

7. Lists

Overview

One last special built-in data type remains to be discussed. This is the *list*. Although it is similar in many ways to the *array* type found in other languages, it is simpler to use and much more powerful. Operations may be performed on entire lists of data automatically with Prograph. This chapter will show you how to take full advantage of lists.

Lists -- Collections of Things

Many computer languages support the creation of *groups* of variables. In BASIC, Pascal and C++, you can make *arrays* of variables, in which each individual variable, or *array member*, is given a unique *index* in the array to access it. For example, if your array has 200 variables in it, the array indices range from 1 to 200 (or in the C++ language, from 0 to 199). The 30th variable in the array would have an index of 30 in Pascal (or 29 in C++). Each variable in the array is stored sequentially in memory.

Prograph provides a similar construct called a *list*. Lists can be used to hold groups of integers, real numbers, strings or objects. There is no built-in list data type in C++ -- the closest correlate is an array. Arrays only allow you to set or get individual elements -- *all other actions on arrays must be written by the programmer*. In addition, before you can use the array, you must declare the single type of data it will contain and how many elements it will contain. This is not very flexible.

Unlike C++ arrays, Prograph lists are designed to be flexible. Their storage type is flexible. Their size is flexible. Even their contents are flexible -- the elements of a Prograph list *do not* need to be all of a single data type. One element can be an integer, another a float, another a string, and so on. You can also perform actions on the entire list at once without having to write code yourself to do so. If you send a list to an operation, each element in the list is acted upon in turn automatically for you! In fact, many of the Prograph primitives were constructed with this in mind -- they can accept either single variables or lists as inputs.

List Multiplexes

It's actually quite easy to adapt a program for using lists. Let's show you an example.

Create a new program named Sales Tax. Complete the Sales Tax window as follows. This method will ask the user to input a sales tax rate, then the prices of several items whose total prices (price + tax) will be calculated and displayed. The rightmost ask primitive's prompt should read "List of prices (with spaces between each and enclosed in parentheses):" to reflect that a *list* of prices will be entered. But wait! The method, as

shown in Figure 7.1, looks like it will only ask for *one* price and recalculate its price with sales tax added.

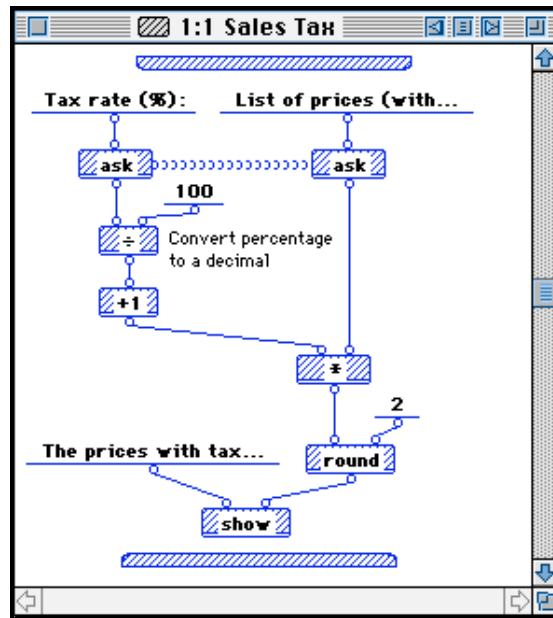
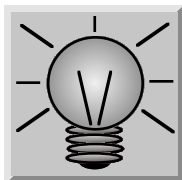


Figure 7.1: The Sales Tax method without list processing

How do we get the program to work on the *several* prices contained in the list? Simple. Highlight the right terminal node and the root node of the * primitive icon. Select the List item from the Controls Menu to convert them to list-handling nodes called *list multiplexes* (see Figure 7.2). That's all there is to it. The primitive will now expect a *list* of before-tax prices, and will multiply *each price* in the list by the *tax rate plus 1* to yield a *list* of after-tax prices. The new appearance of these list-handling nodes visually reinforces the fact that they process many values rather than one.



A Hint...

To change the list-handling nodes back to nodes that expect a single variable as input, highlight the list node and select the Simple item from the Controls Menu.

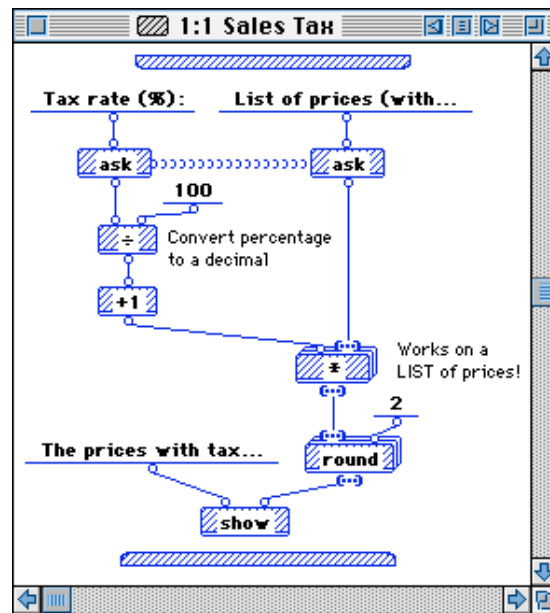
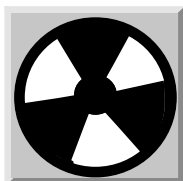


Figure 7.2: The Sales Tax method with list processing

Let's see how the program will run. Select the Execute Method menu command. First, you should be presented with a prompt to enter the sales tax rate in percent (Figure 7.3).

Figure 7.3: Prompt for tax rate

Next, you'll be asked to enter the list of prices to be worked on. *The correct way to enter an entire list at once is to enclose the entire list in parentheses () and separate each data item in the list by spaces on either side of it, as shown in Figure 7.4.*



Warning!

The user of this particular program must input the list in the proper list notation format, or else the program will generate an error indicating that the * primitive was expecting a list but didn't get one.

