APPENDIX A

An Editing Environment for RESOLVE

The objectives in this project were to develop a RESOLVE structure editor for a small workstation that:

- permits only the creation of syntactically and static-semantically correct programs
- makes minimal use of the keyboard
- minimizes mouse movement

This appendix describes the prototype of this editor, implemented on the Macintosh. Two interesting characteristics are 1) the only use of the keyboard is to name an identifier, and 2) there is extensive use of hierarchical "popup" menus that minimize mouse movement.

This appendix is neither a user manual nor a tutorial, but rather a demonstration of the "look and feel" of the editor using a handful of sample editing sessions. A general knowledge of the Macintosh user interface is assumed (e.g., windowing, clicking, etc.).

Figure 27 is a snapshot of the Macintosh screen taken during an editing session for some realization module. The programmer has already instantiated two facilities — Int_Fac and Int_Stk_Fac[37] — and has declared a local operation called Copy_Stack that makes a copy of a variable of type Int_Stk_Fac.Stack. The programmer is in the process of creating the code for this procedure, using the algorithm presented in Figure 24.

---

[37] The conceptualizations for these are given in Figures 21 and 2, respectively.

```
 File  Edit
                         Demo1.real
   realization auxiliary
      facilities
         facility Int_Fac is Bounded_Integer_Template
               realized by <REALIZATION NAME>
         end Int_Fac

         facility Int_Stk_Fac is Stack_Template (Int_Fac.Int)
               realized by <REALIZATION NAME>
         end Int_Stk_Fac
      end facilities

      operations
         procedure Copy_Stack
            parameters
               preserves s1 : Int_Stk_Fac.Stack
               produces s2 : Int_Stk_Fac.Stack
            end parameters
            ensures "s2 = #s1"
            variables
               garbage : Int_Stk_Fac.Stack
            end variables
            begin
               <VAR NAME> :=: <VAR NAME>
         end Copy_Stack
      end operations
```

Figure 27

RESOLVE Editor Screen Snapshot

Items within a RESOLVE editor window are classified as *keywords*, *special symbols*, *placeholders*, *identifiers*, or *assertions*. Keywords appear bolded (e.g., **`variables`**), special symbols are sequences of one or more punctuation characters (e.g., :=:), placeholders begin and end with “<” and “>” (e.g., `<VAR NAME>`), identifiers appear as plain text (e.g., garbage), and assertions are surrounded by double quote marks (e.g., `"s2 = #s1"`).

Editing is accomplished by pointing to an item in the edit window (using the mouse) and pressing the mouse button. If something interesting can be done with that item, a popup menu appears on the screen, providing the programmer with a set of allowable actions. The following subsections explore some interesting and useful actions.

Before we begin, it is interesting to note that there are only two menus in the main menu bar shown in Figure 27 — File and Edit. The contents of these menus is shown in Figure 28. The reason there are so few menus is the reliance on popup menus for editing, as demonstrated in the following subsections.

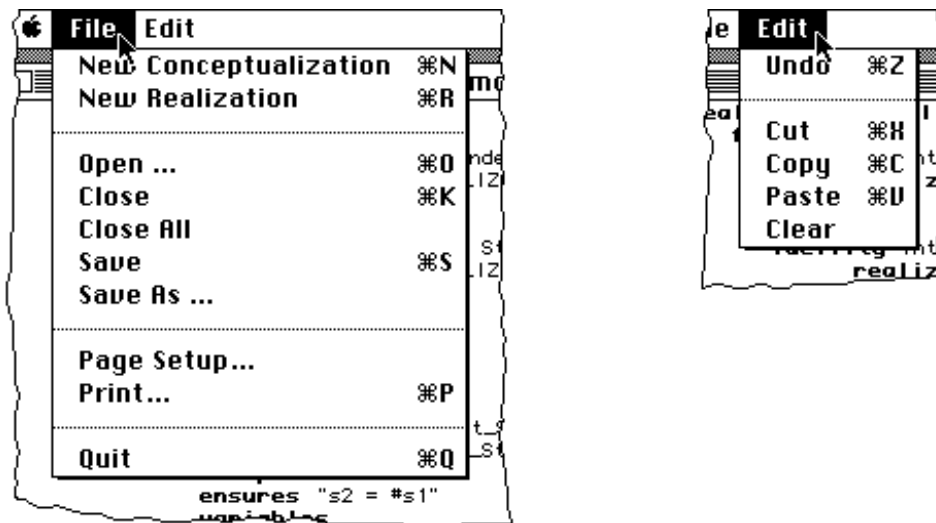

Figure 28

RESOLVE Editor Menus

A.1 <VAR NAME> Placeholders and Variables

Let's start with placeholders, which are items that must be "filled in" by the programmer before the program is considered complete. When the mouse is pressed in a placeholder, a popup menu appears that contains all legitimate replacements for the selected placeholder.

