

OREGON STATE  
DEPARTMENT OF COMPUTER SCIENCE  
OREGON STATE UNIVERSITY  
CORVALLIS, OREGON 97331-3902

UNIVERSITY

COMPUTER

SCIENCE

DEPARTMENT

Graphical Editor for DataLab (GEDL)

A Graphical Editor For Specifying And  
Synthesizing Abstract Data Types

Hae-sung Kim

Dr. T. G. Lewis

Department of Computer Science

Oregon State University

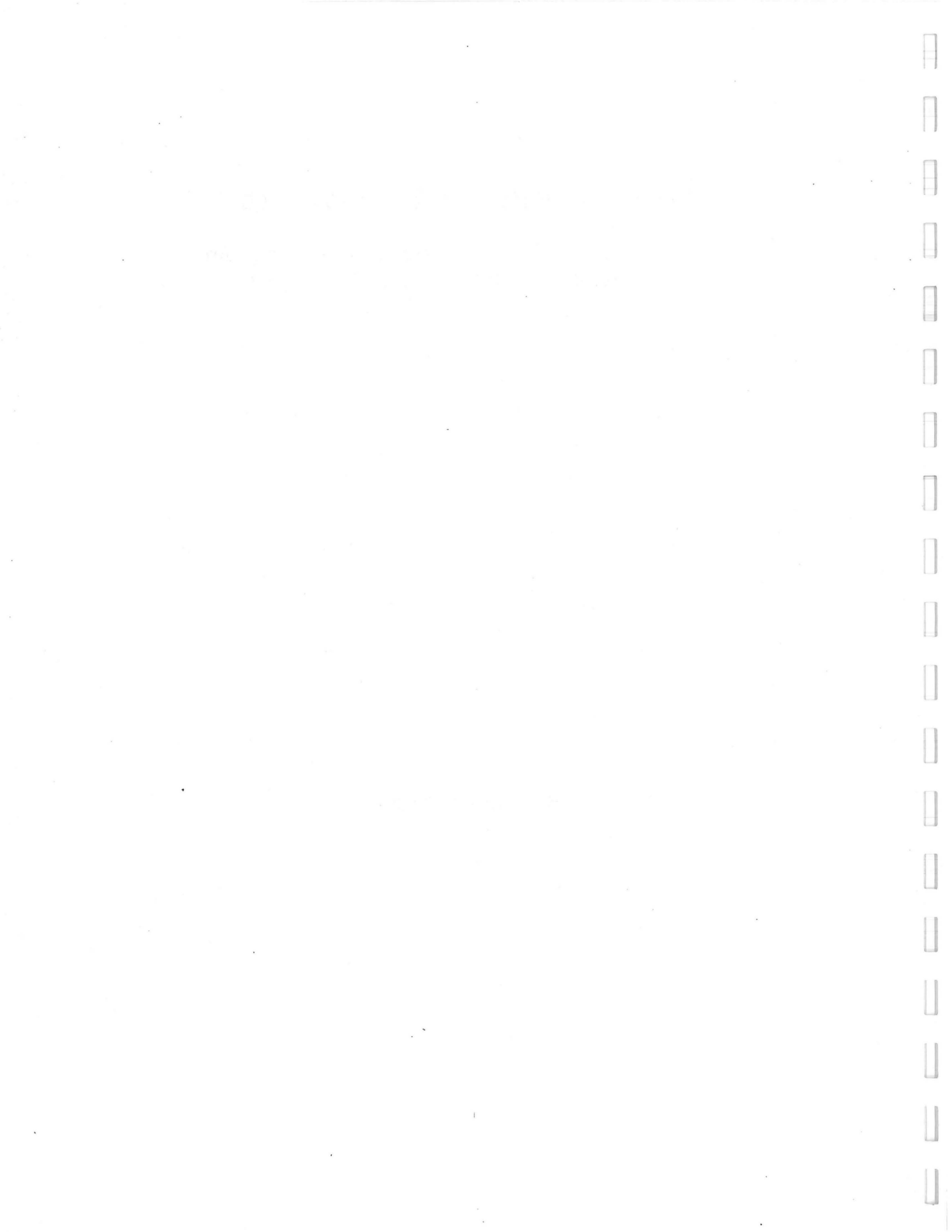
Corvallis, OR 97331-3902

90-60-1

Graphical Editor for DataLab (GEDL)

A Graphical Editor For Specifying And  
Synthesizing Abstract Data Types

Hae-sung Kim



Graphical Editor for DataLab (GEDL)

A Graphical Editor For Specifying  
And Synthesizing Abstract Data Types

by

Hae-sung Kim

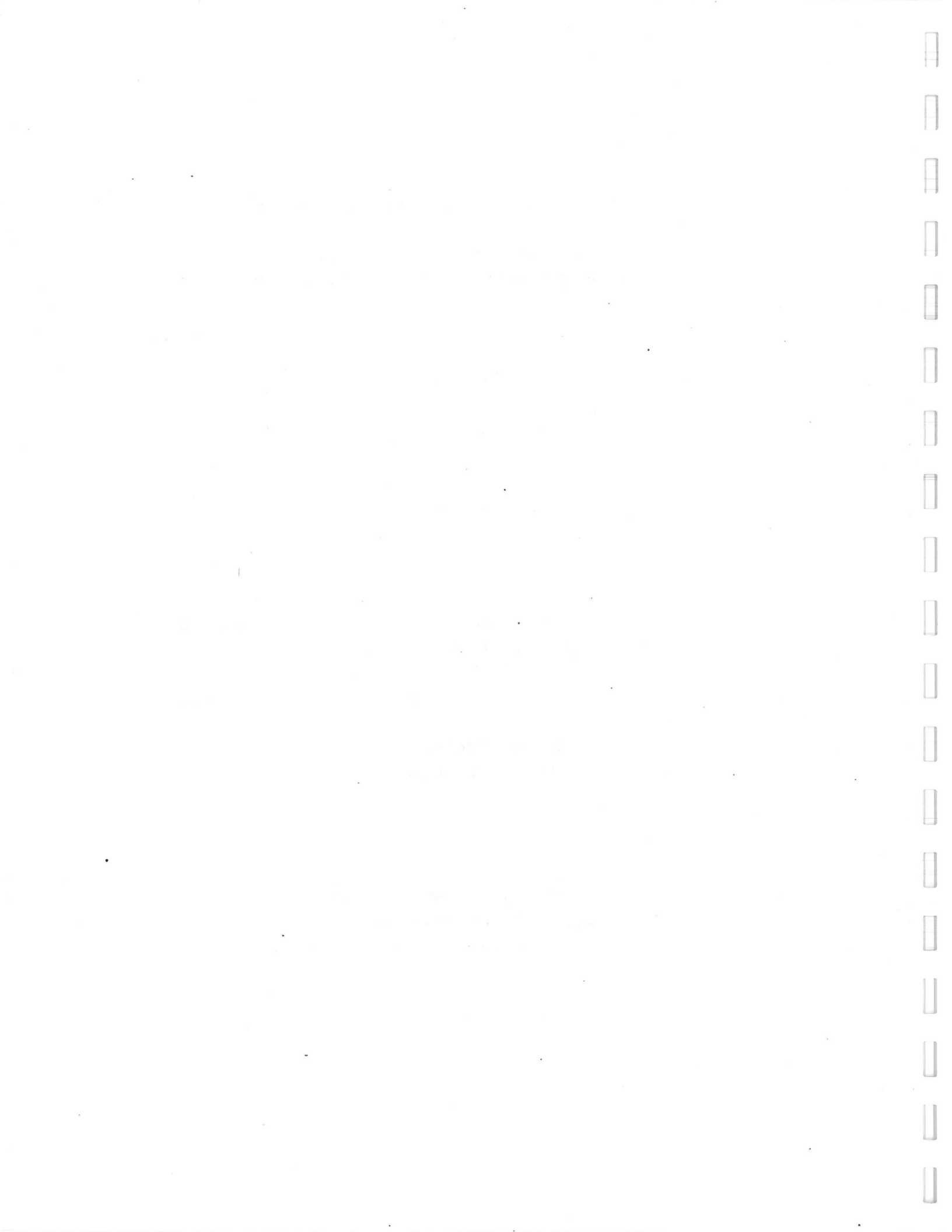
A research project submitted in partial fulfillment of  
the degree of Master of Science

Major Professor

Dr. T. G. Lewis

Department of Computer Science  
Oregon State University  
Corvallis, Oregon

December 1, 1989

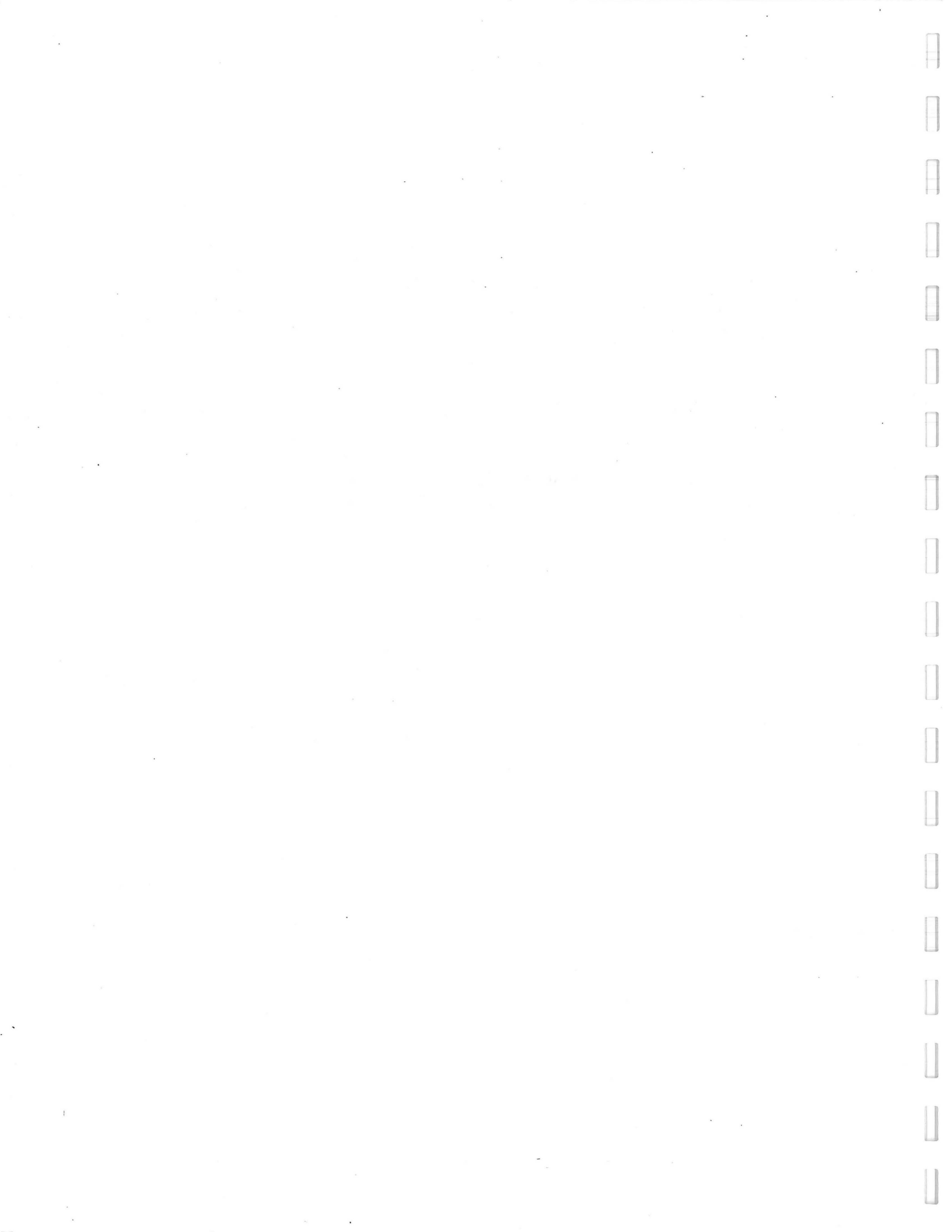


## Acknowledgements

I am grateful to my advisor, Professor Ted Lewis, for his guidance, understanding, helpfulness and encouragement. His experience and assistance have been instrumental at innumerable points in the progression of this project.