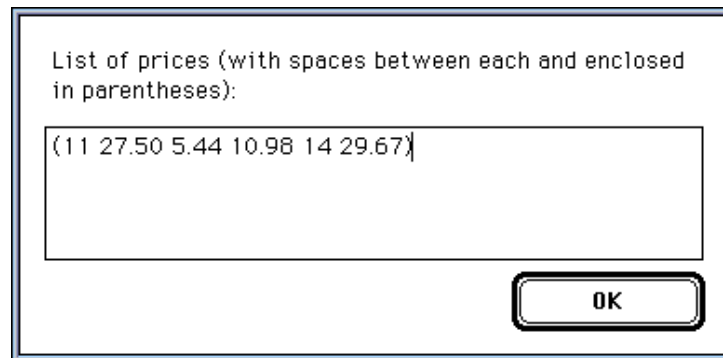


Figure 7.4: Prompt for list of prices

The final output of the program is a list of prices, each price with the sales tax added to it (see Figure 7.5).

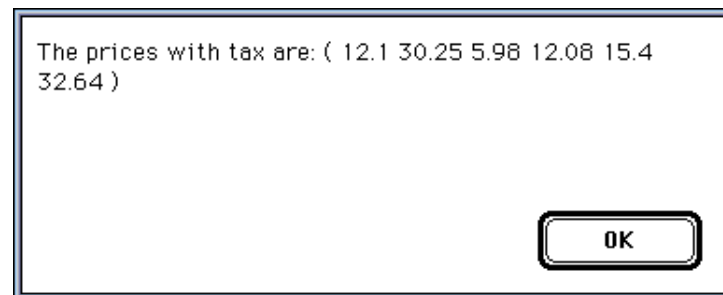


Figure 7.5: Output of the Sales Tax method

Simply by changing the nodes of one primitive to list-type nodes, the program now acts upon *each item in the list one after another automatically*. The process of repeating an operation for every item in a list is called *multiplexing or autoindexing*. In addition, unlike in C++, we don't have to worry about accidentally "going off the end" of the array by incorrectly indexing past the last array element. That is, in C++, if an array has 30 elements, but a loop tries to access element 31 by mistake, the program can crash since just about anything could be stored at that memory location. In contrast, when we send a list to an operation in Prograph, the list automatically "knows" how many elements it has, so this kind of mistake can't happen. Lists are one feature of Prograph that lets Prograph "flex its muscles" over other languages.

Exercise 7.1:

Write a program which will ask the user to input a list of numbers and a divisor, then output the remainder (not the quotient) of each number after division by the divisor. For example, if the divisor is 10, and the input list is (25 37 50 7), the output list should be (5 7 0 7).

Getting and Setting Individual Elements

The Sales Tax program automatically performed a sales tax calculation on each member of the price list. What if we needed to calculate the tax on only *one member* of the list? Can we access individual members of a list? We'll see in this lesson how to do so.

Create a new One Price program, section and method. Start filling in its code diagram a shown in Figure 7.6.

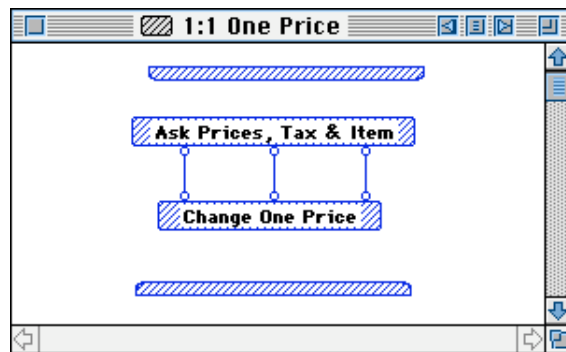


Figure 7.6: The One Price method

The first method used, Ask Prices, Tax & Item, is a modification of the initial code of the Sales Tax program (see Figure 7.7). We've just moved the *ask* commands into a separate method. Open the method's code window and enter the following diagram. Notice that we've added a third *ask* primitive so that the user can indicate *on which* price in the list the program should perform the tax calculation. We've also moved the tax rate factor conversion (from percentage tax rate to a fraction by which to multiply) into this method for clarity.

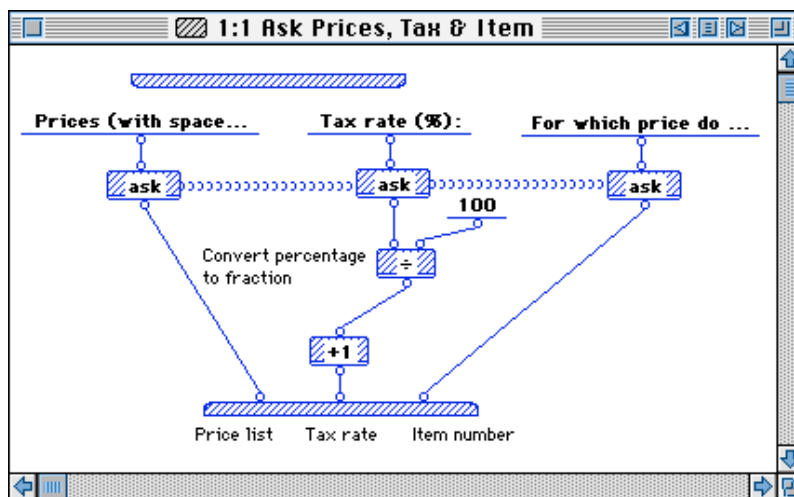


Figure 7.7: The Ask Prices, Tax & Item method

The Change One Price method (Figure 7.8) will carry out the sales tax calculation on only the price you specified in the price *list*. Let's start writing its code. Open the Change One Price code window. We'll use the primitive `get-nth` to get the value of one item in the price list. This primitive expects two inputs: the list to work on, and which item to access. It returns the requested list element, which we'll multiply by the tax rate.

Once the price in question has been adjusted to include sales tax, we'll put the new value of that item back into the list using the `set-nth` primitive. This primitive sets the value of one element of a list. It expects three inputs: the list, the element index, and the new value for that list element. We feed into it the same price list used as input to `get-nth`, the same item index, and the adjusted price of that item produced by the `*` primitive.

We want to return from this method the adjusted (with tax) price of the item. To do this, we simply call `get-nth` again. Since `set-nth` returns the entire *list*, we can feed this *directly* into `get-nth`'s list input node. This avoids cluttering up the code diagram with a long link stretching from the input bar's list input node down to the `get-nth` node. For the present program, we could have just called `get-nth` once, multiplied the list item by the tax rate and output it. We've included the rest of the code in this method to show you how to store the new value of that list element so it could be used again later, should your program need it.

