
        { the list of elements to be sorted }
        in/out

LOCALS
  CONST
  TYPE
  VAR    tempListElement
         <type name>
         <comment>

EXTERNALS :
  TYPE  List
        sortList
```

Figure 3: The declaration view for module bubbleSort.

Notice that in the above figure, templates for <type name> and <comment> of the variable tempListElement have not yet been inserted by the user. The external type "List", and the name of the external module in which it is declared, were not inserted by the user. They were inserted by GUIDE in response to the users declaration of aList as a parameter of this type. When an undeclared identifier is inserted by the user, either in the declaration view or the code view, GUIDE searches the hierarchy to find the module with maximum depth in the ordered tree, which is an ancestor of all occurrences of the current module, and which has a declaration of the identifier under consideration. For such an identifier, GUIDE inserts a line in the externals section of the declaration view consisting of the identifier's name and category (constant, type, variable or module) together with the name of the module in which it is declared. If no such module is found then the special module name "UNDEFINED" is used. If the user makes a local declaration of an identifier which is in the externals list, then GUIDE removes it from the externals list.

The Code View

As in the declaration view, the user edits structures by moving a cursor to a node in this ordered tree and by replacing the information in the selected node. The nodes in this view consist of statements and components of statements. New statments are created by commands which generate statment templates. The code is displayed in an algorithmic notation which reflects the parse tree it represents. A grammer for this notation appears in appendix B and a formal task model for this view appears in [6]. The following figure is a sample screen from the code view of a simulation of the GUIDE prototype.



```
begin ( bubbleSort )
   exchanges <- true;
   while
      (exchanges)
      do
         exchanges <- false;
         I <- 1;
         while
            (I < maxListSize)
            do
               if
                  <condition>
                  then
                     <statement>
               endif
         endwhile
   endwhile
end; ( bubbleSort )
```

Figure 4: The code view for module bubbleSort.

At this point, the user has created a template for an if-then statement, but has not yet replaced the <condition> or <statment> components. At some level, component structure must be abandoned in favor of a string of text. If this is not done then the entry of of a simple addition expression as <identifier> + <identifier> becomes tedious for the user. The choice of level at which structure is abandoned is crucial to a system which guides, but does not hinder the user. Through simulations of GUIDE, we have determined a collection of syntactic structures which are "primitive" in that they should be entered as text. The primitives include: conditions in while, if-then and if-then-else statements as well as complete assignment statements and module invocations. Although the primitives are entered as text, they are not stored this way. The primitives are parsed as they are entered and displayed on the screen in a "standard" format. The format includes the full parenthesization of expressions. This feature allows the user to immediately ascertain GUIDE's interpretation of all primitives which are entered. A primitive is edited in a one line window at the bottom of the screen so that the context of the primitive will remain visible.

The Global Tasks

These tasks are the ones which allow the user to interact with the operating system. The user may change from any view (the hierarchy view or the code or declaration view of any module) to any other view, save the environment or restore it, define command macros and execute an operating system command.

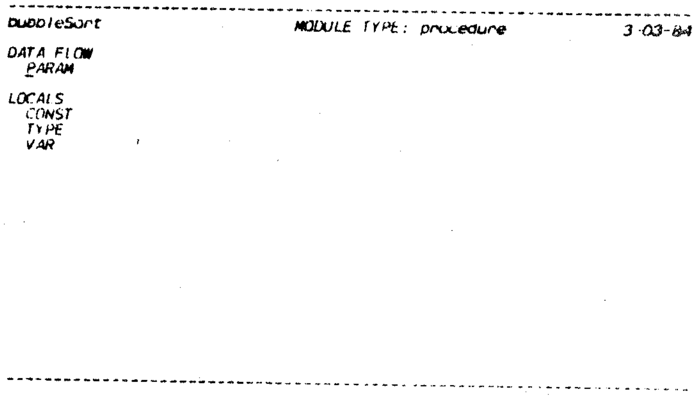
DEMONSTRATION

We now present a brief demonstration of a simulation of the GUIDE system. The user has used the hierarchy view to create the four modules in figure 1. When the declaration view of the module bubbleSort is entered, the screen will appear as in figure 5.

The cursor, at the "P" in "PARAM", is positioned at the start of the parameter declaration section. Therefore, when the user requests a template, GUIDE provides a parameter declaration template. The screen appears as in figure 6.

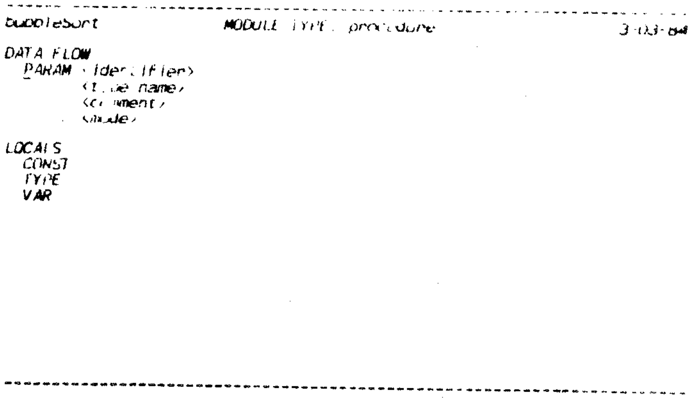
The user can expand the identifier component of the parameter template by performing the following operations. A single command moves the cursor to the right so that it rests on the "<" of "<identifier>". A second command requests replacement of the component by primitive text. The user enters the text and the system checks for semantic errors. The new screen is displayed in figure 7.

In a similar fashion, the user moves down to each of the other components and replaces them by primitive text. In each case, GUIDE checks for semantic errors,