One such placeholder in Figure 27 is `<VAR NAME>`, which is a placeholder for a variable name. The popup menu for a `<VAR NAME>` placeholder contains all variables that can used there. If a type has already been bound to the placeholder (e.g., a swap statement with at least one variable selected), the popup menu contains only the variables of the proper type; if a type has not been bound (e.g., a swap statement with neither variable selected), the popup menu contains all variables, organized hierarchically by type.

For example, Figure 29 shows the programmer replacing the left `<VAR NAME>` placeholder of the swap statement from Figure 27 with variable garbage (of type Int_Stk_Fac.Stack), and Figure 30 shows the programmer replacing the right placeholder with variable s2. Note that in this second situation the popup menu only

contains variables of type Int_Stk_Fac.Stack because this type was bound to the placeholder when variable garbage was selected for the left <VAR NAME>. It does not matter which <VAR NAME> of a swap statement is replaced first.

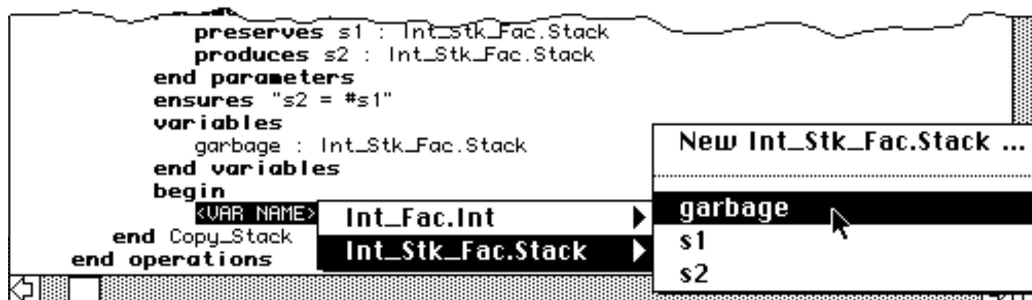

Figure 29

Replacing An Untyped <VAR NAME> Placeholder

Figure 30

Replacing a Typed <VAR NAME> Placeholder

The first (and possibly only) menu item in every variable popup menu (e.g., New Int_Stk_Fac.Stack ...) is used to declare a new variable, allowing the programmer to conveniently declare a variable when it is first used. This saves the programmer from continually moving between the variable declaration section and code. Of course, a programmer can declare a variable in the variable declaration section if he or she chooses. Details of both techniques are discussed in Section A.3.

If the mouse is pressed on a variable name, a variable popup menu appears, allowing the programmer to replace the current variable with another of the same type. This is shown in Figure 31. (Note that the programmer decided not to change s2.)

Figure 31

Changing a Variable

A.2 Inserting Statements and Control Invocations

Inserting a new statement (e.g., a while statement or variable declaration) or section (e.g., realization auxiliary section of a realization) is accomplished by placing the mouse in the white space between the program body and the left edge of the window and pressing the mouse button. When the mouse is in this white space (without the button pressed), an "insertion arrow" appears to its right if (and only if) it is possible to insert a statement or section there. An insertion arrow is displayed immediately to the left of the program body and is never *on* a statement, but always *between* two statements.

Figure 32 shows the insertion arrows displayed as the mouse is moved downward in the white space to the left of the parameters section of procedure Copy_Stack. The insertion arrow changes its shape (i.e., ↗, →, or ↘) based on the indentation of the program, making it more noticeable. Moving the mouse horizontally in the white space does not change the shape or position of the insertion arrow.

Pressing the mouse button when an insertion arrow is displayed causes a popup menu to appear, containing the insertions possible at this location. For example, Figure 33 shows the insertion arrow located within the code section of Copy_Stacks, and the popup menu displayed when the mouse button is pressed. In this example, the

programmer has chosen to insert a “while not” statement. Note that the popup menu appears next to the mouse location, not the insertion arrow.