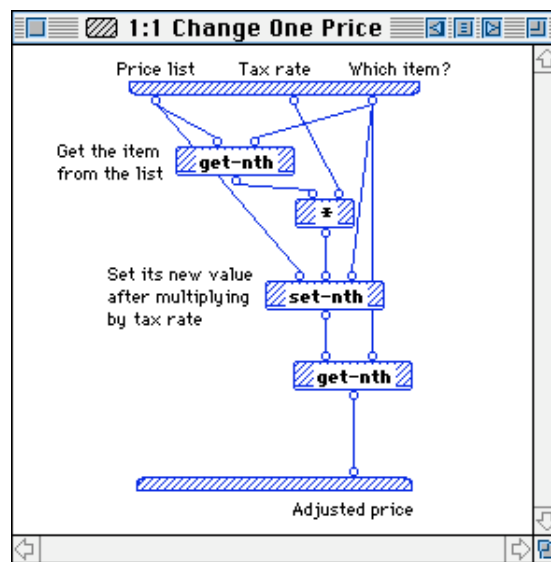


Figure 7.8: The Change One Price method

We now complete the One Price method by printing out the adjusted price for this single item, as shown in Figure 7.9.

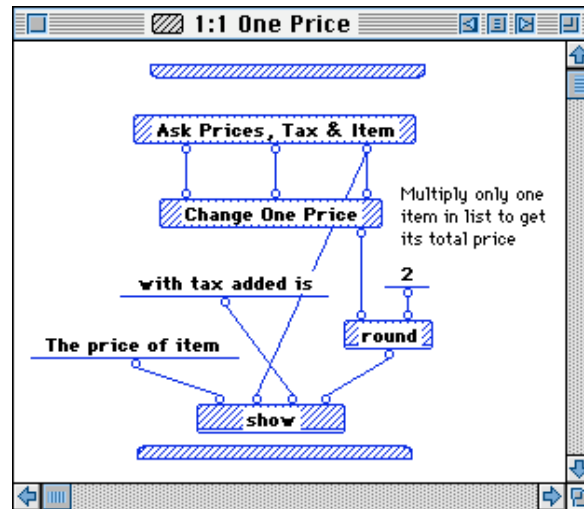


Figure 7.9: The completed One Price method

The output display should look something like what's shown in Figure 7.10:

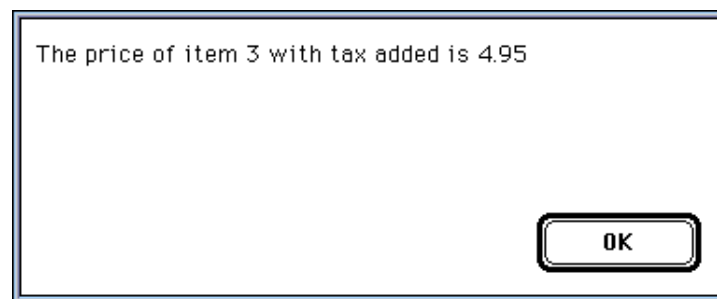


Figure 7.10: Output of the One Price program

Partitioning Lists

Lists do not have to be used all at once as a single unit. We've already seen that single items in a list may be used one at a time. We can also use larger sections of a list at once by splitting off parts of a list into a new list. The list can be split apart according to whatever criterion you wish. For example, you can split apart a list into two new lists so that one list contains only the odd numbers of the original list and the other contains the remaining even numbers. In this lesson, we'll create a list of numbers, then split or *partition* the list into two lists -- one containing numbers over a cut-off value you specify, and the other containing the rest of the numbers that are below the cut-off value.

Create a new Separate A List project, section and method, then place two `ask` commands into it (Figure 7.11). The first `ask` primitive will prompt the user to enter a list of numbers. The second one will ask for a cut-off value with which to partition the list. All numbers in the list with values less than this will go into one new list, and all numbers greater than this will go into a second new list.

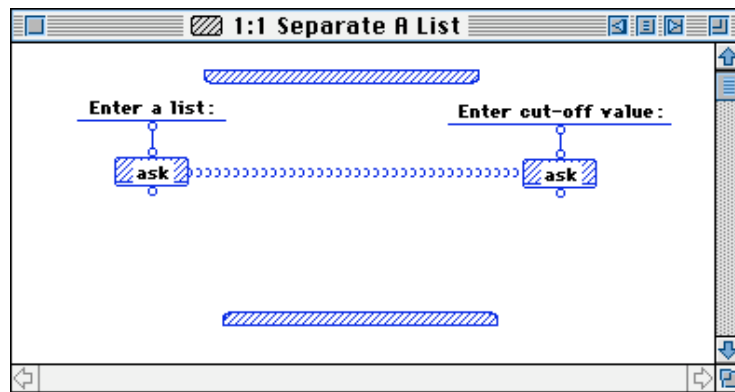


Figure 7.11: Initial code for the Separate A List method

What if the cut-off number entered by the user was greater than or less than any of the numbers in the list? In this case, we'd have no need to partition the list at all into large and small numbers. We therefore have to check for this possibility by testing if the cut-off value is within the range of numbers in the list. We do this in a local method named *Cut-off in range?* (see Figure 7.12).

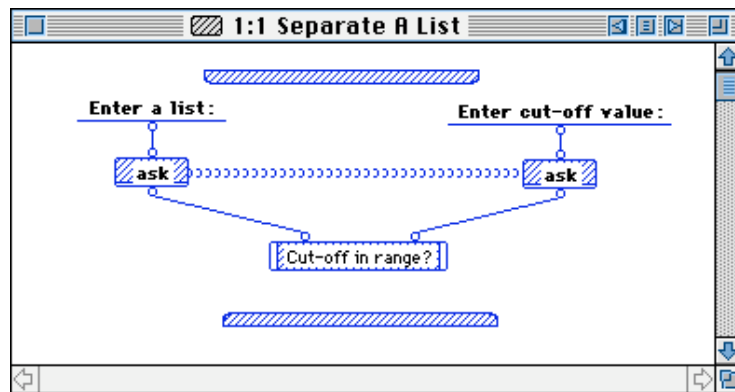


Figure 7.12: Adding a test for the cut-off value in the *Cut-off in range?* local method

The code within the *Cut-off in range?* local is shown in Figures 7.13-7.14. In the first case of the local, we first sort the list in ascending order from the smallest value to the largest using the `sort` primitive. This will make our range-checking much simpler. All we have to do to see if the cut-off value is within the range of numbers in the list is to compare its value to the smallest and largest numbers in the list -- the first and last elements of the sorted list. If the cut-off number is smaller than the first member of the sorted list or larger than its last value, it is not within the range of values in the list and the list need not be partitioned.