```
bubbleSort                         MODULE TYPE: procedure                  3-03-84

DATA FLOW
  PARAM

LOCALS
  CONST
  TYPE
  VAR
```

Figure 5: The initial template for the declaration view.



```
bubbleSort                         MODULE TYPE: procedure                  3-03-84

DATA FLOW
  PARAM <identifier>
        <type name>
        <comment>
        <mode>

LOCALS
  CONST
  TYPE
  VAR
```

Figure 6: A parameter declaration template.

```
bubbleSort              MODULE TYPE: procedure                   3-03-84

DATA FLOW
  PARAM aList
        <type name>
        <comment>
        <mode>

LOCALS
  CONST
  TYPE
  VAR
```

Figure 7: Replacement of an identifier template.

```
bubbleSort              MODULE TYPE: procedure                   3-03-84

DATA FLOW
  PARAM aList
        List
        { the list of elements to be sorted }
        in/out

LOCALS
  CONST
  TYPE
  VAR

EXTERNALS :
  TYPE  List
        sortList
```

Figure 8: A complete parameter declaration.

```
bubbleSort              MODULE TYPE: procedure                   3-03-84

DATA FLOW
  PARAM aList
        List
        { the list of elements to be sorted }
        in/out

LOCALS
  CONST
  TYPE
  VAR   <identifier>
        <type name>
        <comment>

EXTERNALS :
  TYPE  List
        sortList
```

Figure 9: A variable declaration parameter.

```
bubbleSort              MODULE TYPE: procedure                   3-03-84

DATA FLOW
  PARAM aList
        List
        { the list of elements to be sorted }
        in/out

LOCALS
  CONST
  TYPE
  VAR   exchanges
        boolean

  VAR   tempListElement
        integer

EXTERNALS :
  TYPE  List
        sortList
```

Figure 10: Completed declaration view of module bubbleSort.

before accepting the text. The resulting screen is shown in figure 8. As was mentioned earlier in the previous section of this paper, the external reference to the type "List" was generated by GUIDE.

In order to declare a local variable, the user gives the following sequence of commands. A single command moves the cursor left to the "P" of "PARAM". Three consecutive commands move the cursor down, first to the "C" of "CONST", then to the "T" of "TYPE" and finally to the "V" of "VAR". Since the user is now at the variable declaration section, a request for a template results in the generation of a variable declaration template. The result is shown in figure 9.

After several more editing commands, the declaration view of this module is in its final form. It appears in figure 10.

```
begin { bubbleSort }
   <statement>
end; { bubbleSort }
```

Figure 11: The initial template for the code view.