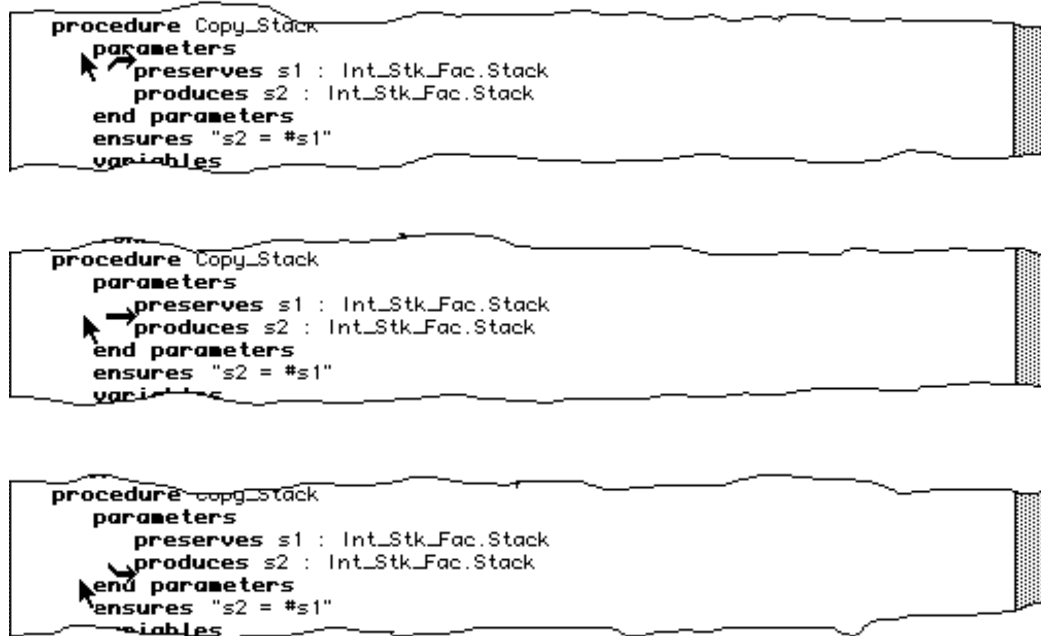

Figure 32

Insertion Arrow Positions

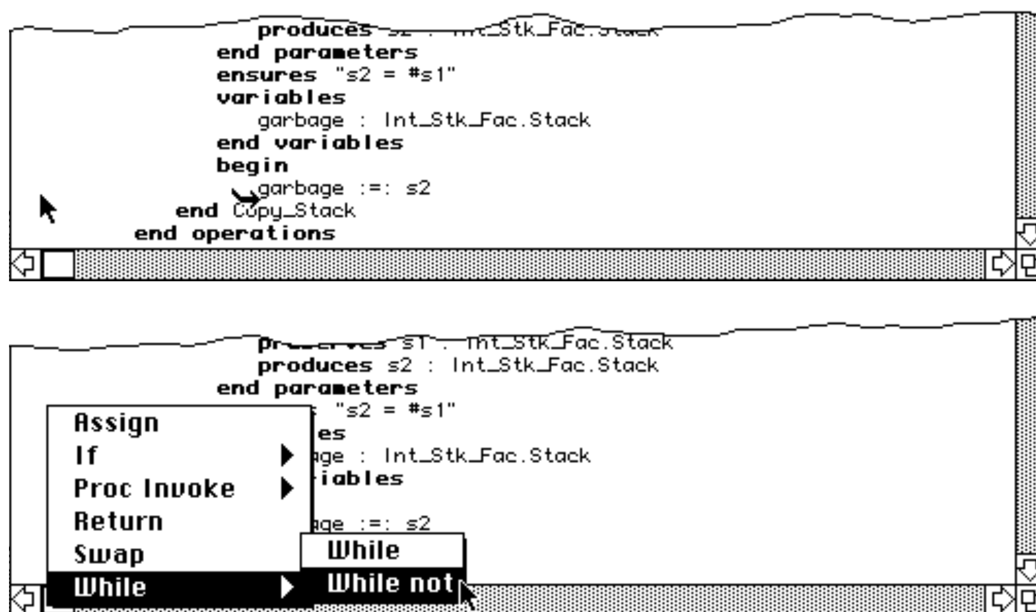

Figure 33

Inserting a While Statement Into Copy_Stack

Figure 34 shows the insertion arrow positioned within a code section, and the popup menu by which the programmer is inserting an invocation of procedure Pop provided by facility Int_Stk_Fac. Note that the procedure invocation menu is organized hierarchically by facilities providing procedures.