I thank Muhammed Al-Mulhem ,who is in charge of the code generation part of DataLab project, for his cooperations.

Also I would like to thank my parents for their support, as well as my wife, Hee-chan, and my son, Chang-whan, for their patience, help and understanding.



## Table of Contents

<b>Abstract</b>	-----	1
<b>1.0 Introduction</b>		
1.1 O.S.U. project	-----	2
1.2 GEDL(Graphical Editor for DataLab) and O.S.U.	-----	2
<b>2.0 Background</b>		
2.1 Abstract Data Type and Object Oriented Design	-----	3
2.2 Visual Programming	-----	4
2.3 Statement of The Problem	-----	5
2.4 Approach	-----	6
<b>3.0 Graphical Editor for DataLab (GEDL)</b>		
3.1 What is GEDL?	-----	7
3.2 Graphical Language Syntax	-----	8
3.3 Semantics of User Action	-----	14
<b>4.0 Using GEDL</b>		
4.1 Menu Reference	-----	17
4.2 Tutorial (Binary tree ADT example)	-----	21
<b>5.0 Implementation</b>		
5.1 Design Considerations	-----	31
5.2 Data Structure	-----	33
5.3 Possible Extensions	-----	38
5.4 Requirements and Limitations	-----	38
5.5 Application Statistics	-----	38
<b>6.0 Summary</b>	-----	39
<b>7.0 Bibliography</b>	-----	40
<b>8.0 Appendices</b>		
8.1 Unit Frame Syntax	-----	41
8.2 UNT_TYPE_DSD Listing	-----	42
8.3 GEDL Data File Format	-----	45
8.4 Data File Example (BIN_traverse)	-----	46



## List of Figures

Fig. 3.1	An object in the window	-----	8
Fig. 3.2	Two representations of the pointer	-----	9
Fig. 3.3	Assigning the "next" field of "node" to nil	-----	10
Fig. 3.4	Pointer assignment methods	-----	11
Fig. 3.5	"Don't Care" object	-----	11
Fig. 3.6	Object referred by "tree" can be nil or non-nil	-----	11
Fig. 3.7	Constant objects	-----	12
Fig. 3.8	Case "start" symbol	-----	12
Fig. 3.9	"transform" symbol	-----	12
Fig. 3.10	"return" symbol	-----	13
Fig. 3.11	"loop" symbol	-----	13
Fig. 3.12	"Exp   Stmt" symbol	-----	13
Fig. 3.13	Tools in the Palette	-----	14
Fig. 3.14	Object information dialog	-----	15
Fig. 4.1	File menu	-----	17
Fig. 4.2	Edit menu	-----	18
Fig. 4.3	Window menu	-----	19
Fig. 4.4	Operation display dialog	-----	21
Fig. 4.5	Interface part of the binary tree example	-----	22
Fig. 4.6	Parsing Error diagnosis	-----	23
Fig. 4.7	Case sequence dialog and a start symbol	-----	24
Fig. 4.8	Object kind dialog	-----	25
Fig. 4.9	Object Name Dialog	-----	25
Fig. 4.10	Type list dialog	-----	26
Fig. 4.11	A pointer object	-----	26
Fig. 4.12	Automatic creation of a dynamic object	-----	27
Fig. 4.13	Placing "transform" icon	-----	28
Fig. 4.14	Expression/Statement dialog	-----	28
Fig. 4.15	Specification for the BIN-traverse procedure	-----	29
Fig. 5.1	Data structure of UNIT unit	-----	35
Fig. 5.2	UNT_objectRecord list	-----	36
Fig. 5.3	The top-level DFD of GEDL	-----	37

## Abstract

This report describes the Graphical Editor part of DataLab, a visual programming tool for the specification and synthesis of abstract data types (ADTs). DataLab consists of two major components: graphical editor and source code generator. The graphical editor is used to design an abstract data structure and its operations by direct manipulation of data objects. The code generator uses the graphical editor's internal data structure to generate target source code.

DataLab is powerful enough to specify most elementary and intermediate data structures. However, the main result of this work has been to understand the benefits and limitations of graphical (visual) programming within the narrow domain of ADT synthesis.

Experiments suggest that graphical programming is useful in some aspects of programming, and textual programming remains the most effective in other aspects.

## 1.0 Introduction

### 1.1 O.S.U. Project

Oregon Speedcode Universe(O.S.U.) is a software development system employing on-screen editing of standard user interface objects, prototyping, program generation, and automatic analysis tools which are typically used to accelerate the production of running applications. A programmer uses OSU to design and implement all user interface objects such as menus, windows, dialogs, and icons. These objects are then incorporated into an application-specific sequence which mimics the application during program development, and performs the desired operations of the application during program execution.

### 1.2 GEDL(Graphical Editor for DataLab) and O.S.U.

Experimental results suggest that the techniques employed by OSU can be used to develop 50-90% of an application without explicit programming, yielding productivity improvements of 2-10 fold.[Lewis 88]

However, OSU is currently limited in its functionality, although it is aimed at wide-spectrum prototyping. To gain wide-spectrum prototyping functionality, many domain-specific tools must be designed and implemented.

DataLab is one of several domain-specific software development accelerators for OSU. It generates an Abstract Data Type (called a "unit") by showing and modifying a data structure in graphical form.

## 2.0 Background

### 2.1 Abstract Data Type and Object Oriented Design

An abstract Data Type (ADT) is a user defined type with its operations. A classical example of an ADT is, for example, a stack which has pop, push, and testEmpty operations. ADT enforces modular design of software and information hiding, which are among the most powerful techniques for combating software complexity.

Object Oriented Design (OOD) is an approach to software design in which a system is decomposed by identifying data/function encapsulations called objects, and arranging these objects into a whole by defining messages which connect objects together. OOD starts from ADT (OOD is ADT plus inheritance). OOD emphasizes objects and their decomposition rather than functional or data structure decomposition. Like ADTs, objects contain or encapsulate both data and operation. OOD is a very powerful force for designing systems with high communicational cohesion, low coupling with its corresponding desirable information hiding, and ease of maintenance.

GEDL itself was implemented following OOD and it enforces the OOD concept by generating an ADT. GEDL's code generator produces a Pascal source code module (ADT) in the form of a unit.

- Unit is an ADT (similar to class in object oriented programming),
- The interface part of a unit is public,
- Implementation part contains private data and functions for the class,
- The uses clause provides an import mechanism for connecting ADTs together.

However, the ADT units synthesized by DataLab do not implement inheritance. Details on unit structure will be found in the appendix section.

## 2.2 Visual Programming

Programmers often encounter difficulties when they attempt to transform the human mind's multidimensional, visual, and often dynamic concept of a problem's solution into the one dimensional, textual, and static representation required by traditional programming languages.

A new approach to software creation involves languages and means of problem specification that dramatically increase our ability to express requirements to the computer. One approach is the creation of graphical languages like GEDL. Visual programming goes one step further than conventional text based languages by providing visualization of the software. This is claimed to improve the process of software development and maintenance.

Visual programming has a very short history and I believe some shortcomings need to be eliminated, through both refinement of visual programming concepts and hardware improvement. After designing and implementing GEDL, I realized the pros and cons of visual programming as follows.

### Pros:

- Visualization of the software -- excellent for maintenance,
- Intuitive and natural because of its direct simulation of human mind,
- Random access of any information in the screen -- easy to understand,
- Encoding information can be more compact in theoretical sense.

### Cons:

- Restrictions on displaying dialogs and graphical objects on the limited screen,
- Hard to change object property -- text based language has efficient tool for making changes (lexical analyzer and parser) ,but graphical language is actually a collection of internal data structures which represent graphical objects in the screen,

- Current methods, which just hold graphic information in memory as an internal data structure, are slow and take more memory space than conventional text based systems. We need to find a way of storing graphic images directly and restoring them by using a special graphic parser.

### 2.3 Statement of the Problem

The problems addressed by this research are :

- 1) How to represent data and the operations on the data in a graphical manner,
- 2) How to "edit" these representations, and efficiently store the editing information,
- 3) How best to use both text and graphics in combination such that programming is made "easier", "faster", and more "maintainable".

In addition, this research attempts to answer the following questions, but were not addressed by GEDL :

- 4) How is actual code synthesized from data and operations represented in graphical symbols?
- 5) How can "realistically large" and complex ADTs be automatically synthesized?

## 2.4 Approach

The approach to these problems was to :

- 1) Invent a new text/graphical language for describing both static and dynamic behavior of ADTs,
- 2) Implement a program (called GEDL) which incorporates direct manipulation of text and graphics in order to devise editing and storage techniques for the new language, and
- 3) Apply this new technology to the creation of many ADTs. The results of this empirical study should provide some early indications of the usefulness of the approach.

Accordingly, I have participated in the design and implementation of the graphical language for DataLab, and implemented a running prototype of the direct manipulation editor, called GEDL.

### 3.0 Graphical Editor for DataLab (GEDL)

#### 3.1 What is GEDL?

As its name implies, GEDL is a graphical editor for DataLab. GEDL lets a user create data objects and algorithms, which represents an operation in an ADT. DataLab generates target code by interpreting GEDL's internal data structure.