```
begin { bubbleSort }
   exchanges <- true
end; { bubbleSort }
```

Figure 12: Insertion of an assignment statement.

On entering the code view of module bubbleSort, the user is presented with the screen of figure 11. The cursor is positioned at the "<" of "<statement>". The user issues a command to replace the statement template by an assignment statement. Since an assignment statement is primitive, it is entered as text. The assignment statement is parsed and if it is legal, it is saved in the parse tree for the module. Otherwise the user is immediately notified of the error so that it can be edited. The user must correct the error before the primitive is placed in the parse tree. The resulting screen appears in figure 12.

A single command is then used to create a statment template after the current statement. The cursor is positioned at the "<" of the new "<statement>". The new screen is shown in figure 13. Notice that GUIDE has inserted a semicolon between statements.

In order to enter a while statement, the user issues a command to change the "general" statement template to a while statement template. The result appears in figure 14. GUIDE enters a while statment into the parse tree for this module.

```
begin { bubbleSort }
   exchanges <- true;
   <statement>
end; { bubbleSort }
```

Figure 13: Insertion of a general statement template.

```
begin { bubbleSort }
   exchanges <- true;
   while
      <condition>
      do
         <statement>
   endwhile
end; { bubbleSort }
```

Figure 14: Insertion of a while statement template.

To move the cursor to the "<condition>" component of the while statement, the user issues a move right command. The user then gives a command to replace the "<condition>" by primitive text. The screen is updated as shown in figure 15. Because of the fact that GUIDE stores all primitive text in parsed format, all expressions are displayed in fully parenthesized form. This is true regardless of whether or not the user enters the parentheses.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         <statement>
   endwhile
end; { bubbleSort }
```

Figure 15: GUIDE displays fully parenthesized expressions.

In order to expand the body of the while statement template, the user first moves the cursor down to the "d" in "do" and then right to the "<" in "<statement>". Since three statments are to be entered in the body of the while, the user issues the insert statement above command twice. The screen now appears as in figure 16. The cursor remains on the first "<statement>".

After replacing the first component by primitive text, moving the cursor down, replacing the second component by primitive text, and moving the cursor down again, the screen in figure 17 is displayed.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         <statement>;
         <statement>;
         <statement>
   endwhile
end; { bubbleSort }
```

Figure 16: Multiple statement templates in a while template.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         exchanges <- false;
         i <- 1;
         <statement>
   endwhile
end; { bubbleSort }
```

Figure 17: Replacing statement templates by assignment statements.

The user once again uses a single command to replace the current statement by a while statement template and figure 18 is the result.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         exchanges <- false;
         i <- 1;
         while
            <condition>
            do
               <statement>
         endwhile
   endwhile
end; { bubbleSort }
```

Figure 18: Insertion of a nested while statement.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         exchanges <- false;
         i <- 1;
         while
            (i < maxListSize)
            do
               if
                  <condition>
                  then
                     <statement>
               endif
         endwhile
   endwhile
end; { bubbleSort }
```

Figure 19: Insertion of an if-then template.

The user moves the cursor to the right, replaces the "<condition>" by text, moves down to the "<statement>", and replaces it by an if-then statement template. The result of these actions is displayed in figure 19.

After further editing, the final code view of this module appears as in figure 20.

```
begin { bubbleSort }
   exchanges <- true;
   while
      (exchanges)
      do
         exchanges <- false;
         i <- 1;
         while
            (i < maxListSize)
            do
               if
                  (aList[i] > aList[i + 1])
                  then
                     tempListElement <- aList[i];
                     aList[i] <- aList[i + 1];
                     aList[i + 1] <- tempListElement;
                     exchanges <- true
               endif;
               i <- i + 1
         endwhile
   endwhile
end; { bubbleSort }
```

Figure 20: Completed code view of module bubbleSort.