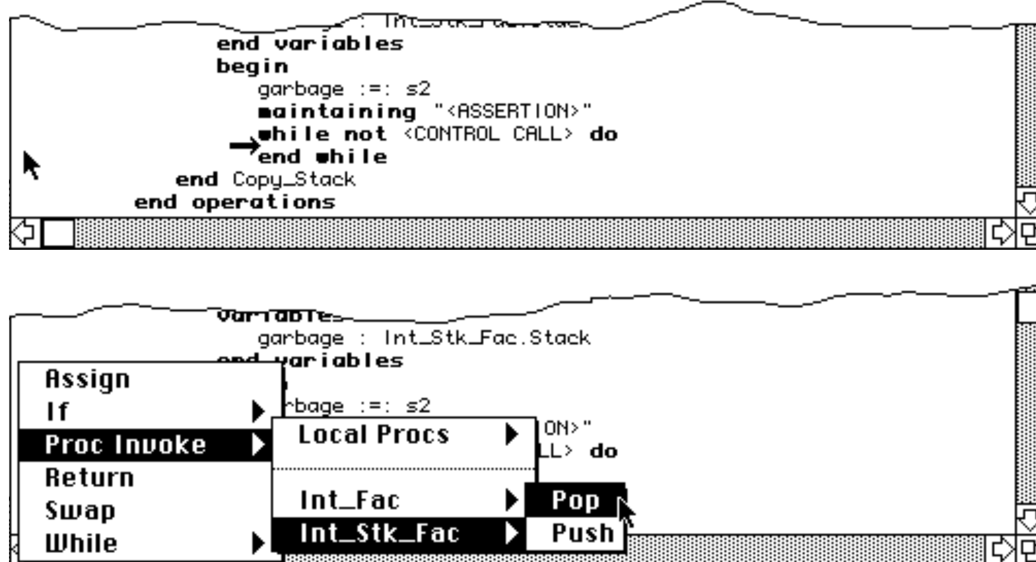

Figure 34

Inserting an Invocation of Procedure Pop

Figure 35 shows the popup menu that is displayed when the mouse is pressed in a `<CONTROL CALL>` placeholder. Here the programmer is inserting an invocation of control Is_Empty provided by facility Int_Stk_Fac. Note that the menu is organized hierarchically, similar to the procedure invocation menu in Figure 34. (Also note that the programmer previously inserted an invocation of procedure Push in the while loop.)

Figure 35

Inserting an Invocation of Control Is_Empty

A.3 Variable Declaration

The control and procedure invocations inserted into the example program in the previous section have a total of five actual parameters. Recall from Section 3.3.1.3 that all actual

parameters are variables, so every actual parameter appears as a `<VAR NAME>` placeholder in the code. Note that a type is bound to each actual parameter[38]. Replacing these `<VAR NAME>` placeholders with variable names is done as discussed in Section A.1.

These five `<VAR NAME>` placeholders need to be replaced by three variables — s1, temp, and catalyst (see Figure 24 in Section 3.7.1). However, two of these variables — temp and catalyst — need to be declared first.

One way to declare a variable is to insert a variable declaration statement in the variable declaration section. Figure 36 shows the popup menu containing the available types that appears for a variable declaration. (The insertion arrow was between the declaration of variable garbage and the **`end variables`** statement when the mouse button was pressed.) The name of the variable is entered in a dialog box that appears, as shown in Figure 37. Note that the dialog box contains the names of all declared variables; this information may be useful to the programmer in determining a unique name for the new variable.

As shown in these figures, the programmer is declaring a variable named catalyst of type Int_Stk_Fac.Stack. (Also shown is the fact that our programmer can't spell very well! Hopefully this mistake won't go unnoticed for long.)