DataLab is one of several new visual programming systems, but its approach to creating software is different from most other systems. Simply speaking, a program is a data structure plus algorithm (or control). Most previous work in graphical programming emphasizes "control", such as V.I.P which models a conventional flow-chart.[Mainstay]

A modern software development tool should enforce good programming style to improve software maintainability. Visualization of the data object is very useful in development and maintenance, because most programming languages support dynamic data type objects (run-time allocation variable -- pointer type in Pascal) as well as static data type objects, such as simple type or array.

DataLab emphasizes visualization of data objects and modular design of the software. In fact, DataLab's syntax restricts the user's arbitrary programming style, enforcing encapsulation and modularity. DataLab uses "control", of course, but it is much more abstract and declarative than conventional flow-chart-like control.

The theory of program transformation is used for control flow -- "condition/action" transformation. The programmer simply draws several "condition" and "action" cases, then the system generates an "operation" (function or procedure) for the module (ADT).



### 3.2 Graphical Language Syntax

Before I explain the language syntax, I want to acknowledge that "Transformation", "Loop", and "Return" constructs which form "Condition/Action transformation", and "Don't care object" are incorporated by Muhammed Al-Mulhem. The precise semantics of DataLab language will be presented in his paper, "DataLab: A Graphical System for Specifying and Synthesizing ADT's". [Al-Mulhem 89]

There are five categories of graphical objects in GEDL : objects, pointers, constants, controls, and expressions/statements.

#### 3.3.1 Objects

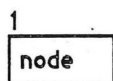


Fig 3.1 An object in the window

This object symbol represents any non-pointer variable or parameter. An object can be considered as a five-tuple :

object = (name, type, kind, sequence, object ID)

The real data structure of an object in the window has the additional graphical information.

Name-- The object name is defined by the user and can be any legal Pascal identifier. The name is displayed inside the graphical representation and is truncated if it does not fit. The object name can be changed if the object is a static variable or parameter, by double clicking on the object and clicking on "Change name" button.

Type-- The object type is defined by the user by either selecting a predefined or user defined type from the type list of the "type display dialog". Type information will be displayed by double

clicking on the object.

Kind-- The object kind can be a local variable, global variable, or procedure/function parameter. The user specifies the kind by selecting a radio button which shows object kind from the "object kind dialog". The object kind will be displayed by double clicking on the object.

Sequence-- The object sequence (or order) number corresponds to the creation order of the object in the "Condition/Action" transformation. This number is created automatically by GEDL and is displayed as part of the graphical representation of the object.

Object ID-- The object's unique ID number (OID) will be assigned to each object when it is created. The OID of any object is unique for each file or internal data structure, so that GEDL could find user selected object and update all object drawings.

### 3.3.2 Pointers

Pointers are represented graphically as follows :

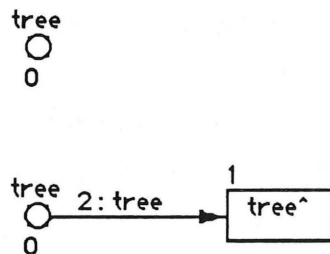


Fig. 3.2 Two representations of the pointer

Initially, there is no "arrow", only the circle. The circle has the same meaning as the object's rectangle on the screen -- it represents a memory location for an address or dereferenced space. The user can establish pointer by dragging the circle. The arrow is the conventional graphic way of representing "dereference space".

A pointer object also can be considered as a five-tuple,

```
pointer = (name, type, kind, sequence, dereference Object ID)
```

The dereference OID stores the OID of the pointer's dereferenced object.

This is a good place to show how to assign a pointer. Consider a record "node" of type "nodeRecord" as follows :

```
nodePtr = ^nodeRecord;
nodeRecord = record
    data : integer;
    next : nodePtr;
end;
```

Assume that we want to set the "next" field of "node" to nil, which is represented in Fig. 3.3.

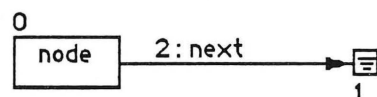


Fig. 3.3 Assigning the "next" field of "node" to nil

Note that the object "nil" is created before dragging the pointer "next" out of "node". This is indicated by the sequence number "1" for the nil object and "2" for the "next" field.

Now consider a pointer "list," shown in Fig.3.4. "list" can be set to point to an existing object of compatible type, in this case, "node". GEDL does type checking so that a pointer must assign to compatible type of an existing object. Assigning "next field of node" to a default object (node.next^)^ is done by dragging the pointer out of "node" to an empty location on the screen. GEDL will automatically create an object of compatible type and make "node" point to it, as shown in Fig. 3.4.

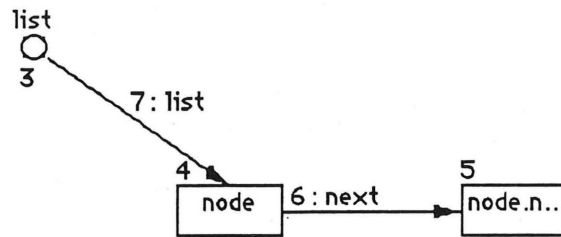


Fig. 3.4 Pointer assignment methods

A special object that is related to pointers is the "Don't Care" object, which is shown in Fig. 3.5.

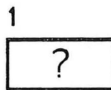


Fig. 3.5 "Don't Care" object

This object when associated with a pointer, represents a "Don't Care" instance of that pointer. A "Don't Care" instance means the pointer can be either "nil or non-nil". The pointer "tree" in Fig. 3.6, can be either nil or non-nil.



Fig. 3.6 Object referred by "tree" can be nil or non-nil

### 3.3.3 Constants

This category includes three constant objects : the "nil", and boolean "true" and "false". These objects have only a sequence number and do not have types, kinds or names. They are shown in Fig. 3.7.

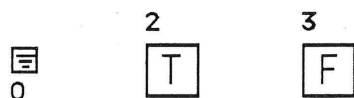


Fig. 3.7 Constant objects

### 3.3.4 Controls

There are four controls used to build transformations. They do not have names, types or kinds; they have only a sequence number.

The first control is "start". This has a "case" sequence number, which has a different meaning from the object sequence number.



Fig. 3.8 Case "start" symbol

The next control symbol is "transform". It separates the "Condition" part from the "Action" part of a transformation.



Fig. 3.9 "transform" symbol

The next control symbol is "return". It returns a single value from a function. The returned value is represented graphically in the "Action" part of the transformation.

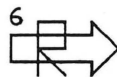


Fig. 3.10 "return" symbol

The last control symbol is "loop" which is used to construct a loop transformation.



Fig. 3.11 "loop" symbol

### 3.3.5 Statement and Expressions

These are represented by the "expression/statement" symbol. It allows the user to define any legal Pascal statements or expressions that include :

- Procedure calls, including recursive calls, e.g. `compare(s1,s2);`.
- Assignments, e.g. `x := 6;`
- Relational expressions, e.g. `tree <> nil;`.
- Input/Output statements, e.g. `writeln('Current Node :', tree^.data)`

The specified expression or statement is displayed inside this symbol if it fits, otherwise it is truncated to fit. For example, the statement `writeln('hi');` is represented as shown in Fig. 3.3.5.

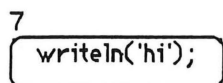


Fig. 3.12 "Exp | Stmt" symbol

### 3.3 Semantics of User Action

The macintosh user interface uses "clicking and dragging". GEDL takes full advantage of this simple-yet-powerful method, by careful observation of mapping between the user action and the semantics of the computer language.

The meaning of a user action is given by the meaning of the current "tool selection". The palette (or Tool window) is shown in Fig. 3.13.












Tool		
1. Selection arrow		2. Start icon
3. Pointer Drag arrow		4. Object icon
5. Transformation icon		6. Don't care object icon
7. Loop icon		8. NIL object icon
9. Return icon		10. TRUE object icon
11. Expression/Statement		12. FALSE object icon
		
		
		
		
		

Fig. 3.13 Tools in the Palette

-- Click on a graphical object with "selection arrow":

Select a graphical object. The selected object will be highlighted and the item(s) of the "Edit" menu will be activated or inactivated according to object kind. For example, if the user selects a "start" object, "Clear" and "Copy" under the "Edit" menu will be activated. Note that "Clear" just deletes graphical object(s) from the screen, but does not mean "algorithmic delete operation".

-- Double click on graphical object with "selection arrow":

Invoke a dialog which displays the object's properties (name, type, and scope) for variables or parameters (Fig. 3.14). For statement/expression, it shows the full description for browsing or modification.

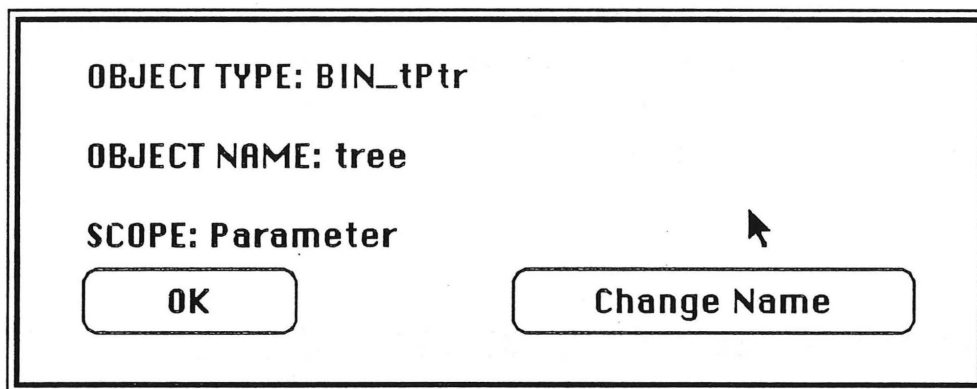


Fig. 3.14 Object information dialog

-- Click on vacant space with any tool but "selection arrow" or "pointer drag arrow":

This action creates a graphical object and appends it to the object list of the internal data structure. If the user selected "object" tool from the palette, consequent dialogs ask the user to enter object name, to select object type and scope. This is equivalent to the "variable declaration" in Pascal syntax, such as "var fooPtr: Pointer;". At this point, if the object has any pointer field(s) then the pointer field(s) are instantiate as well.



-- Drag from an object to another object with "pointer drag arrow":

Pointers can only be dragged out of objects which contain pointer fields. The dereferenced object type should be compatible with the pointer, or NIL, or "Don't care object". Note that GEDL performs type checking between pointer type and dereferenced object to prevent the user from incorrect pointer assignment.

Other than those conditions, the pointer (a line with arrow head) will not be drawn. This is equivalent to "pointer assignment" of Pascal syntax, such as "aPointer := @aPointedObject;".

-- Drag from an object to vacant space with "pointer drag arrow":

This action creates a default dynamic object which has the same type as the dragged pointer, and assigns the pointer to the automatically created dynamic object. Equivalent Pascal syntax of this action will be;

```
aPointer := Object-type(New(size of(dereferenced object)));
```

```
{Create a dynamic object and type cast to object type}  
{and aPointer is assigned to newly created object}
```

## 4.0 Using GEDL

### 4.1 Menu Reference

#### 4.1.1 Apple Menu:

About GEDL... : Display the author name and GEDL version number.

