

8. Applying Procedural Programming

Overview

Let's complete the section on procedural programming with a concrete example. In this chapter, we'll build an example programmer's calculator program that will convert numerical data from one number base system to another; that is, between decimal (base 10) to hexadecimal (base 16), octal (base 8), or binary (base 2). This program will build upon what we've learned so far about methods, strings, lists, loops and matches.

The Programmer's Calculator Program

The programmer's calculator program allows its user to convert numbers between four different number bases commonly used in programming tasks -- decimal, hexadecimal, octal and binary. With four possible number bases to choose from, we are faced with 2^4 or 16 possible conversions. This makes the programming task quite formidable. However, in the process of designing the program, we could reduce the number of possible conversions. This involves choosing one number base for an intermediate number storage format. In other words, all input numbers will be converted to this intermediate format, then this intermediate format is converted to the final output number. Let's say we used decimal (base 10) as our intermediate storage format. Then we'd need only 6 number conversions -- hexadecimal to decimal, octal to decimal, and binary to decimal to convert the input number, and decimal to hexadecimal, decimal to octal and decimal to binary to produce the output number. This reduces our programming task considerably.

The main method of our programmer's calculator reflects this design. In Figure 8.1, we show the main calling method Number Converter. The user selects the base of the number to be converted, the number itself, then the base to which the number should be converted. The program then converts the input number to decimal format, then to the final number base. The program ends by displaying the original number and the converted value.

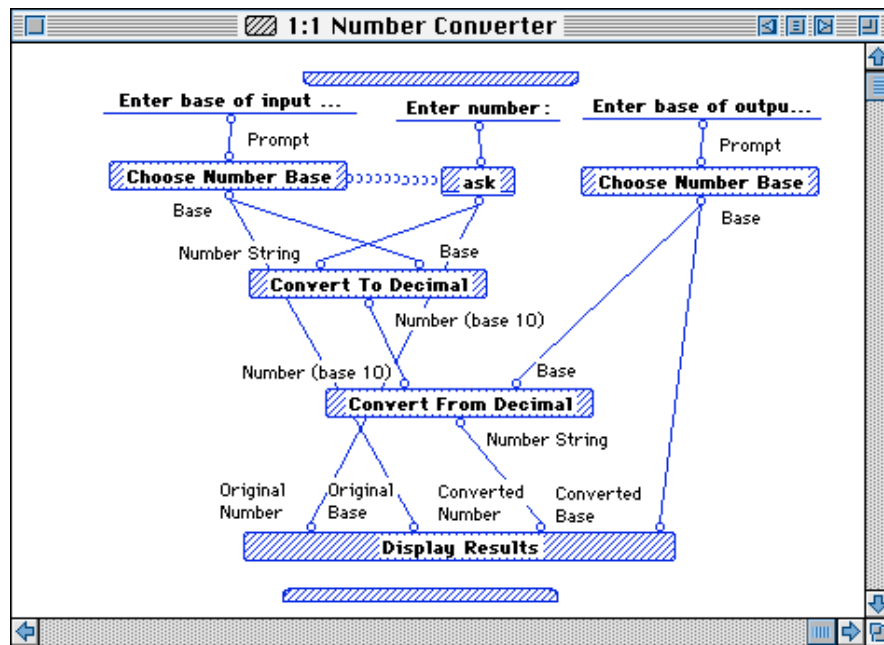


Figure 8.1: Number Converter method

The Choose Number Base method is quite simple (see Figure 8.2). The method is used twice, once to select the input number base and again to select the output number base. We therefore input the prompt presented to the user as an input to the method so the same code may be used twice, with a different prompt each time. This method presents the user with a set of four buttons to be selected by means of the `select` primitive. We provide the names of the buttons in a list. For this method, the list contains the four possible numerical bases -- decimal, hexadecimal, octal and binary. The output of the primitive and this method is the user's selection.

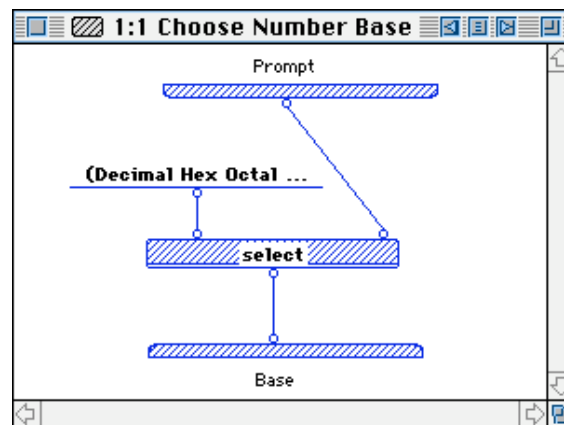


Figure 8.2: Choose Number Base method

Similarly, the Display Results method is simple, as shown in Figure 8.3. All it does is display the original input number and its base, and the final number and its base to the user.