We use *match* tests to compare the cut-off value to the first element of the list (removed from the list with `detach-1`) and the last element of the list (removed with

`detach-r`). If either match succeeds, the second case of the local is entered to present an error message and the list is not partitioned (see Figure 17.14). If the matches fail (that is, the cut-off value is in the correct range), the Partition the list local method is entered in case 1.

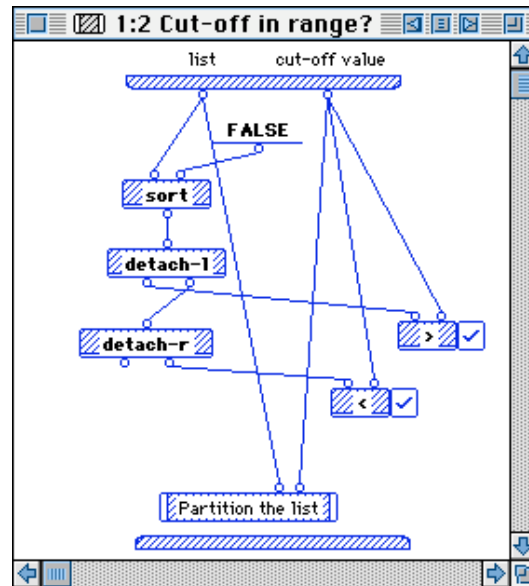


Figure 7.13: Checking if the cut-off value is within the range of the numbers in the list

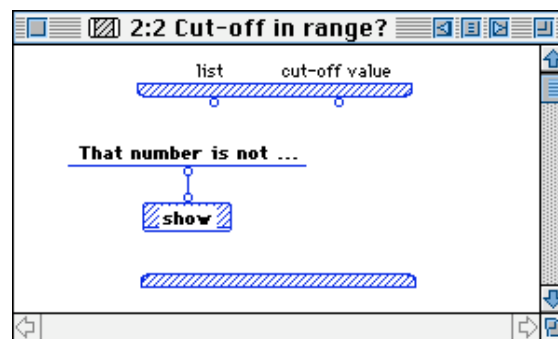


Figure 7.14: Error message if entered cut-off value is not in range of list

The Partition the list local method (Figure 7.15) splits apart the list according to the cut-off value. Under the input bar, create a new program element, and turn it into a < primitive to check if a value is less than the cut-off number. Feed the *list* of numbers into the left terminal node of the < primitive, and the cut-off number into its right terminal node.

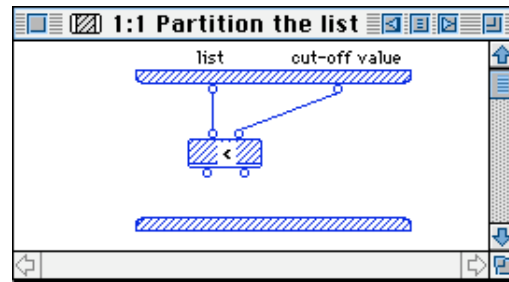


Figure 7.15: Initial code of the Partition the list local method

In order to make this primitive output two new *lists* according to whether or not each number in the original list is less than or greater than the cut-off number, we highlight the left terminal node of the < primitive (the one that accepts the list), then select the Partition item of the Controls Menu. Two *list-partitioning* root nodes appear on the < primitive (see Figure 7.16) -- one to output a list containing numbers *less than* (<) the cut-off value and the other containing all the remaining numbers, which are *greater than* the cut-off value.

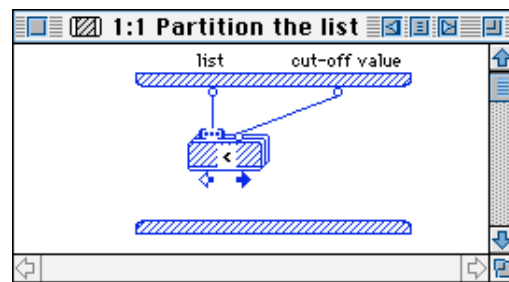


Figure 7.16: Partitioning the list

We feed each of these lists into two separate *show* primitives to display each list in separate dialog boxes. The prompts of the *show* primitives that are truncated in Figure 7.17 should read “The list members less than ” and “The list members greater than ”, respectively.

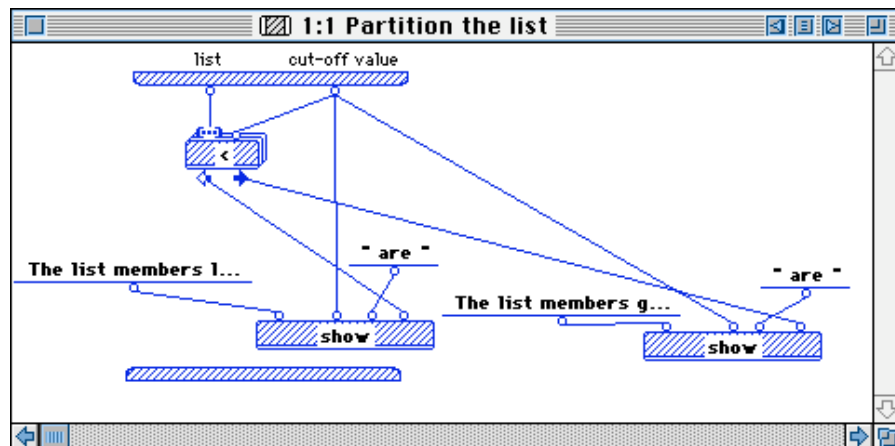
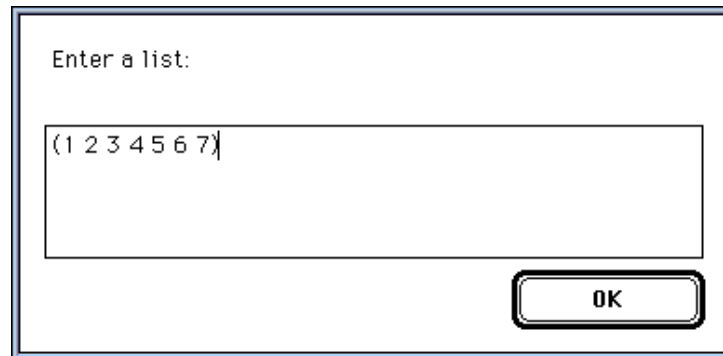