#### 4.1.2 File Menu:

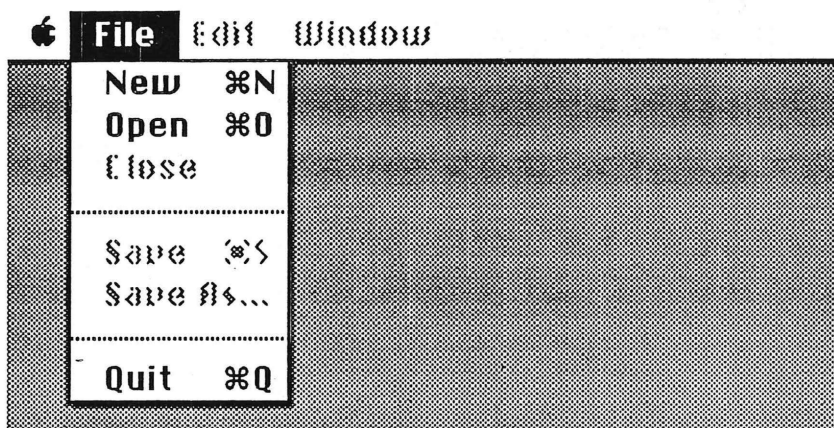


Fig. 4.1 File menu

**New** : Create a new "unit window" to specify its types and operations (function or procedures). Syntax of unit is the same as LightSpeed Pascal's unit syntax, except implementation part's operations only have their names like interface part's operations. Details on unit syntax will be shown in appendix.

**Open** : Open an existing GEDL file.

**Close** : Close the current GEDL file.

**Save/Save As...** : Save current state of GEDL's data into a file.

**Quit** : Quit GEDL application and to back to finder.

\*\*\* Note that unlike common applications such as text editor, the GEDL data file includes the unit frame window, all local type windows and operation windows. Opening or closing a file is not related to opening or closing such windows. Showing each window is handled by the "Window" menu and the user must click on the "GoAway box" to close the window.

#### 4.1.3 Edit Menu:

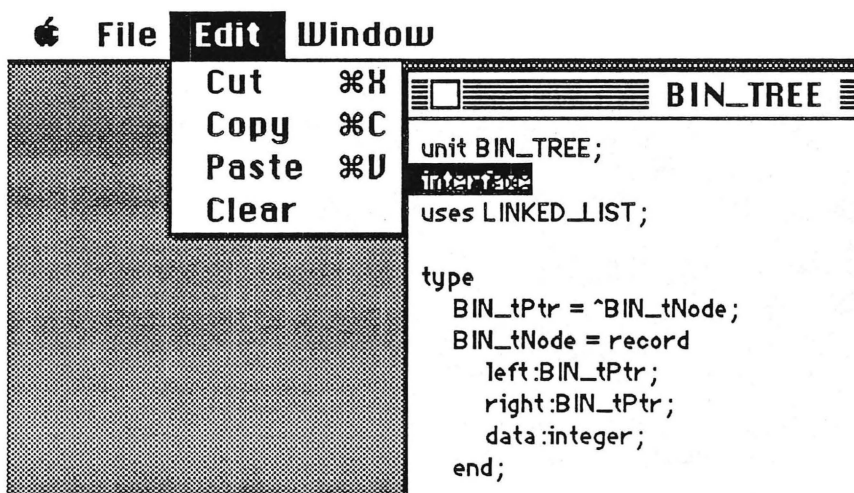


Fig. 4.2 Edit menu

Copy, Cut, Paste, and Clear: These are standard editing features for text windows (unit window and local type window).

For the graphic window (operation window) only "Clear" and "Copy" will be activated depending on which object is selected.

Copy : Copy all objects of the "Condition" part and paste them right after the "transform symbol". This will be enabled only if there exist a transform symbol in the case, and the user selected a "start symbol".

Clear : Delete graphical object(s) including pointers. Enabled only if the user selected an object, which is:

- The last object the user created,
- Transformation object. Delete all objects from transformation to the last object within a case,
- Start object. Delete selected case.

#### 4.1.4 Window Menu:

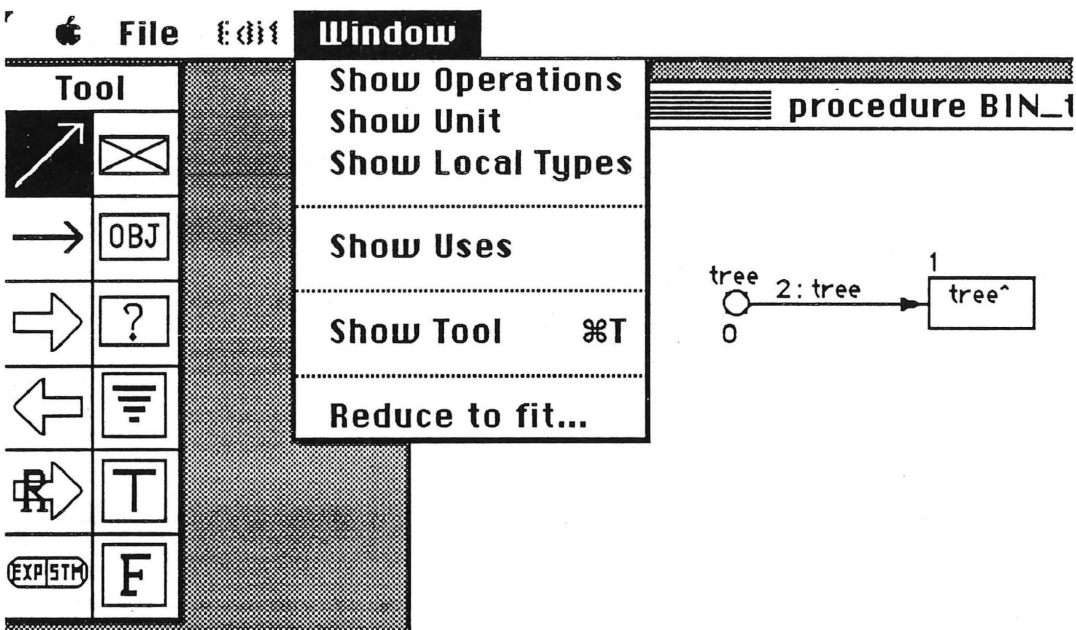


Fig. 4.3 Window menu

Show Operations : Put up a dialog which contains the list of operations (function or procedures) to be chosen by the user. An operation window will be shown if user choose an operation from this dialog. If there is no operation window, then the tool window (or palette) will be shown with the operation window.

Show unit : Show the unit frame window for referencing or modifying unit.

Show Local types : Show currently active operation window's local types for reference or modifying local type.

Show Uses : Show only interface parts of other GEDL files, which are included in uses clause of current file, for reference. Uses window is just for browsing and it can not be modified.

Show Tool : Make the tool window visible. Used when the tool window is covered by other windows.

Reduce to fit... : Miniaturize current operation window's content. Click on the content of reduced window to close.

## 4.2 Tutorial (Binary tree ADT example)

An example, binary tree ADT, will be presented to illustrate DataLab's operations. The GEDL has a palette, as shown above Fig. 4.2.1, which contains a variety of icons to define data structures, constants, statements, expressions, and control structures. This binary tree example is based on the paper "DataLab: A graphical system for specifying and Synthesizing ADT" by Al-Mulhem and Lewis. Users who are more interested in the theory of DataLab graphical language syntax and semantics should read this paper.[Al-Mulhem 89]

The ADT's interface part is defined textually while its operations are defined graphically. To define an ADT operation, the user selects an operation window from the "Show operation" menu item under the menu "Window" and chooses an operation from the "Operation display dialog" shown in Fig.4.4.

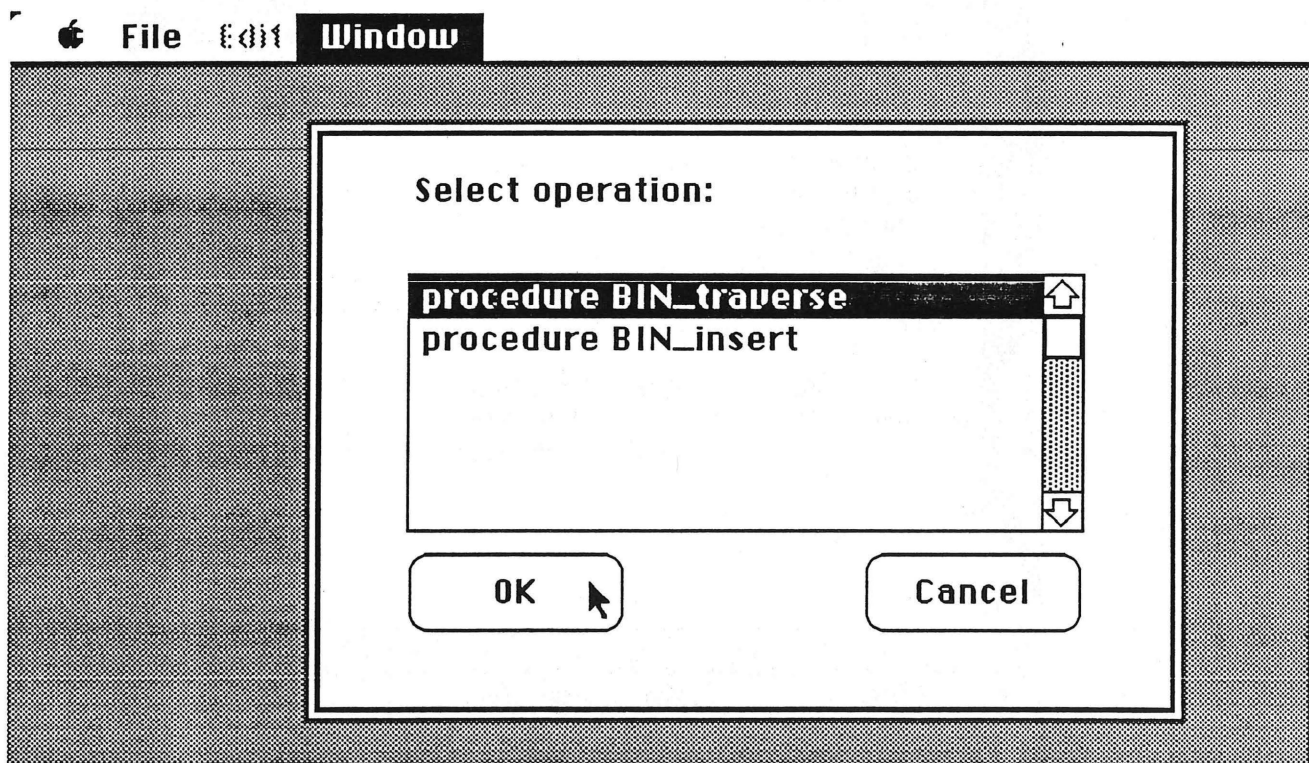


Fig. 4.4 Operation display dialog

Now you have a window where you can define the operation (procedure or function) through a set of algorithmic examples. These examples are built by selecting graphical icons from a palette and placing them on the screen. The graphical representations are mapped into an internal representation which is used by the source generator to generate Pascal modules.