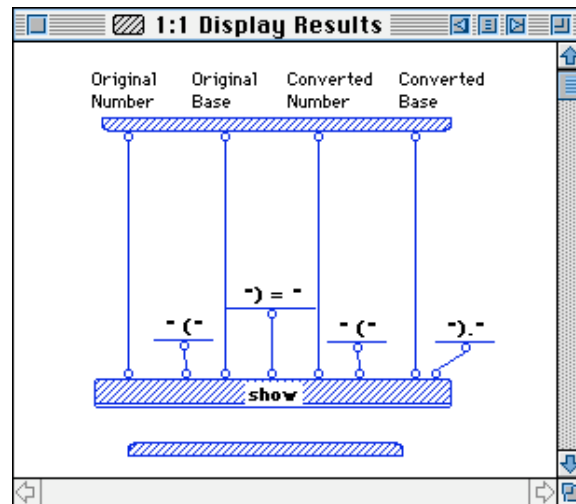


Figure 8.3: Display Results method

Converting Numbers to Decimal Format

The bulk of the work of the program is done in the Convert To Decimal and Convert From Decimal methods, so let's get to them now. The Convert To Decimal method accepts the output of an `ask` primitive that contains the number to be converted. This number can be in any of the four possible bases -- decimal, hexadecimal, octal or binary. However, the `ask` primitive only recognizes one type of number -- decimal. Any other number is interpreted as being a string. Therefore, we will convert all input numbers into string format before converting them into decimal numbers. As another safety check, we'll also strip out any spaces that the user may have typed in as well.

The first case of Convert To Decimal is shown in Figure 8.4. Here, we accept a *decimal* input number. We then convert it to a string, remove any extraneous spaces, then call the `from-string` primitive, a relative of the `to-string` primitive we used in Chapter 6. The `from-string` primitive is similar in function to the `atoi()` ANSI C library function, but much more versatile. It takes a string and converts it to any other appropriate Prograph data type, such as an integer, real, Boolean, list and so on.

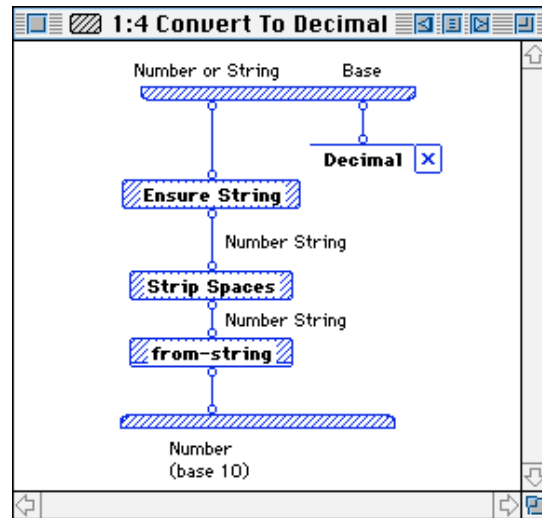


Figure 8.4: First case of the Convert To Decimal method

The second case of Convert To Decimal accepts a *hexadecimal* input number (Figure 8.5). If this input hexadecimal number does not contain any of the special characters *a*, *b*, *c*, *d*, *e* or *f* (hexadecimal equivalents of the numbers 10-15, respectively), the input number will be interpreted as a decimal number by the `ask` primitive. Otherwise, it will be interpreted as a *string* containing both digits and letters instead of a number. We therefore first make sure that it is in string format if it is not. We then remove any extraneous spaces in the string.

At this point, we need to convert the string containing the hexadecimal number into a decimal number. We'll call the `from-string` primitive to do this. But how will this primitive know that its input string represents a hexadecimal number instead of a decimal number? The Prograph language represents integers and numbers in string format in several ways. *Decimal* numbers are represented as strings containing digits alone. *Hexadecimal* numbers are represented as strings of numbers and the special characters *a-f* preceded by the substring "*16#*" to denote that the string is a hexadecimal number. *Octal* numbers in strings are preceded by the substring "*8#*", and *binary* numbers are preceded by "*2#*". In other words, a number in any base other than 10 (decimal) is preceded by the number base and a "*#*".