Figure 7.17: The completed Separate A List method

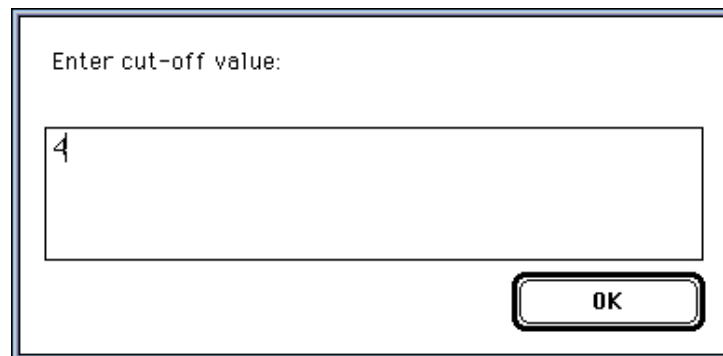
Let's see how the program works. When you execute the Separate A List method, a dialog box like that shown in Figure 7.18 prompts you to enter a list of numbers.



A dialog box with a title bar. Inside, the text "Enter a list:" is followed by a text input field containing the text "(1 2 3 4 5 6 7)". At the bottom right is an "OK" button.

Figure 7.18: Prompt to enter a list

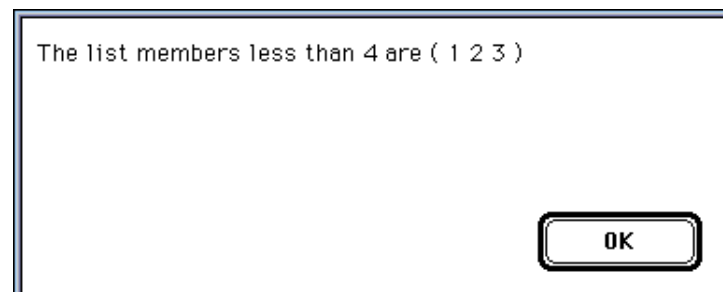
You are then asked for a cut-off value, the number at which you'll partition the list (Figure 7.19).



A dialog box with a title bar. Inside, the text "Enter cut-off value:" is followed by a text input field containing the text "4". At the bottom right is an "OK" button.

Figure 7.19: Prompt to enter a cut-off value at which to partition the list

The output of the program is shown below in Figure 7.20. The first output is a list of numbers from the original list which are less than your cut-off number. The second is a list of numbers equal to or greater than the cut-off number.



A dialog box with a title bar. Inside, the text "The list members less than 4 are (1 2 3)" is displayed. At the bottom right is an "OK" button.

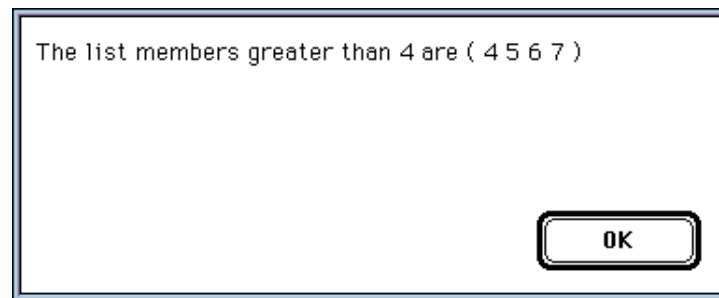


Figure 7.20: Outputs of the Separate A List program

Nested Lists (2-dimensional or higher-dimensional Lists)

In languages such C++, we are not limited to simple arrays containing just one number after another -- we can also create more complicated ***multi-dimensional arrays***. Let's give an example. A simple array might contain a number representing black (0) or white (1) for each pixel or dot in *one* row on the monitor screen. This is a *one-dimensional* array - it has only one index (pixel number within the row on the screen). It's just a single group of numbers for pixel 1, pixel 2, pixel 3, etc., as shown in Figure 7.21.

Element 1	Element 2	Element 3	Element 4	Element 5	...
0	0	1	0	1	...

```
short array[ NElements ];
```

Figure 7.21: One-dimensional array with its C++ equivalent

A more complicated array would itself contain *many* of these simple arrays -- one for *each* row of pixels on the screen. This is a *two-dimensional* array - it has *two* indices (pixel number within each row on the screen, and row number for each row). This array contains many groups of numbers - for row 1 pixel 1, row 1 pixel 2, row 1 pixel 3, etc., row 2 pixel 1, row 2 pixel 2, row 2 pixel 3, etc. (see Figure 7.22). This array could tell you the state (on or off) of the 23rd pixel in row 18 on the screen by accessing the array member using the indices 23 and 18 for the pixel number and row number, respectively.

	Column 1	Column 2	Column 3	Column 4	Column 5	...
Row 1	0	0	1	0	1	...
Row 2	1	1	1	0	0	...
Row 3	0	1	0	1	1	...
Row 4	0	0	1	0	1	...
Row 5	1	0	0	1	1	...
...

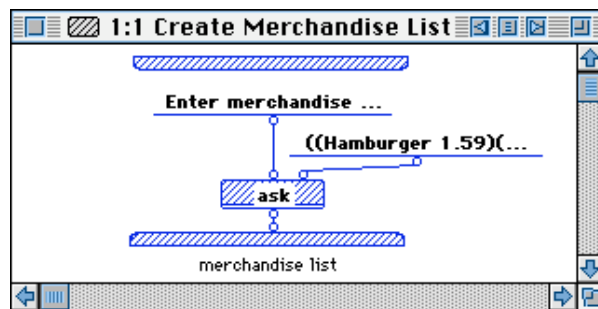
```
short array[ NRows ][ NColumns ];
```

Figure 7.22: Two-dimensional array with its C++ equivalent

Prograph provides a similar construct in its *lists*. We may *nest* lists together so that one list *contains* other lists (*sublists*). For example, a simple list may look like this: (1 2 3 4 5 6 7 8). It's a single list containing all eight numbers, one after another. A nested list would look like: ((1 2 3) (4 5 6) (7 8)). Its eight numbers are divided into three separate sublists. Each sublist can contain a different number of elements.