A binary tree module is defined as follows. First, the interface part is entered in textually using the "unit frame" window which is invoked by selecting the "New" menu item under the "File" menu. The interface part of a binary tree is shown in Fig. 4.5.

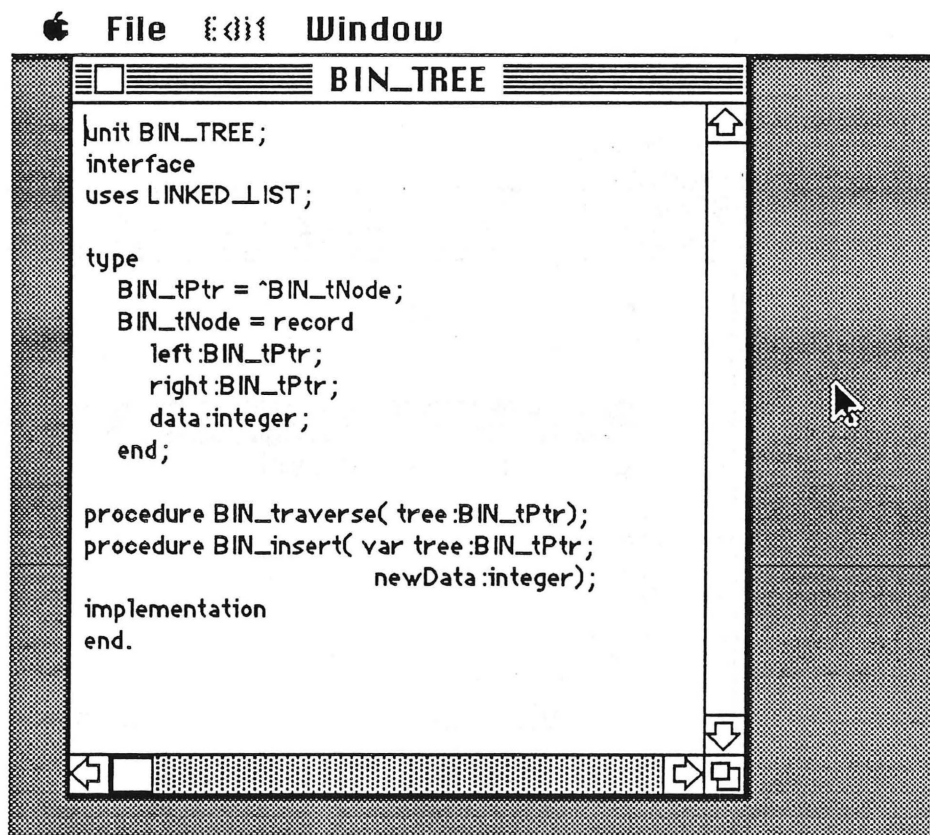


Fig. 4.5 Interface part of the binary tree example

After entering all interface specifications as above, close the unit frame window by clicking on the "GoAway" box for the window. The interface part is then parsed to generate the specified type objects, which include externally defined types (uses clause types) if any, and the list of operations( procedure/ functions) which will be used as separate operation windows. The parser checks unit syntax so that if there is any syntax error, it stops parsing and reports the error diagnosis with the line number (Fig.4.6) and does not close the unit frame window.

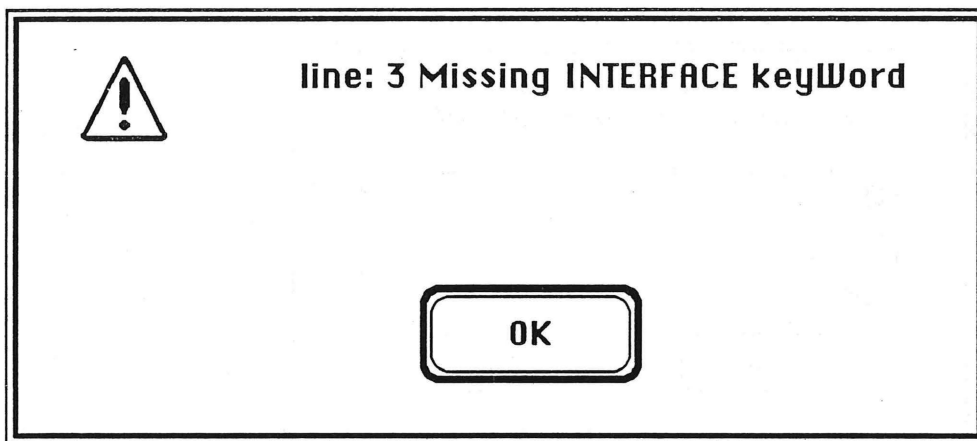


Fig. 4.6 Parsing Error diagnosis

The operations are defined next. Let us define the operation "BIN\_traverse" first. The user selects "Show operation" under the "Window" menu, then the operation list dialog will be shown for the user to select the procedure names. Select "BIN\_traverse" from the operations list, then GEDL opens an operation window with window title "BIN\_traverse". If this is the first operation window, GEDL automatically shows the palette too. If the user selects an operation which is already open, GEDL simply brings this operation window to the front. This is convenient when an operation window is completely covered by other windows



Now we have the empty window named by "BIN\_traverse", and let's make the BIN\_traverse procedure as follows.

The procedure "BIN\_traverse" is defined by describing its behavior as a set of "Condition/Action" transformations. Each transformation must start with the "start" icon, which represents its beginning. The first transformation for "BIN\_traverse" is created by selecting the "start" icon from the palette and placing it on the screen, then we see a dialog which asks "case order" (Fig. 4.7). Type "0" and click OK button and you will see a start icon on the screen.

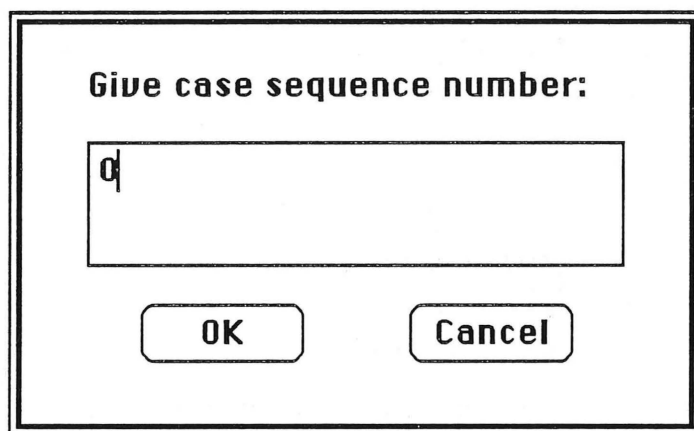
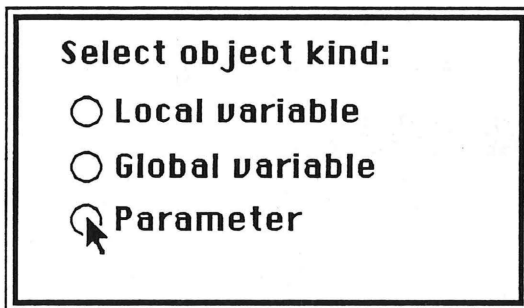


Fig. 4.7 Case sequence dialog and a start symbol

Next, the transformation's condition (the "Condition" part) is defined. The condition that we want to represent is the procedure's parameter "tree" being not nil. To represent this first the "object" icon is selected from the palette and placed on the screen. Then DataLab will ask the user to specify its kind from the "object kind dialog" (Fig. 4.8), give a name to the object (Fig. 4.9), and specify its type by selecting from the "type list dialog" (Fig. 4.10).

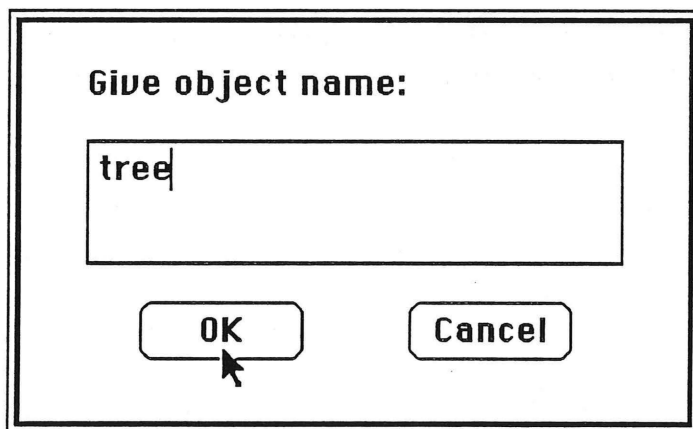


**Select object kind:**

- Local variable
- Global variable
- Parameter

A dialog box with a double-line border. The title is "Select object kind:". Below the title are three radio button options: "Local variable", "Global variable", and "Parameter". The "Parameter" option is selected, indicated by a mouse cursor arrow pointing to the radio button.

Fig. 4.8 Object kind dialog



**Give object name:**

tree

OK Cancel

A dialog box with a double-line border. The title is "Give object name:". Below the title is a text input field containing the text "tree". At the bottom of the dialog are two buttons: "OK" and "Cancel". A mouse cursor arrow is pointing to the "OK" button.

Fig. 4.9 Object Name Dialog

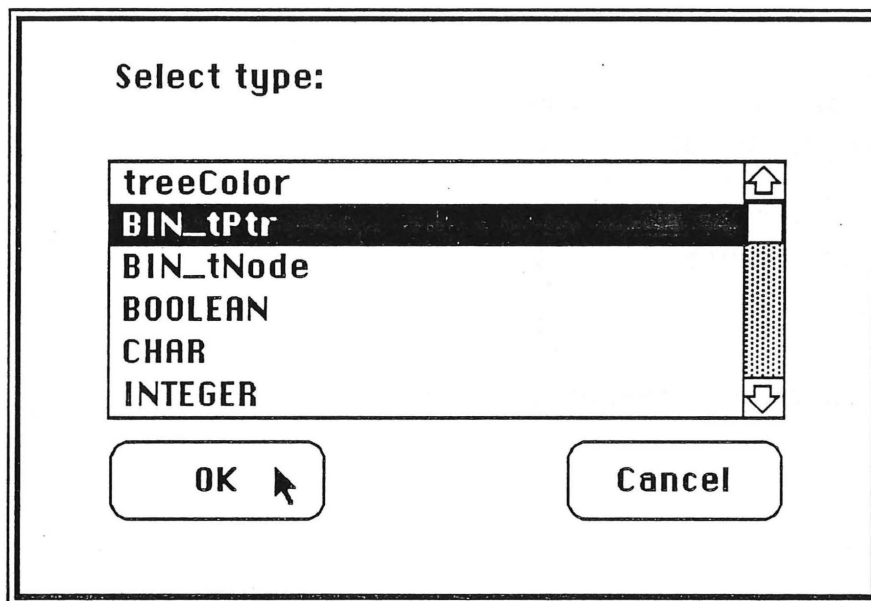


Fig. 4.10 Type list dialog

We specify its kind as parameter, choose the name, "tree", and specify its type as "BIN\_tPtr". "tree" is then displayed as a small circle, which represents "pointer object" :



Fig. 4.11 A pointer object

The integer number "0" below the "tree" pointer object is a sequence number that shows the order in which objects are created, this provides the sequence of the algorithm as well as the visual effect (later the user can see what he/she did by tracing these numbers).