---

[38] It is possible that the type of an actual parameter may not be known. This occurs when a type parameter to a facility declaration has not been filled in. For example, the type of the second parameter to procedure Push is whatever type is passed to the instantiation of Stack_Template. If an invocation of Push is inserted into the code before the facility's parameter is filled in, the `<VAR NAME>` placeholder for the second parameter to Push cannot be replaced.

Figure 36

Type Declaration Menu

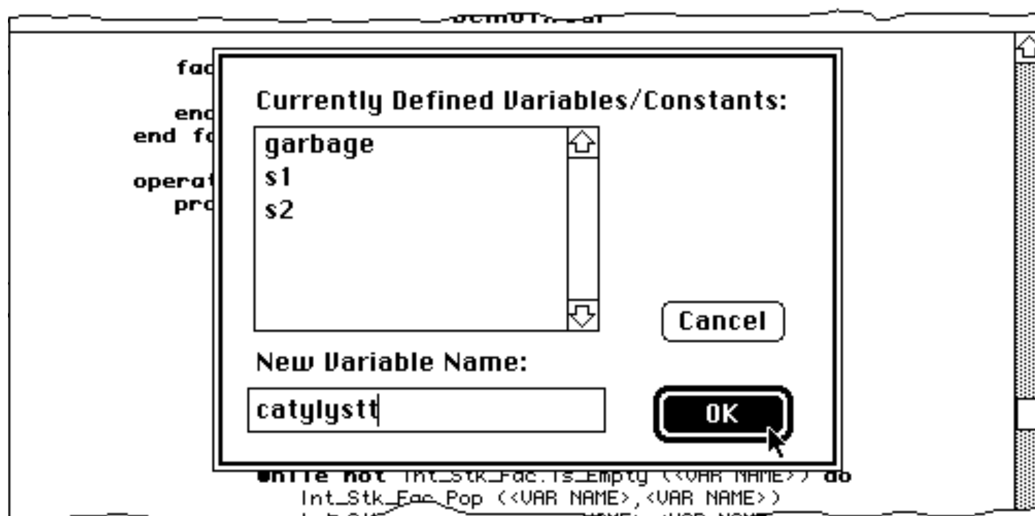

Figure 37

Dialog Box to Name a Variable

The second way to define a variable is to select the "New ..." menu item in the `<VAR NAME>` placeholder menu. Figures 38 and 39 show the menu and dialog box used by the programmer to declare variable temp of type Int_Fac.Int. Note in Figure 38 that the type of variable is known (i.e., Int_Fac.Int), and that no variables of this type are currently defined. The radio buttons in the dialog box in Figure 39 are used to indicate the scope

of the new variable (i.e., local to the operation or global to the module) and the position within the variable declaration list (i.e., either first in the list, or last)[39].