References

[1] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Commun. ACM 24, (9, 1981), pp. 563-573

[2] S. R. Z. Wood, "The 95% Program Editor", Sigplan 16, (6,1981), pp. 1-7

[3] C. W. Fraser, "Syntax Directed Editing of General Data Structures", Sigplan 16, (6, 1981), pp. 17-21

[4] T. R. Wilcox, A. M. Davis, and M. H. Tindall, "The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System", Commun. ACM 19, (11, 1976), pp. 609-616

[5] W. J. Hansen, "User Engineering Principles for Interactive Systems", Fall Joint Computer Conference 39, (1971), pp. 523-532

[6] D. Szafron, J. Adria, B. Chadramouli and B. Wilkerson, "A Formal Task Specification Language", to appear

APPENDIX A

The following is an extract from the formal specifications of the task model for the GUIDE system. All of the tasks from the hierarchy view are included. An explanation of the formal specification language appears in [6].

```
class Module;
   name : String;
end;

class ModuleOTree;
   OTree ();
   info : Module;
end;

class ModuleOTreeDictionary;
   Dictionary (ModuleOTree);
end;

class Hierarchy;
   root           : ModuleOTree;
   cursor         : ModuleOTree;
   saveDictionary : ModuleOTreeDictionary;
end;

task movePred
      (aHierarchy : Hierarchy);

 { This task moves the cursor to the ModuleOTree  which
   is the predecessor of the current ModuleOTree.  }

   pre :
      aHierarchy.cursor.pred <> nil;
   delta :
      aHierarchy.cursor <-- aHierarchy.cursor.pred;
end;

task moveSucc
      (aHierarchy : Hierarchy);

 { This task moves the cursor to the ModuleOTree  which
   is the successor of the current ModuleOTree.  }

   pre :
      aHierarchy.cursor.succ <> nil;
   delta :
      aHierarchy.cursor <-- aHierarchy.cursor.succ;
end;

task moveFirstChild
      (aHierarchy : Hierarchy);

 { This task moves the cursor to the ModuleOTree  which
   is the first child of the current ModuleOTree.  }

   pre :
      aHierarchy.cursor.firstChild <> nil;
   delta :
      aHierarchy.cursor <--
         aHierarchy.cursor.firstChild;
end;

task moveParent
      (aHierarchy : Hierarchy);

 { This task moves the cursor to the ModuleOTree  which
   is the parent of the current ModuleOTree.  }

   pre :
      aHierarchy.cursor.parent <> nil;
   delta :
      aHierarchy.cursor <-- aHierarchy.cursor.parent;
end;

task moveToRoot
      (aHierarchy : Hierarchy);

 { This task moves the cursor to the root  ModuleOTree.
   }

   delta :
      aHierarchy.cursor <-- aHierarchy.root;
end;

task insertPred
      (aHierarchy : Hierarchy;
      moduleName : String);

 { This task inserts a ModuleOTree as the predecessor
   of the current ModuleOTree. If a module with the
   given name already exists, then a copy of the old
   ModuleOTree whose root module has that name, is in-
   serted. Otherwise, a new ModuleOTree containing a
   single module is created and inserted. The cursor is
   then moved to the root of the inserted ModuleOTree.
   Inserting a predecessor of the root ModuleOTree is
   forbidden.  }

   pre :
      aHierarchy.cursor.parent <> nil;
   delta :
      instance oldModuleOTree of ModuleOTree;
      oldModuleOTree <-- ModuleOTree$Find
         (aHierarchy.root, moduleName);
      if oldModuleOTree <> nil then
         ModuleOTree$insertPred (aHierarchy.cursor,
            oldModuleOTree);
      else
         ModuleOTree$insertPred (aHierarchy.cursor,
            ModuleOTree$create (moduleName));
      endif;
      aHierarchy.cursor <-- aHierarchy.cursor.pred;
end;

task insertSucc
        (aHierarchy : Hierarchy;
        moduleName : String);

 { This task inserts a ModuleOTree as the successor  of
   the current ModuleOTree.  If a module with the given
   name already exists, then a copy of the old ModuleO-
   Tree  whose  root module has that name, is inserted.
   Otherwise, a new  ModuleOTree  containing  a  single
   module  is  created and inserted. The cursor is then
   moved to  the  root  of  the  inserted  ModuleOTree.
   Insertion  of a successor of the root ModuleOTree is
   forbidden.  }

   pre :
      aHierarchy.cursor.parent <> nil;
   delta :
      instance oldModuleOTree of ModuleOTree;
      oldModuleOTree <-- ModuleOTree$Find
         (aHierarchy.root, moduleName);
      if oldModuleOTree <> nil then
         ModuleOTree$insertSucc (aHierarchy.cursor,
            oldModuleOTree);
      else
         ModuleOTree$insertSucc (aHierarchy.cursor,
            ModuleOTree$create (moduleName));
      endif;
      aHierarchy.cursor <-- aHierarchy.cursor.succ;
end;

task insertLastChild
        (aHierarchy : Hierarchy;
        moduleName : String);

 { This task inserts a ModuleOTree as the last child of
   the current ModuleOTree.  If a module with the given
   name already exists, then a copy of the old ModuleO-
   Tree  whose  root module has that name, is inserted.
   Otherwise, a new  ModuleOTree  containing  a  single
   module  is  created  and inserted. The cursor is not
   moved.  }

   delta :
      instance oldModuleOTree of ModuleOTree;
      oldModuleOTree <-- ModuleOTree$Find
         (aHierarchy.root, moduleName);
      if oldModuleOTree <> nil then
         ModuleOTree$insertLastChild