The "Start" symbol has the sequence number that was provided by the user through the dialog (Fig. 4.7) instead of an automatically generated

number like those used for other graphical objects. The order of the "start" icon has a different meaning from other graphical objects' sequence orders. The start symbol means "here we start a new case of algorithm description", and sometimes the order among the cases is very important. For example, if we have three cases;

```

if (condition) then begin ----- end
else if (condition) begin ----- end
else begin ----- end;

```

The order of the conditions ("Condition" part of cases) is critical in forming an algorithm. Thus GEDL provides flexibility in representing case order.

To sum up, the start icon's order affects the operation's algorithm while any other objects' order applies only within the scope of one case.

Next we want to show that "tree" points to a non-nil object; this is done by dragging the pointer out of the "tree" pointer object, using the "pointer drag arrow" tool. GEDL then automatically creates a non-nil object of compatible type and assigns it the name "tree^". Our transformation now looks like this :

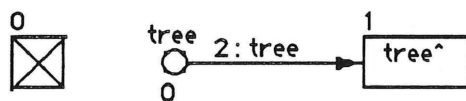


Fig. 4.12 Automatic creation of a dynamic object

This completes our specifications of the "Condition" part of the transformation, which represents the condition "tree is not nil".

Now we need to define the "Action" part. This is done by first selecting the "transform" icon from the palette and placing it on the screen. The transformation now looks like this :

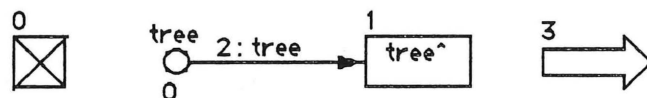


Fig. 4.13 Placing "transform" icon

Next we specify the actions of the "Action" part that need to be performed if the "Condition" part is true. For the "BIN\_traverse" procedure there are three actions that need to be done if "tree" is not nil. These actions are (1) printing the value of the node, (2) calling "BIN\_traverse" recursively on the left subtree, and (3) calling "BIN\_traverse" recursively on the right subtree, for the pre-order tree traversal. Each of these actions is specified by selecting the "Exp | Stmt" icon from the palette. When this icon is selected, GEDL asks the user to type any legal Pascal expression or statement textually (Fig. 4.14).

**Give statement/expression:**

```
writeln(tree^.data);
```

OK

Cancel

Fig. 4.14 Expression/Statement dialog

After specifying the statement, it is displayed inside the icon as long as it fits; otherwise it is truncated to fit. Specifying the three actions as "writeln(tree^.data)", "BIN\_traverse(tree^.left)", and "BIN\_traverse(tree^.right)" would make our transformation appear as shown in Fig. 4.15.

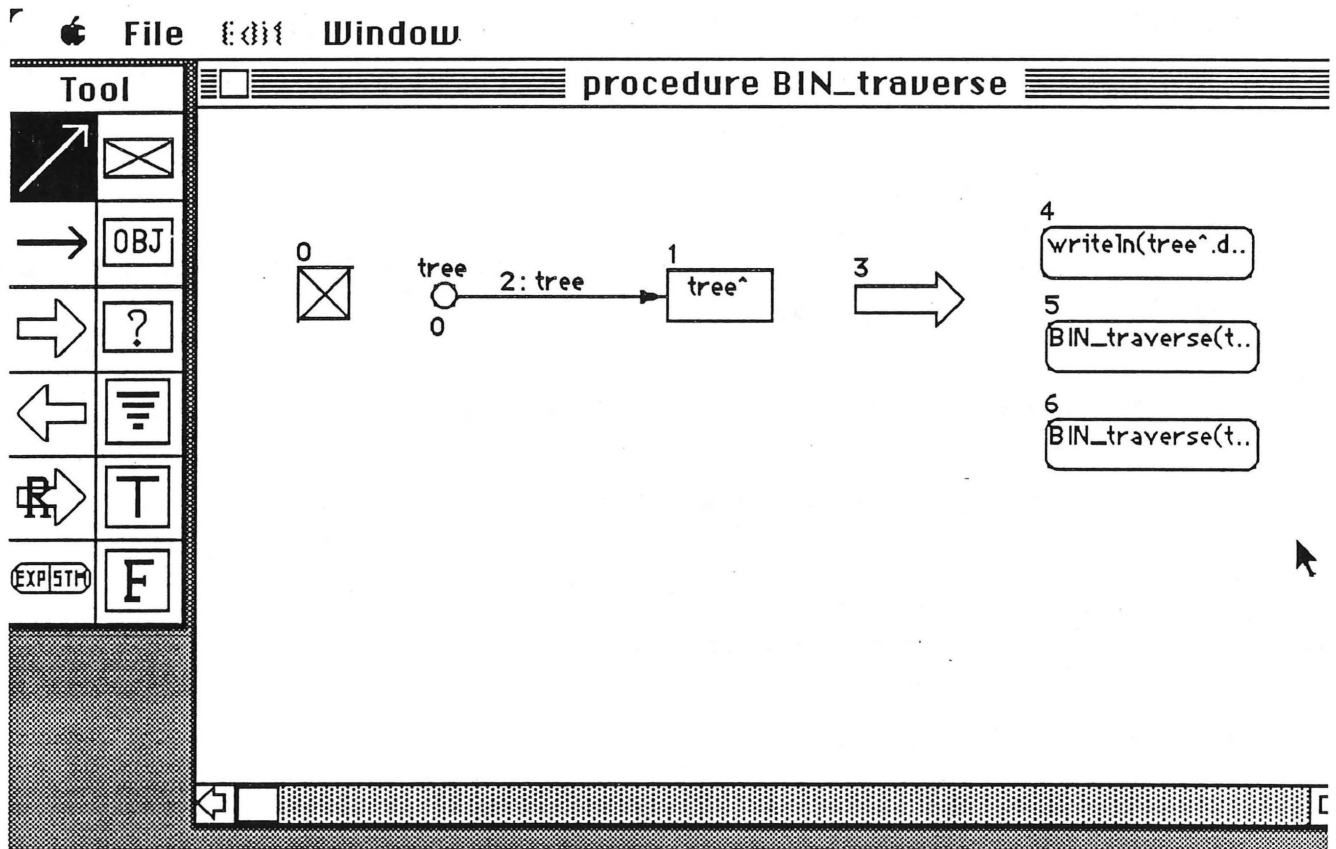


Fig. 4.15 Specification for the BIN-traverse procedure

This single transformation is all we need to specify the "BIN\_traverse" procedure, because, if the tree "tree" is empty, BIN\_traverse returns nothing. The semantics of this transformation is as follows :

Condition --"tree" has at least one node, i.e. it is not empty. This is represented by making the "tree" pointer point to the non-nil object, "tree^".

Action -- Do three actions : print the value of the current "tree" node, call "BIN\_traverse" recursively on the left subtree, and finally call "BIN\_traverse" recursively on the right subtree.

This report does not cover the "code generation" part of DataLab, but the "BIN\_traverse" example will generate Pascal code as follows:

```
procedure BIN_traverse( tree: BIN_tPtr );  
begin  
    if (tree <> nil) then begin  
        writeln(tree^.data);  
        BIN_traverse(tree^.left);  
        BIN_traverse(tree^.right);  
    end; {if}  
end; {BIN_traverse}
```

## 5.0 Implementation

### 5.1 Design Considerations

The hardest thing in designing software is making "trade-offs", and designing/implementing GEDL was no exception. Some major design considerations for GEDL are as follows:

- Easy to use:

GEDL follows Macintosh's standard user interface conventions so that menus and dialogs are intuitive and natural.

- Portability:

GEDL does not produce executable code, instead it generates target source code (currently Pascal code) to provide flexibility and ease of optimization. By using text description of "unit frame" and "local type or constant information", it should be easy to change the "target language" without rewriting most of the GEDL source code -- the only changes required are in the parser section.

- Combination of graphics and text:

This is related to ensuring easy-to-use and portability as mentioned above, but it has more meaning. We do not believe graphics is the panacea, but both graphics and text have their pros and cons, and we tried to use them in appropriate places. Imagine, if we try to visualize all data object's properties and operations. For example, parameters of the procedure, and function return values are not easy to represent in graphical forms, and further there is no common interpretation of those situations. GEDL tried to use the graphical representation where it is most common, undoubtedly this is the "pointer". We also believe that "simplicity" is the most important factor of a successful user interface design and as a consequence, all objects are represented as simple box shapes, and if an object has pointer(s) then it can be dragged out to connect with other objects. Statements and expressions are represented as "expression/statement" boxes and the user can see or change their content by double clicking on them.



- Object Oriented Design of GEDL:

GEDL is a good example to demonstrate the power of OOD. It took more time to define module (or class) and design inter-module dependencies, but this time was compensated by ease of implementation, finding bugs, and changing/adding operations in the module. The source code is much more readable and maintainable, and size was reduced. Also by extensive use of functions/procedures for accessing public part data structures -- types which are in the interface part of the unit, I could minimize side-effects and maximize information hiding. Actually there are no public global variables (interface part variable in unit) in GEDL.

- Data file format:

I used a text file instead of a binary file to store the application's data. One of the main reasons to choose the text file format is that the user can examine his/her saved internal data structure of what he/she did by using a simple text editor. It was also very helpful during development and debugging of GEDL. Another reason is that by using text format I can maintain a single data file which includes both plain text information and formatted object information. This is very hard if I use a binary file for text information. File format is shown in appendix 8.3.

## 5.2 Data Structure

GEDL project consists of 15 units as follows: For historical reasons, all files have the extension, "DSD", which stands for Data Structure Designer.

1) GLOBAL\_DSD : It contains operations, which are used in most units, such as application finish flag, cursor shape changing routines.

2) DIALOG\_DSD : It contains all dialog or alert routines.