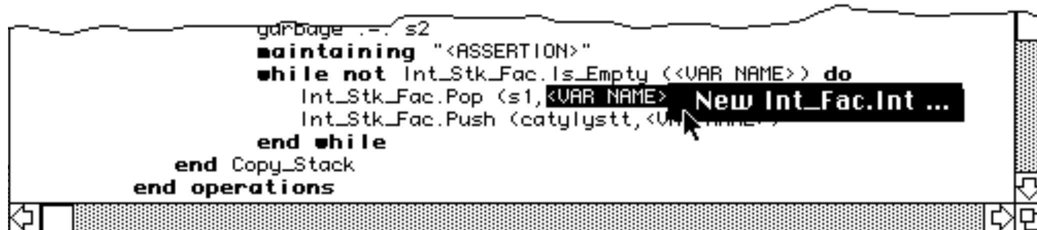

Figure 38

Menu for In-Place Variable Declaration

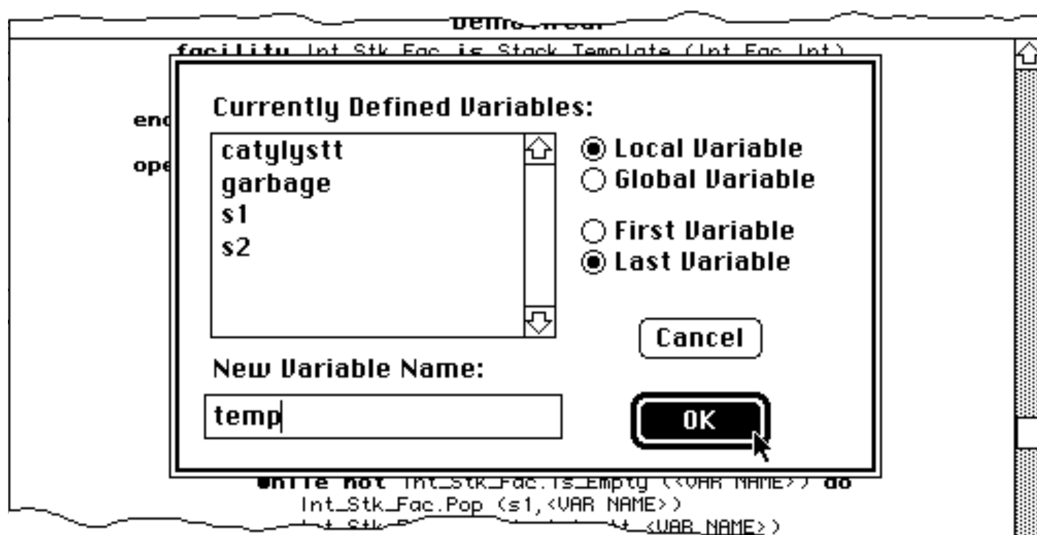

Figure 39

Dialog Box for In-Place Variable Declaration

The name of a variable can be changed by pressing the mouse button on the variable name in its declaration. When this is done a popup menu appears containing only the item "New Variable Name ...". If this item is selected, a dialog box very similar to the one in Figure 37 appears, allowing the programmer to enter a new name for the variable.

---

39 The position of the variable declaration in the list has no effect on the declaration but is purely cosmetic.

The editor then changes that variable’s name everywhere it is used in the program. Thus, our klutzy programmer can easily change variable catylystt to catalyst.

A.4 Inserting Function Invocations

The righthand side of a function assignment statement is a function invocation, which appears as a `<FUNCTION CALL>` placeholder when a function assignment statement is inserted into a program. Pressing the mouse button on this placeholder causes a popup menu to appear containing all functions that can be invoked. Figure 40 shows the programmer selecting function Add provided by Int_Fac. Note that because the assignment variable k is of type Int_Fac.Int, the popup menu contains only functions of this type. The menu is hierarchically organized by facilities providing functions of the appropriate type (in this case Int_Fac is the only one), similar to the organization of the control invocation menu in Figure 35. If a type is not bound to the `<FUNCTION CALL>` placeholder the first hierarchy level contains all types for which functions exist.

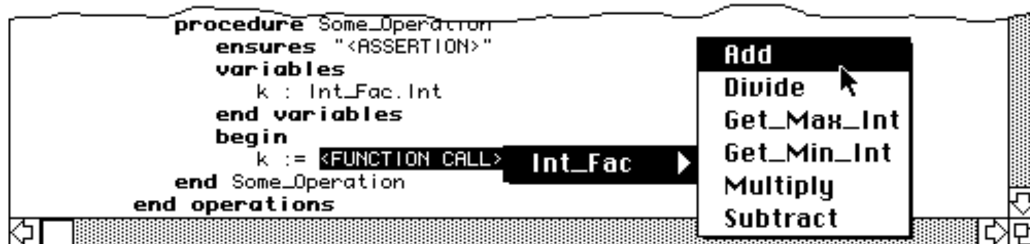

Figure 40

Replacing a Typed <FUNCTION CALL> Placeholder

If the mouse is pressed in a function identifier, a popup menu just like the one in the above figure is displayed, allowing the programmer to replace the current function invocation with another one of the same type.

A.5 If and Return Statements

Figure 41 shows the popup menu displayed for inserting an if...then...else statement into a program. Note that the four if statement forms are placed in a hierarchical menu, reducing the size of the first statement menu.

Figure 41

Inserting an If...Then...Else Statement

Recall from Section 3.3.3 that the simple return statement is allowed in functions and procedures, whereas a control must use either a return yes or return no statement. Figure 42 shows the programmer inserting a return yes statement in a control operation. Note that Return is a hierarchical menu, which is not the case in the menus shown in Figures 33, 34, and 41, because these figures showed the programmer inserting statements into procedures or functions, not controls.