Let's write a program to handle a *nested or multi-dimensional list*. This program will contain a simple list of merchandise for a restaurant, where each sublist contains the name of the item and its price. The user of the program enters the list of merchandise, then may ask for a given item's price. While this may not be the ideal way to write a merchandise or inventory program (databases are much better suited to the task), it will illustrate how to access items within nested lists.

Create a new project named Merchandise List Project, then create a Merchandise List section. The first method for this program will ask the user to enter a two-dimensional list of items and their prices (see Figure 7.23). For our example program, we'll use a short list as a default input into the `ask` primitive so that you don't have to type in a lot of items. The list should read “((Hamburger 1.59)(Fries 0.89)(Shake 1.09))”. The method will output this nested list to be used by the rest of the program.

**Figure 7.23: The Create Merchandise List method**

The Ask For Item's Price method prompts the user to enter the name of the merchandise item for which a price is needed (Figure 7.24).

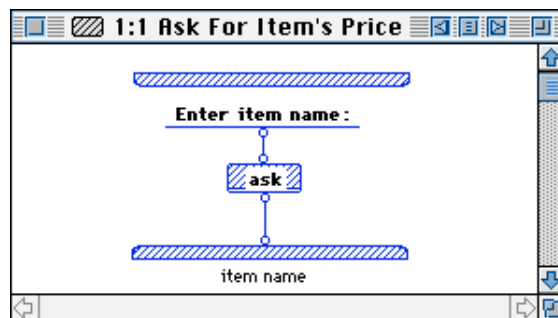
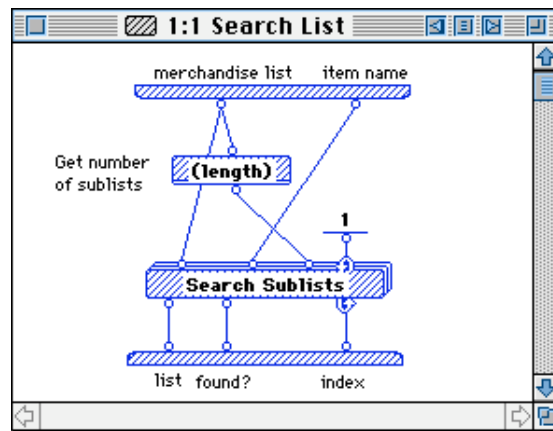
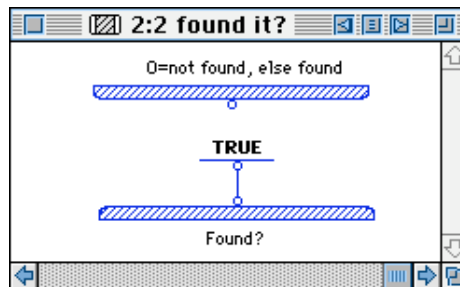


Figure 7.24: The Ask For Item's Price method

Once we are given the name of a merchandise item, we search the merchandise list for that item. The Search List method, depicted in Figure 7.25, does this for us. Search List gets the length of the merchandise list (how many sublists it contains), then enters a loop to search the sublists for an item with that name.

**Figure 7.25: The Search List method**

The Search Sublists method (Figure 7.26) forms the contents of the loop within the Search List method. This method simply uses the `get-nth` method to retrieve one sublist of the merchandise list, as indicated by the loop counter value. Next, it calls the `(in)` primitive to check if an item exists in that sublist with the requested name. Finally, a local method named `found it?` examines the output of the `(in)` primitive. This primitive returns the index of the item in the list (or sublist) it checks if that item was found; otherwise, it returns 0. Therefore, the `found it?` local “knows” that the item was found if the returned value from `(in)` was not zero, and that it was not found if that returned value is zero. The local method then outputs a Boolean value of `TRUE` or `FALSE` to indicate the results of this test. The Search Sublists method continues until it finds the requested merchandise item in one of the merchandise sublists or until the last sublist has been reached.



The Report Price method (Figure 7.27) will present the user with the price of the requested item. If this item is not found in the list, an error message is displayed. Note the difference in the way the `get-nth` primitive is called. Besides knowing which item to use within a sublist, `get-nth` must know which *sublist* inside the list contains that item. We do so by adding a third input node to `get-nth` to take in the sublist number. The `get-nth` primitive expects a sublist number if it has *more than two input nodes*; otherwise, a simple list containing no sublists is expected. That's all there is to it!

109

input the merchandise list as the first input of `get-nth`, then the sublist number (from the index parameter) as the second input, and the index within the sublist for the item name (item 1 in the sublist) and its price (item 2 in the sublist). These values are then presented to the user.

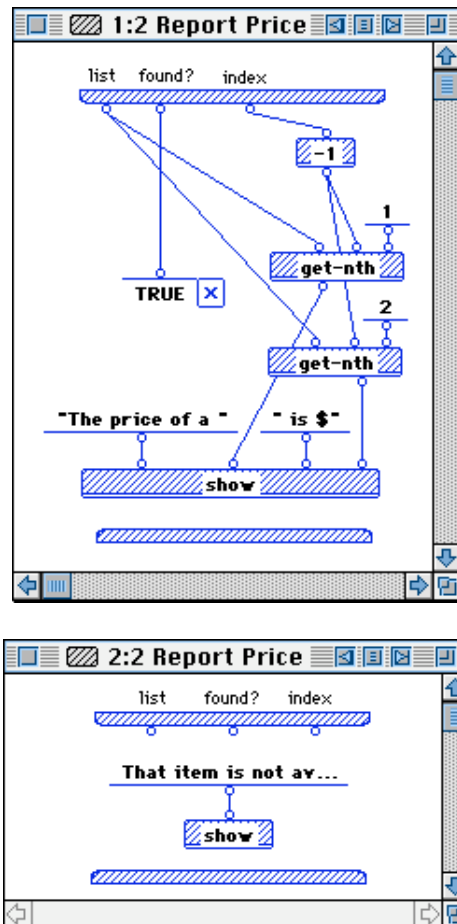


Figure 7.27: The Report Price method

Let's tie all of these methods together. We'll write a method called `Get One Price`, shown in Figure 7.28, which calls all of the methods that handle asking for an item's price, searching the sublist for that item, and reporting the price. At the beginning of this method we'll ask the user whether they want to find a price or quit the program. This is because we'll make the `Get One Price` method repeat itself, allowing the user to ask for any item's price in any order until they wish to quit.