3) TYPE\_DSD : It contains all type object manipulation routines. Its "type structure" is used for creating objects with type.

4) LEXER\_DSD : It performs lexical analysis of the given text. Produces the token string and its token type.

5) PARSER\_DSD : It performs parsing on unit frame and local type/constants. Makes an operation list to be used in creating the unit record list in UNIT2\_DSD, and type objects.

6) DRAW\_DSD : It contains all object drawing routines.

7) UNT\_TYPE\_DSD : 7,8,9,10 units are actually one unit in functionality, but they are divided because of LightSpeed Pascal's size restriction on the unit (smaller than 32K). UNT\_TYPE\_DSD has only interface part types and procedures.

8) UNT\_GLOBAL\_DSD : Only has implementation part global variables.

9) UNIT1\_DSD, UNIT2\_DSD : They have all operations on UNT\_unitRecord list, objects and pointers, such as creating, deleting or selecting an object. More explanations on above UNIT data structures will be given.

10) WINDOW\_DSD : It has all window, which include drawing, text and browsing windows, manipulation routines.

11) FILE\_DSD : It contains all file manipulation routines, such as opening, closing, saving, reading and writing files.

12) MENU\_DSD : It has all menu operations.

13) EVENT\_DSD : It contains all event related routines for user actions, such as activate/deactivate windows, mouse event, key event and event main loop for the application.

14) DSD.pas : This is the main program, which calls initialization routines and enters the main event loop.

The most important data structure of GEDL is, of course, the UNIT data structure. Its data structure is shown in Fig. 5.1 and Fig. 5.2, which show data structure of the "BIN\_traverse". Note that there are some global variables that hold current state of GEDL as follows:

- unitHandle: Holds whole data structure of UNIT unit.
- currentPart: Indicates which UNT\_unitRecord is currently being used.
- currentObject: Indicate which object is selected.
- currentPointer: Indicate which pointer is selected.

More information on the object record can be found in appendix 8.2, "UNT\_TYPE\_DSD" listing.

The top-level data flow diagram of GEDL is shown in Fig.5.3

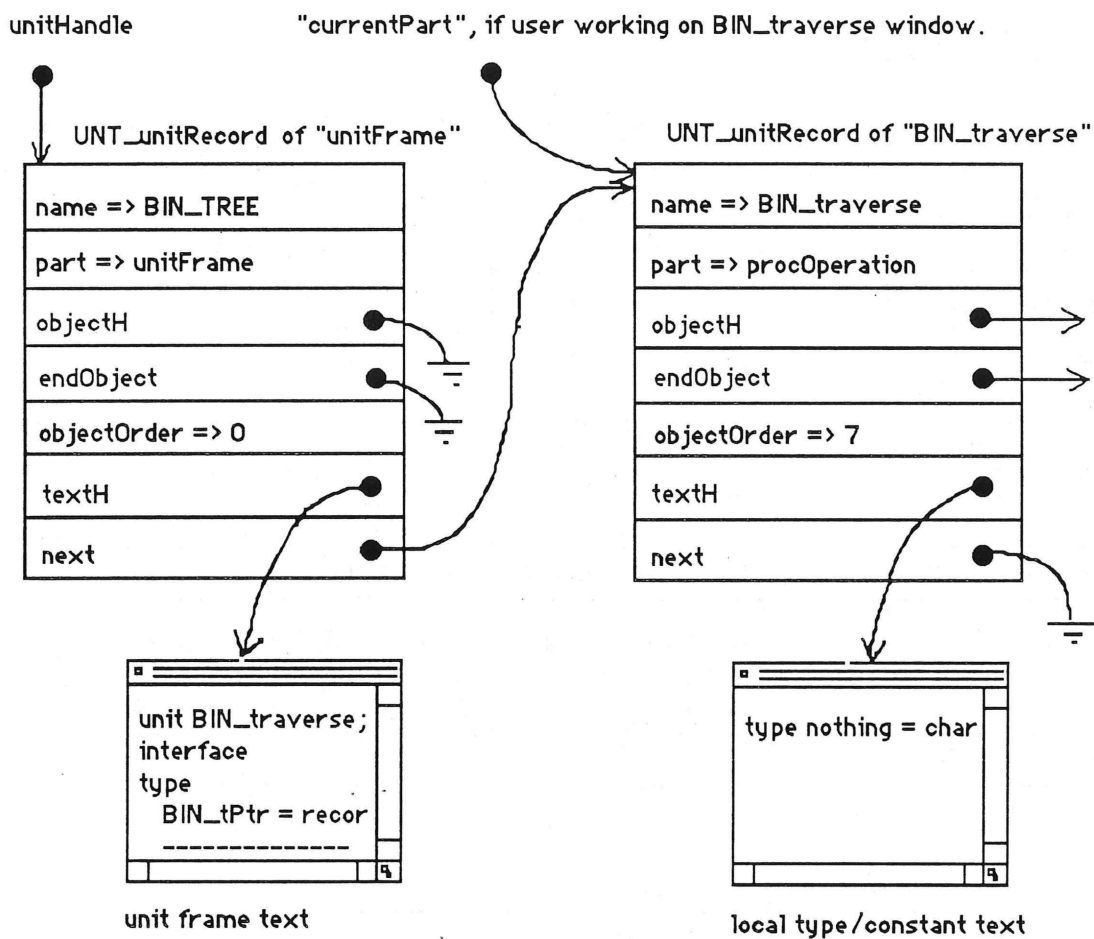


Fig. 5.1 Data structure of UNIT unit (continued in Fig.5.2)

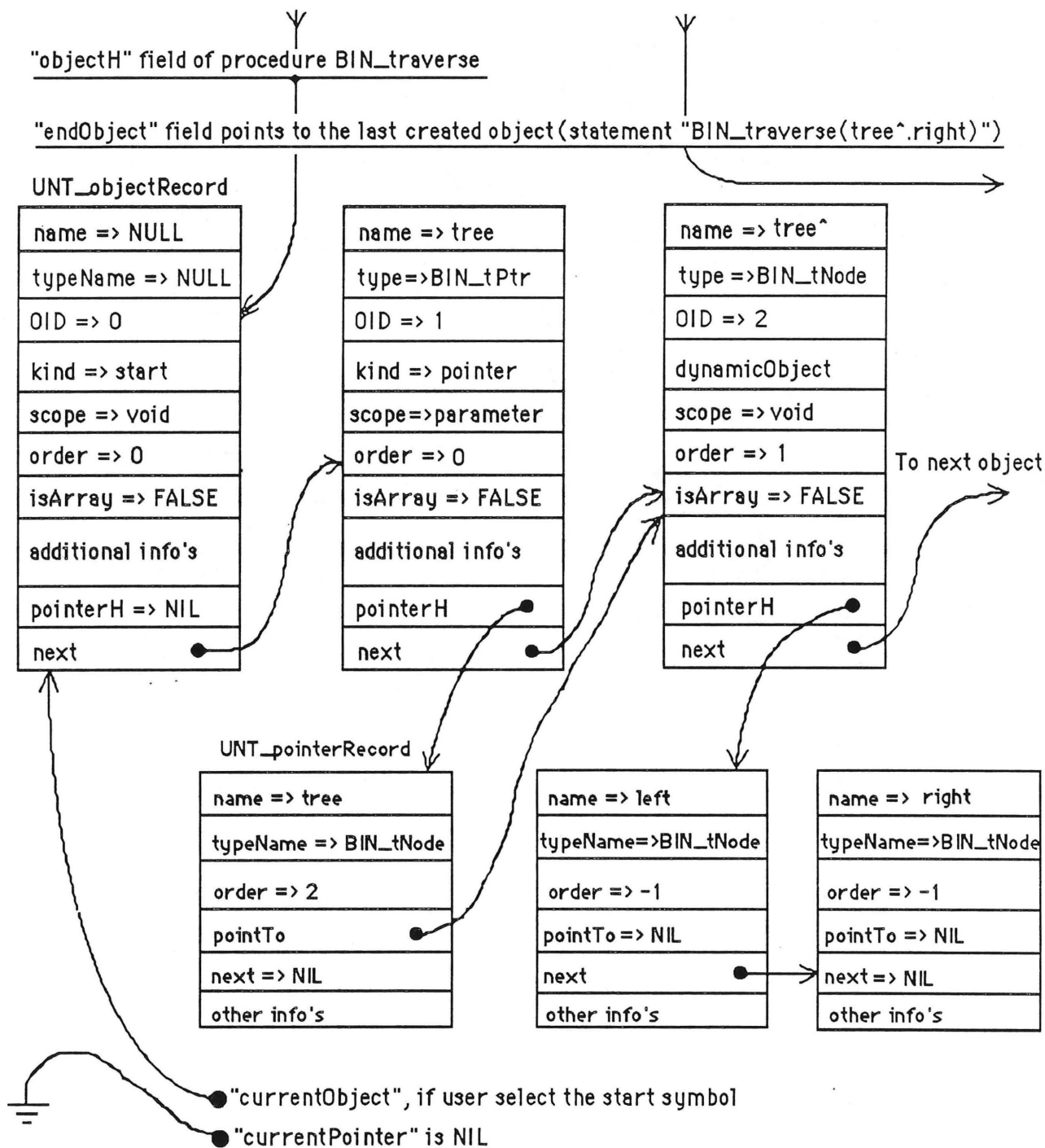


Fig. 5.2 UNT\_objectRecord list (continued from Fig.5.1)