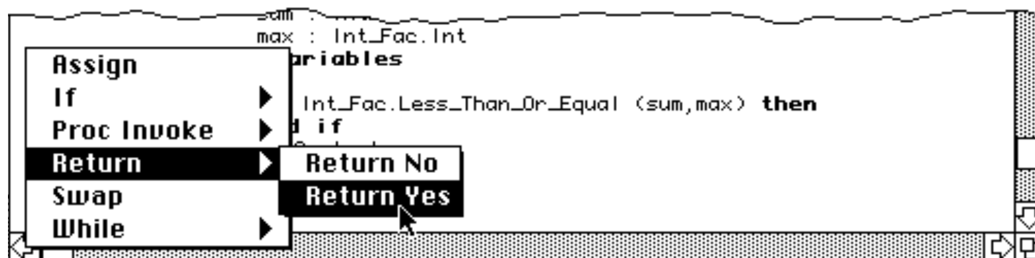

Figure 42

Inserting a Return Yes Statement

A.6 Editing Assertions

An assertion in a RESOLVE program currently consists of text surrounded by quote marks, and is meant ultimately for processing by a verifier. Assertions are treated as “free text” by the editor, and editing them is done using the standard Macintosh text editing facilities.

When the mouse is pressed within an assertion, a rectangular box is drawn around the text, as shown in Figure 43; all typing on the keyboard and mouse actions within this rectangle (i.e., clicking and dragging) are processed following the standard Macintosh user interface. Note that inserting and deleting return characters causes the rectangle to enlarge and shrink accordingly. When the mouse is pressed outside of the editing rectangle, the rectangle disappears, and RESOLVE editing resumes as normal.

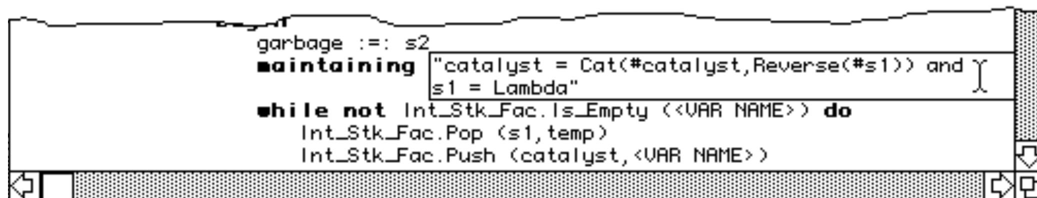

Figure 43

Editing an Assertion

A.7 Creating Conceptualizations, Types, and Operations

A conceptualization is created by selecting the appropriate menu item from the File menu (see Figure 28). A window is created containing the structure of the new conceptualization, as shown in Figure 44. The conceptualization is named by pressing the mouse button on the <CONCEPTUAL NAME> placeholder, selecting the "New Conceptual Name..." item from the popup menu that appears (it's the only item in the menu), and typing the name in the dialog box that is displayed. Conceptualization sections (e.g., parameters, auxiliary, interface, description) and items within them (e.g., facility parameters and operations) are inserted the same way that statements were inserted in Section A.2.

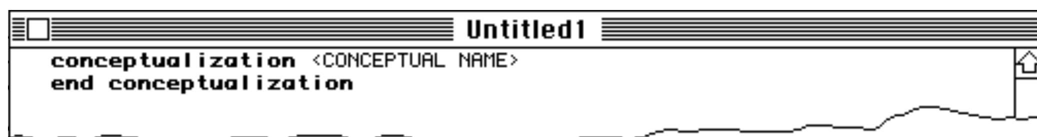

Figure 44

Newly Created Conceptualization

In a facility declaration, the conceptualization is specified by pressing the mouse button on the <CONCEPTUAL NAME> placeholder in the facility declaration, and selecting the "Select Conceptualization..." menu item that appears, as shown in Figure 45. A dialog box containing all available conceptualizations is displayed, and the programmer selects the desired one. (Note that the programmer is creating a conceptualization for bounded stacks, shown in Figure 15.)