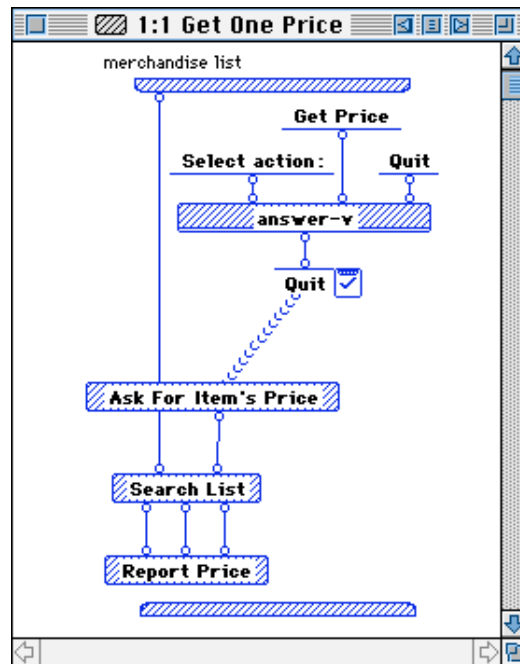


Figure 7.28: The Get One Price method

Finally, a main Merchandise List method (Figure 7.29) will coordinate the actions of this program by creating the list of merchandise, then entering the list-querying loop of Get One Price.



Figure 7.29: The Merchandise List method

Let's see how the program runs. After highlighting the Merchandise List universal method icon, select Run Method from the Execution Menu. You'll be prompted to enter the nested merchandise list as in Figure 7.30. The default list will be presented so just click the OK button or press the Return key to accept this list.

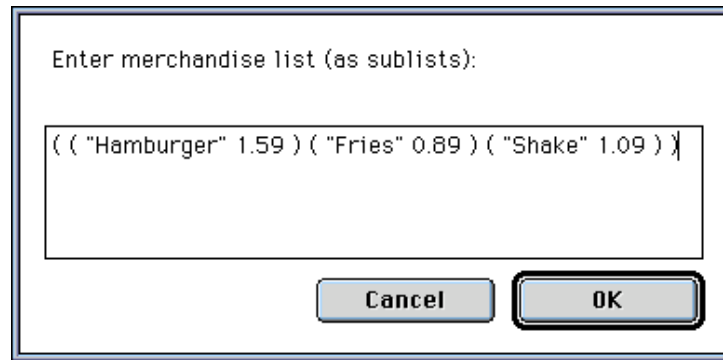


Figure 7.30: Prompt to enter a two-dimensional merchandise list

You'll then be prompted to decide what action to take next -- asking the price of an item or quitting the program (Figure 7.31).

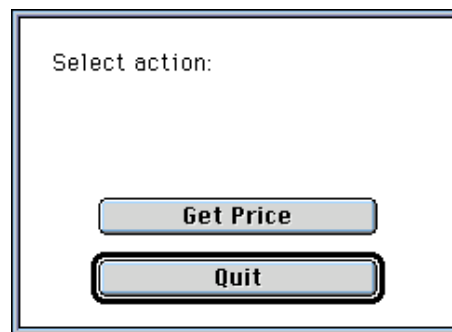


Figure 7.31: Prompt to select an action

Next, you'll be asked to the name of the item for which you want a price (see Figure 7.32).

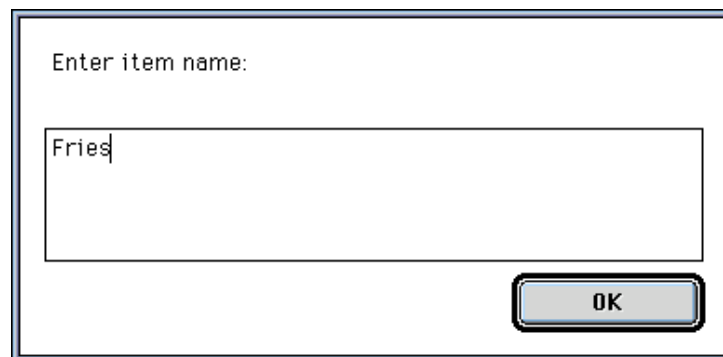


Figure 7.32: Prompt to enter a merchandise item's name

The final display, seen in Figure 7.33, will show the price of the item (in this example, the price within the second sublist). When you are asked again to select an action, you may either find another item's price or quit the program.



Figure 7.33: Output of the Nested Lists program

The Merchandise List program has shown us two different ways to access sublists: either you can extend the list-accessing primitives (`get-nth` or `set-nth`) by adding another input node to them that will accept the index of an item within a sublist, or you may use a loop to examine each sublist in turn. Either or both techniques may be needed in your applications.

Filling and Using a New List

The final lesson involving lists will show you how to fill up a list “on the fly” with a series of strings. The program will have you enter a list of names (each is a *string*), and then it will request the age (a *number*) of each person named. A second list will be formed automatically that will contain these ages. We will end up with two “parallel” lists with a one-to-one correspondence between the data of each list. That is, the first name of list one will correspond to the first age in list two, etc.

Create a new Names & Ages project, section and method. Complete the Names & Ages method as pictured in Figure 7.34. Notice the right terminal node of the Get Ages method. It contains an “*empty list*”, similar to the NULL value initially given to numbers and strings. Its symbol is simply two parentheses with nothing (no list) between them. The right terminal and the root node are converted into a *loop* so we can fill up the empty list with ages, one by one. We will start with the empty list, put an age into the list with Get Ages, then feed this single-element list back into the Get Ages method to put a second age into it, etc.