            (aHierarchy.cursor, oldModuleOTree);
      else
         ModuleOTree$insertLastChild
            (aHierarchy.cursor, ModuleOTree$create
            (moduleName));
      endif;
end;

task insertParent
        (aHierarchy : Hierarchy;
        moduleName : String);

 { This task inserts a ModuleOTree consisting of a sin-
   gle  new  module, as the parent of the current Modu-
   leOTree. The cursor is then moved to the root of the
   inserted ModuleOTree.  }

   pre :
      ModuleOTree$Find (moduleName) = nil;
   delta :
      ModuleOTree$insertParent (aHierarchy.cursor,
         ModuleOTree$create (moduleName));
      aHierarchy.cursor <-- aHierarchy.cursor.parent;
end;

task pastePred
        (aHierarchy     : Hierarchy;
        saveDictionary : ModuleOTreeDictionary;
        saveName       : String);

 { This task pastes a ModuleOTree as the predecessor of
   the  current  ModuleOTree.   The  ModuleOTree  to be
   pasted is copied from from the save dictionary loca-
   tion specified by the save name.  The cursor is then
   moved to the root of the pasted ModuleOTree.   Past-
   ing a predecessor of the root ModuleOTree is forbid-
   den.  }

   pre :
      aHierarchy.cursor.parent <> nil;
      saveDictionary[saveName] <> nil;
   delta :
      ModuleOTree$insertPred (aHierarchy.cursor,
         saveDictionary[saveName]);
      aHierarchy.cursor <-- aHierarchy.cursor.pred;
end;

task pasteSucc
        (aHierarchy     : Hierarchy;
        saveDictionary : ModuleOTreeDictionary;
        saveName       : String);

 { This task pastes a ModuleOTree as the  successor  of
   the  current  ModuleOTree.   The  ModuleOTree  to be
   pasted is copied from from the save dictionary loca-
   tion specified by the save name.  The cursor is then
   moved to the root of the pasted ModuleOTree.   Past-
   ing  a  successor of the root ModuleOTree is forbid-
   den.  }

   pre :
      aHierarchy.cursor.parent <> nil;
      saveDictionary[saveName] <> nil;
   delta :
      ModuleOTree$insertSucc (aHierarchy.cursor,
         saveDictionary[saveName]);
      aHierarchy.cursor <-- aHierarchy.cursor.succ;
end;

task pasteLastChild
        (aHierarchy     : Hierarchy;
        saveDictionary : ModuleOTreeDictionary;
        saveName       : String);

 { This task pastes a ModuleOTree as the last child  of
   the  current  ModuleOTree.   The  ModuleOTree  to be
   pasted is copied from from the save dictionary loca-
   tion  specified by the save name.  The cursor is not
   moved.  }

   pre :
      saveDictionary[saveName] <> nil;
   delta :
      ModuleOTree$insertLastChild (aHierarchy.cursor,
         saveDictionary[saveName]);
end;

task writeFromDictionary
        (saveDictionary : ModuleOTreeDictionary;
        saveName       : String;
        fileName       : String);

 { This task writes a ModuleOTree from  the  save  dic-
   tionary at the given save name, to a disk file.  }
   pre :
      saveDictionary[saveName] <> nil;
   delta :
      ModuleOTree$writeToFile
         (saveDictionary[saveName], fileName);
end;

task readIntoDictionary
         (saveDictionary : ModuleOTreeDictionary;
         saveName        : String;
         fileName        : String);

 { This task reads a ModuleOTree from a file to the
   save dictionary at the location of the given save
   name. }

   delta :
      saveDictionary[saveName] <--
         ModuleOTree$readFromFile (fileName);
end;

task copySubtree
         (aHierarchy     : Hierarchy;
         saveDictionary : ModuleOTreeDictionary;
         saveName       : String);

 { This task makes a copy of the current ModuleOTree
   and saves it in the ModuleOTree save dictionary at
   the position of the save name. }

   delta :
      saveDictionary[saveName] <-- ModuleOTree$copy
         (aHierarchy.cursor);
end;

task deleteSubtree
         (aHierarchy     : Hierarchy;
         saveDictionary : ModuleOTreeDictionary;
         saveName       : String);

 { This task moves the current ModuleOTree to the Modu-
   leOTree save dictionary at the position of the save
   name. The root ModuleOTree cannot be moved. If the
   current ModuleOTree had a successor, then the cursor
   moves to it. Otherwise, if the current ModuleOTree
   had a predecessor then the cursor moves to it. Oth-
   erwise, the cursor moves to the parent of the
   current ModuleOTree. }

   pre :
      aHierarchy.cursor.parent <> nil;
   delta :
      if aHierarchy.cursor.succ <> nil then
         Cursor$moveSucc (aHierarchy.cursor);
         aHierarchy.cursor <-- aHierarchy.cursor.succ;
         saveDictionary[saveName] <--
            ModuleOTree$delete
            (aHierarchy.cursor.pred);
      else
         if aHierarchy.cursor.pred <> nil then
            aHierarchy.cursor <--
               aHierarchy.cursor.pred;
            saveDictionary[saveName] <--
               ModuleOTree$delete
               (aHierarchy.cursor.succ);
         else
            aHierarchy.cursor <--
               aHierarchy.cursor.parent;
            saveDictionary[saveName] <--
               ModuleOTree$delete
               (aHierarchy.cursor.leftChild);
         endif;
      endif;
end;
```