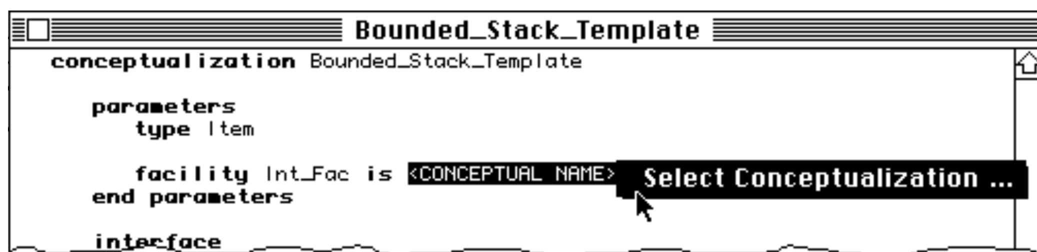

Figure 45

Replacing a <CONCEPTUAL NAME> Placeholder in a Facility Declaration

Type and operation declarations are inserted in the interface section using the same technique. For example, Figure 46 shows the programmer inserting a function of type Int_Fac.Int. A dialog box appears, in which the programmer types in the name of the function.

Figure 46

Inserting a Function Declaration

Figure 47 shows the programmer inserting a Bounded_Stack formal parameter declaration into the parameters section of function Get_Max_Size. The name and mode of the formal parameter are specified in a dialog box that appears, as shown in Figure 48. Note that the parameter in these figures is to a function, which must be a preserves parameter (see Section 3.1.6); this is why preserves is the only selectable parameter mode.

Figure 47

Inserting a Formal Parameter Declaration

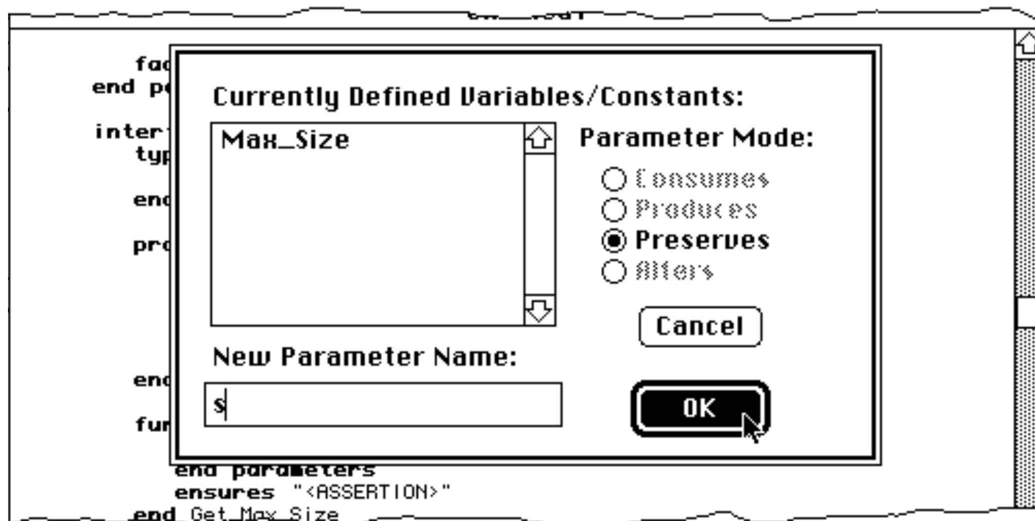

Figure 48

Dialog Box for Naming a Formal Parameter

A.8 Creating Realizations

A realization is created by selecting the appropriate menu item from the File menu (see Figure 28). A window is created containing the structure of the new realization, as shown in Figure 49. The realization is named by pressing the mouse button on the <REALIZATION NAME> placeholder, selecting the "New Realization Name..." item from the popup menu that appears (it's the only item in the menu), and typing the name in the dialog box that is displayed.

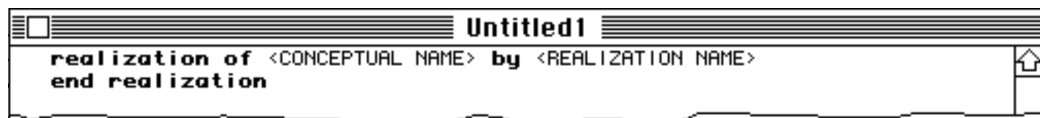

Figure 49

Newly Created Realization

The conceptualization that the realization implements is specified in the same manner as for a facility declaration discussed in the previous section — the mouse button is pressed in the <CONCEPTUAL NAME> placeholder, the item from the displayed popup menu is selected, and the conceptualization is selected from a list of available conceptualizations. When the conceptualization is selected, the editor uses the interface section of the conceptualization to automatically create the interface section in the realization, as shown in Figure 50. The items inserted into the interface section from the conceptualization (e.g., operation names and formal parameters) are not editable in the realization.