The left terminal node on the Get Ages method is a list-handling node that accepts a list of names from the `ask` primitive. This forces Prograph to *multiplex* the Get Ages method, meaning it will be repeated once for each name in the list -- that is, Get Ages will ask you for the age of each person in the name list, one by one.

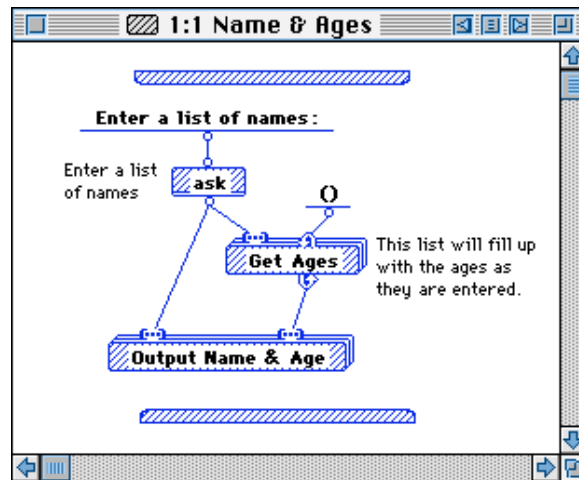
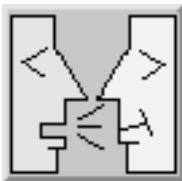


Figure 7.34: The Names & Ages method

The Get Ages method code is shown below in Figure 7.35. It accepts the name list and the existing list for storing the ages. On the first iteration through this method, the age list is empty. We prompt the user for the age of the first person in the name list, using the “join” primitive to place the person’s name in the prompt. The age entered is attached to the end of the age list with the attach-r primitive (attach this element to the right end of the list).



By The Way...

If we’d used the attach-l primitive, the new age value would be attached to the beginning (left end) of the list. Other primitives like (join) can join together two lists into a new, larger list, or tell you how many elements are in a list, such as the (length) primitive.

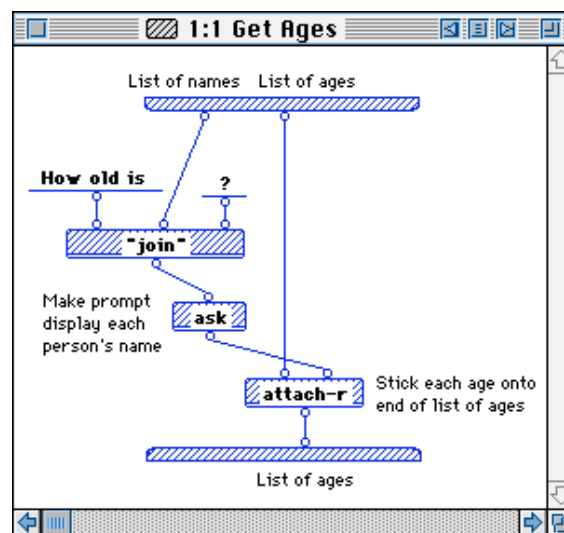


Figure 7.35: The Get Ages method

The Output Name & Age method (Figure 7.36) simply prints out the names and ages of each person in turn, using both the name and age lists.

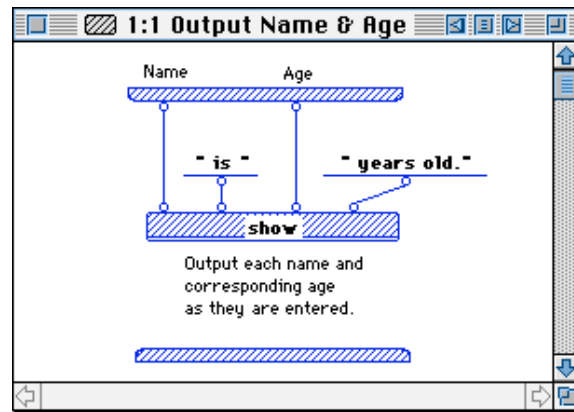


Figure 7.36: The Output Name & Age method

When the program is run, it asks for a list of names (see Figure 7.37).

Figure 7.37: Prompt to enter a list of names

For each name in the list, it will next ask for an age (Figure 7.38).

Figure 7.38: Prompt to enter the age of each person

Finally, it will display the name and age of each person in the list in turn. One of these displays is shown in Figure 7.39.

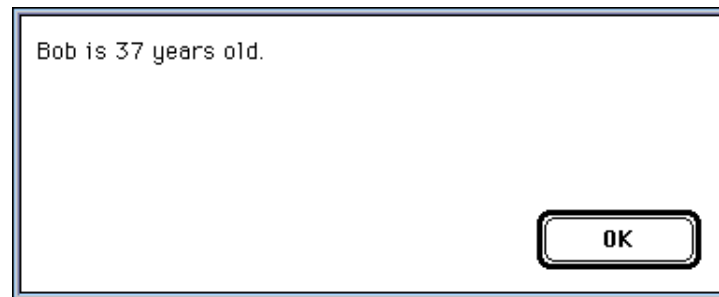


Figure 7.39: Output of the Name & Ages program for one person in the list

Exercise 7.2:

Write a program that does exactly the same thing, in the same order, as the Names & Ages program. Here's the catch -- use the `attach-l` primitive instead of `attach-r`! Hint: Use the `reverse` primitive to take your list and reverse the order of its elements.

Exercise 7.3:

Modify the Dice Game program so that when a dice throw other than 2, 7, 11 or 12 occurs, the user is presented with the message "Your point is <the dice throw>". The program should then keep track of that dice throw and ask the user to roll the dice again. If the current dice roll equals the stored "point" value, output the message "You made your point! You win!". If the current dice roll equals seven or eleven, output the message "You lose!". If the current dice roll is any other value, keep rolling the dice.

Summary

- The *list* is Prograph's closest relative to a C++ array. Like an array, it holds sequences of data. Unlike an array, its data does not need to be homogeneous - each data item in a list can be of a different data type. More importantly, list multiplexes give you a way of iterating an operation over an entire list without having to write a loop! While some of the functionality of Prograph lists may be obtained in C++, it requires purchasing extra code libraries since lists are not built into the C++ language.