We therefore take our number string, and append the string "*16#*" at its beginning with the "*join*" primitive. We then use this modified string as the input to the `from-string` primitive. The `from-string` primitive will then convert it to its decimal-format equivalent number.

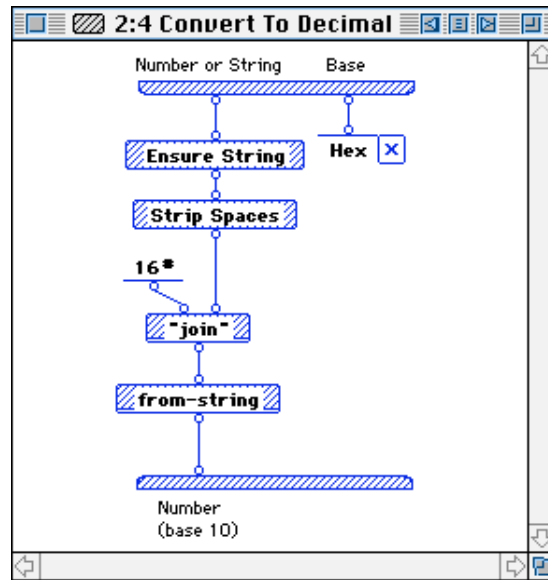


Figure 8.5: Second case of the Convert To Decimal method

Similar logic is used for the remaining two cases of Convert To Decimal (Figures 8.6-8.7). In the third case, used to convert an *octal* number to a decimal number, we precede the number string by "8#" before converting it to a decimal number. In the fourth case, we append "2#".

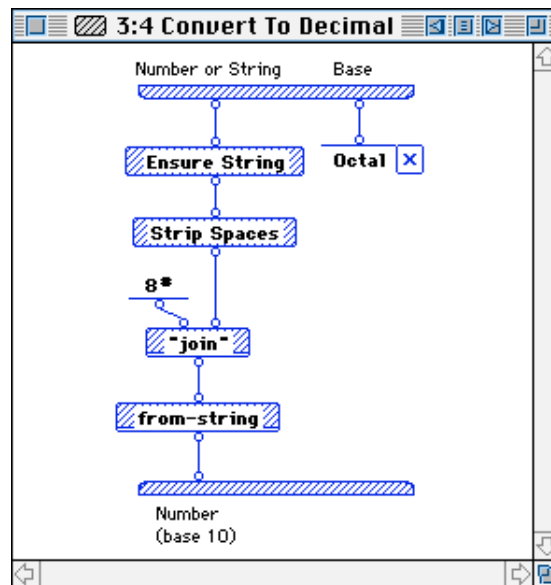


Figure 8.6: Third case of the Convert To Decimal method

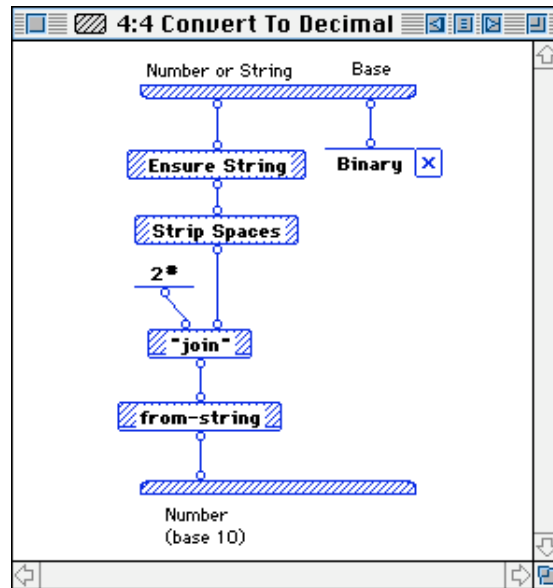


Figure 8.7: Fourth case of the Convert To Decimal method

Before we examine how these decimal numbers are converted back to other number bases, let's look at the Ensure String and Strip Spaces methods used in Convert To Decimal. The Ensure String method, shown in Figures 8.8 and 8.9, checks if its input, the output of an ask primitive, is a number or not. If it isn't a number, it must be in a string format already, so we need not do anything to it. If it is a number, we enter the second case of **Ensure String** and convert the number to a string.