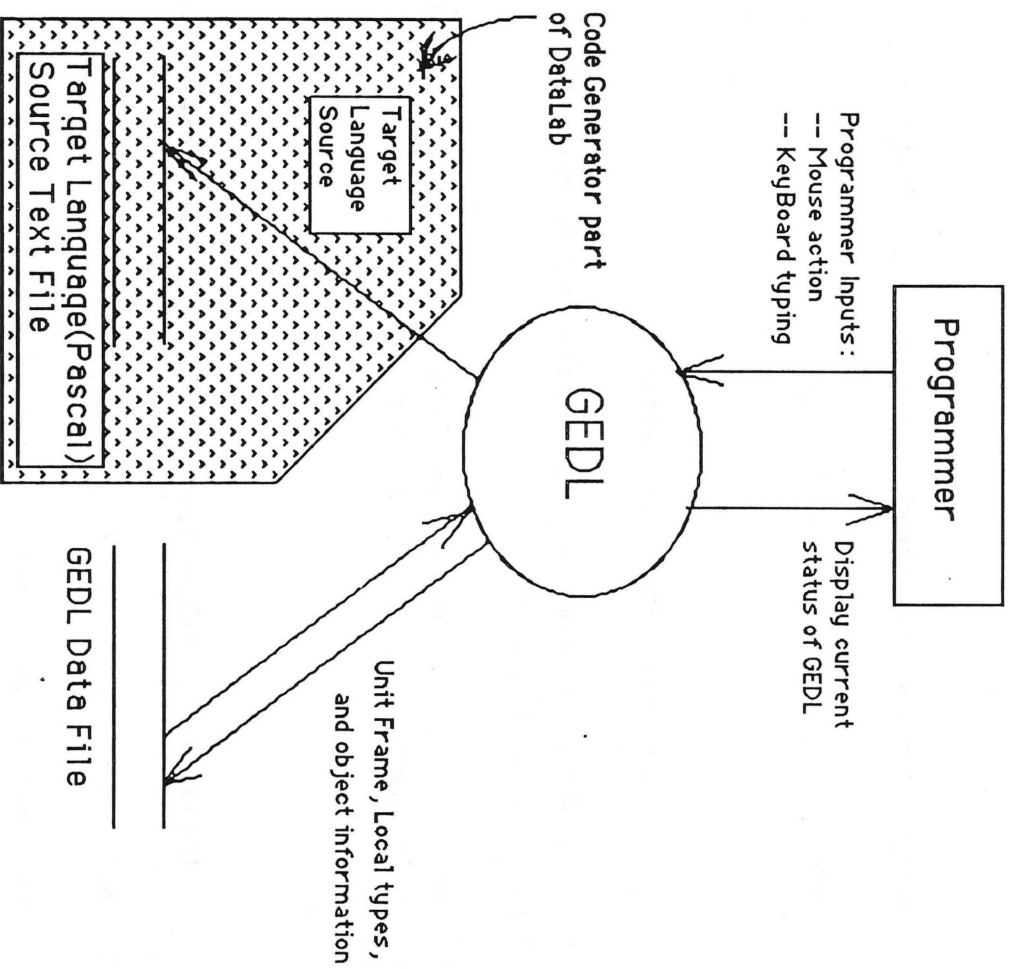


Fig. 5.3 The top-level DFD of GEDL

### 5.3 Possible Extensions

- 1) Currently the code generation part of DataLab produces Pascal source code, but it can be extended to generate other target languages, such as C or C++,
- 2) Array bound and enumerated type range check facility,
- 3) Support Macintosh tool box types.

### 5.4 Requirements and Limitations

1) Users should be familiar with modular design concepts and LightSpeed Pascal experience is required.

2) Generally, graphical language lacks the flexibility to provide the consistency in changing specification, and GEDL is not an exception as we discussed before.

### 5.5 Application Statistics (without code generation part)

- 1) Application size : 53 K
- 2) The lines of source code : about 9 K
- 3) LightSpeed project size without I/O library : 230 K
- 4) Resource size :
  - Uncompiled (DSD.R) : 14 K
  - Compiled (DSD.RSRC) : 6 K
- 5) The number of units in GEDL project : 15

## 6.0 Summary

DataLab is an experimental tool for exploring the advantages of a graphical language system. We tried to use graphics in most appropriate places, as a result, I think we succeeded in this respect. I am sure that GEDL will be a good example for future graphical language systems.

However, as I mentioned before, there are many problems to be solved before practical use. Text-based languages far exceed graphical-based languages in speed, space usage, and flexibility. I believe graphics is appropriate and has advantages over text-based descriptions in some instances, and improvements in those instances will result in wide acceptance of the graphical paradigm.



## 7.0 Bibliography

- 1) T.Lewis, CASE: Computer Aided Software Engineering, An Information Press Software Engineering Series Book, 1988
- 2) M.Al-Mulhem and T.Lewis, DataLab: A graphical System for Specifying and Synthesizing Abstract Data Type, Computer Science Department, Oregon State University, 1989
- 3) S.Yang, T.Lewis and H.Chia-Chi, OSU: Integrating CASE and UIMS, Computer Science Department, Oregon State University, 1988
- 4) A.Aho, R.Sethi, J.Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1987
- 5) T.Budd, A Little SmallTalk, Addison-Wesley Publishing Company, 1987
- 6) D.Sandberg, What is Object-Oriented Programming?, Computer Science Department, Oregon State University, 1987
- 7) Apple Computer, Inside Macintosh Vol.I-V, Addison-Wesley Publishing Company, 1988
- 8) Symantec Corporation, Lightspeed Pascal User's Manual, Symantec Corporation, 1988
- 9) Mainstay, V.I.P 2.51 (Visual Interactive Programming), 1989

## 8.0 Appendices

### 8.1 Unit Frame Syntax

The unit frame syntax is very similar to the "unit" syntax of Lightspeed Pascal, except the implementation part. DataLab uses the graphical representation of operation's algorithm so that GEDL's unit frame does not have the textual description of operation's algorithm.

```
unit <unit name>;

interface -- public part of the unit
  uses <file name list>;
  const <constants, if any -- optional>;
  type <types, if any -- optional>;
  procedure or function lists with parameter(s);

implementation -- private part of the unit
  const <constants, if any -- optional>;
  type <types, if any -- optional>;
  procedure or function lists with parameter(s);

end.
```

More information on the "unit" syntax can be found in Lightspeed Pascal User's Manual.

## 8.2 UNT\_TYPE\_DSD Listing

{written by hae-sung Kim}

{Created on : 6-18-89}

{Last update on :8-16-89}

{-----}

{This unit contains ONLY unit type templates}

{-----}

unit UNT\_TYPE\_DSD;

### interface

#### uses

DIALOG\_DSD;

### type

{Different object kinds -- ALSO will be used in WINDOW unit's TOOL kind}

{DRAG and POINTERDRAG are NOT object kinds, they will be used for drawing ToolsWindow }

{To summarize, REAL object kinds: from 'transform' to 'dynamicObject'}

{Actually kind of objects are STATICOBJECT, UNDEFINED, POINTER and DYNAMICOBJECT}

{Dynamic object means created by New function, NOT by variable declaration}

{Tools kind used in window unit: from 'drag' to 'falseValue'}

{-----}

UNT\_objectKind = (drag, pointerDrag, transform, loop, return, statement,  
start, staticObject, undefined, nilValue, trueValue, falseValue, pointer, dynamicObject);

{Unit part-- decide wether unit record refers unit frame or operation}

UNT\_part = (unitFrame, procOperation, funcOperation);

{Definitions of handles}

UNT\_pointerH = ^UNT\_pointerP;

UNT\_pointerP = ^UNT\_pointerRecord;

UNT\_objectH = ^UNT\_objectP;

UNT\_objectP = ^UNT\_objectRecord;

UNT\_unitH = ^UNT\_unitP;

UNT\_unitP = ^UNT\_unitRecord;

```

{Object record's pointer field }
UNT_pointerRecord = record
    {general pointer information}
    name: Str255;           {pointer name}
    typeName: Str255;      {pointer's dereference type name -- used for checking}
    pointedOID: longint;   {must keep this for saving to/reading from the file}
    order: Str255;        {object place order for each case}

    {drawing information}
    drawingRect: Rect;
    positionRect: Rect;    {for use of global coordinate}
    startPoint, endPoint: Point; {for use of drawing pointer arrow line}
    highlight: boolean;    {is this object selected?}
    copy: boolean;        {is this object is a copy?}

    {handles to object and next pointer}
    pointTo: UNT_objectH;  {which object is pointed -- initially set to NIL}
    next: UNT_pointerH;    {points next pointerRecord}
end; {UNT_pointerRecord}

{Object record definition}
UNT_objectRecord = record
    {general object information}
    name: Str255;           {object name}
    typeName: Str255;      {object's type name}
    OID: longint;          {Object ID number -- used when reading from the file}
    kind: UNT_objectKind;  {the kind of graphical object}
    scope: DIA_objectScope; {the scope of the normal object}
    order: Str255;        {object place order for each case}
    isArray: boolean;     {If array object, set to TRUE -- used by code generation part}

    {drawing information}
    drawingRect: Rect;
    positionRect: Rect;    {for use of global coordinate}
    highlight: boolean;    {is this object selected?}
    copy: boolean;        {is this object is a copy?}

```

```
    {handles from object record}
    pointerH: UNT_pointerH;    {pointer list if object has any}
    next: UNT_objectH;        {points next objectRecord}
end;    {UNT_objectRecord}
```

{Unit record definition}

**UNT\_unitRecord = record**

```
    name: Str255;                {whether unit name or unit operation's name}
    part: UNT_part;              {whether unit frame or operation}
    objectH: UNT_objectH;        {an operation's drawing part info}
    endObject: UNT_objectH;      {maintains a handle that points the last object}
    objectOrder: longint;        {maintains a last order of the operation}
    textH: Handle;               {an operation's local type info or unit frame info}
    next: UNT_unith;
end;    {UNT_unitRecord}
```

**implementation**

{Nothing here}

**end.** {UNIT\_TYPE\_DSD}

## 8.3 GEDL Data File Format

```

<unit frame text -- the same as the user typed in unit frame window>
<blank line>
%
<blank line>
<operation name -- such as "procedure BIN_traverse">
<local type/constant text, if any -- optional>
BEGIN
% -- object record information start symbol
<object name>
<object type name, object ID, object kind>
<object scope, order, isArray, copy>
<object drawingRect information>
%% -- pointer record information, if the object has pointer(s)
<pointer name, pointer type, pointed object OID>
<order, copy>
<pointer drawingRect information>
<pointer line startPoint, endPoint>
%% -- next pointer record, if any
---
---
% -- next object information start from here
----
----
%%% -- save next sequence order for each operation
<next sequence order>
END; -- an operation's specification end here
<blank line>
<other operations, the same as above format
-----
-----
----->
<next object ID number>

```

#### 8.4 Data File Example (BIN\_traverse)

```

unit BIN_TREE;
interface
uses LINKED_LIST;

type
  BIN_tPtr = ^BIN_tNode;
  BIN_tNode = record
    left:BIN_tPtr;
    right:BIN_tPtr;
    data:integer;
  end;

procedure BIN_traverse( tree:BIN_tPtr);
procedure BIN_insert( var tree:BIN_tPtr;
                     newData:integer);

implementation
end.
%

procedure BIN_traverse
BEGIN
%

  0 6
  3 0 0 0
  66 60 86 80
%
tree
BIN_tPtr 1 12
2 0 0 0
66 126 76 136
%%
tree BIN_tNode 2
2 0
59 153 71 173
71 131 71 216

```

```
%
tree^
BIN_tNode 2 13
3 1 0 0
61 216 81 256
%%
left BIN_tNode -1
%%
right BIN_tNode -1
%

3 2
3 3 0 0
57 292 77 332
%
writeln(tree^.data);
4 5
3 4 0 0
103 277 123 357
%
BIN_traverse(tree^.left);
5 5
3 5 0 0
159 257 179 337
%
BIN_traverse(tree^.right);
6 5
3 6 0 0
203 252 223 332
%%
7
END;

procedure BIN_insert
BEGIN
%%
0
END;

7
```