APPENDIX B

The following is the grammar of the intermediate language for GUIDE, described in Backus-Naur form. Non-terminals are enclosed in angular brackets ("<>") and terminals are underlined.

It should be noted that the definition of expressions in this language are different from those in other languages and notably Pascal. This language does not depend on implicit arithmetic hierarchy to determine the order of the operations, rather, it uses explicit parenthesizing.

```
<module code> ::= begin <statement list> end;

<statement list> ::= <statement> ; <statement list> |
     <statement>

<statement> ::= <simple statement> |
     <structured statement>

<simple statement> ::= <assignment statement> |
     <procedure statement> | <return statement> |
     <empty statement>

<assignment statement> ::= <variable> <-
     <simple expression>

<variable> ::= <identifier> | <indexed variable>

<indexed variable> ::= <variable> [ <expression list> ]

<expression list> ::= <simple expression> ,
     <expression list> | <simple expression>

<procedure statement> ::= <identifier> (
     <parameter list> )

<parameter list> ::= <nonempty parameter list> | <null>

<nonempty parameter list> ::= <actual parameter> ,
     <nonempty parameter list> | <actual parameter>

<actual parameter> ::= <simple expression>

<return statement> ::= return <expression>

<empty statement> ::= '<statement>'

<structured statement> ::= <if statement> |
     <while statement>

<if statement> ::= if <expression> then
     <statement list> <else part> endif

<else part> ::= else <statement list> | <null>

<while statement> ::= while <expression> do
     <statement list> endwhile

<expression> ::= ( <simple expression> )

<simple expression> ::= <term> <operator> <term>

<operator> ::= = | <> | < | <= | >= | > | + | - | or
     | * | div | mod | and

<term> ::= <factor> | - <factor> | not <factor>

<factor> ::= <variable> | <unsigned constant> |
     <expression> | <function designator>

<unsigned constant> ::= <unsigned number>

<function designator> ::= <identifier> (
     <parameter list> )
```