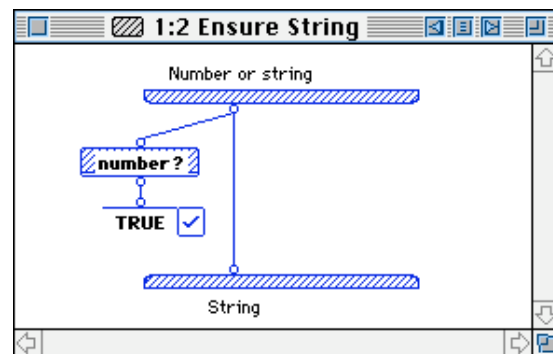


Figure 8.8: First case of the Ensure String method

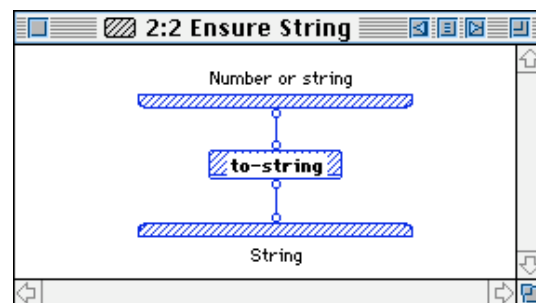
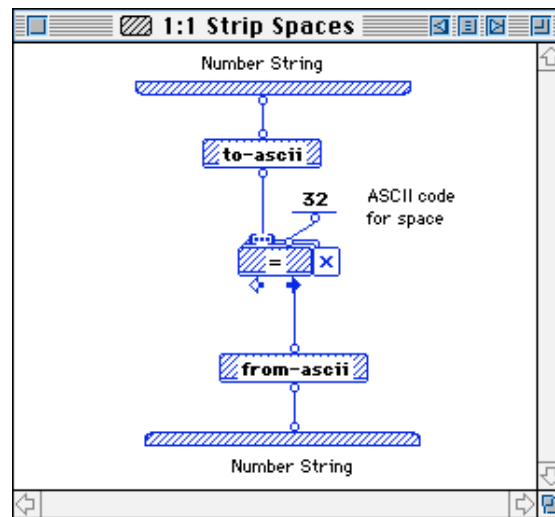


Figure 8.9: Second case of the Ensure String method

The Strip Spaces method is shown in Figure 8.10. This method takes advantage of the list-processing power of Prograph to simplify processing a string. First, the string is converted to a list of numbers -- numerical representations of characters in ASCII format -- with the `to-ascii` primitive. Next, we partition this list into two sublists. One contains all of the ASCII numbers that are equal to 32 (the ASCII equivalent of a *space* character), and the other contains the remaining characters. In other words, we've partitioned the list into spaces and the original string without the spaces. Finally, we convert the list without the spaces back into a string with the `from-ascii` primitive.

**Figure 8.10: The Strip Spaces method**

Reconverting Decimal Formatted Text to Numbers

Now that our original input number has been converted to a decimal number, we must convert it to the number base we desire as the output of the program. This is where the Convert From Decimal method comes in. If the desired number base is decimal, this method does not have to do anything at all (see Figure 8.11). It just passes the number along to be printed.

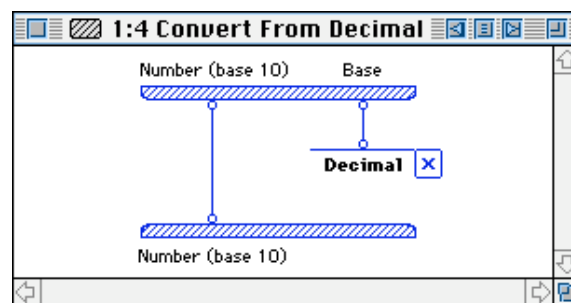


Figure 8.11: First case of the Convert From Decimal method

The remaining cases of the Convert From Decimal method are just a bit more complicated. Let's start with the second case of the method, which handles the conversion to a hexadecimal number (Figure 8.12). First, we convert the decimal number input into a string representing a hexadecimal number. We use the `format` primitive for this task. This primitive performs the same actions as the C language `printf()` string printing function's string formatting codes. We specify how we want the output string to appear by inserting special formatting codes into the input of the `format` primitive. The "16#" formatting code means "output a number as a string in hexadecimal form". The remaining 9's in the format string mean "output any hexadecimal digit or character in this position". Since we have eight 9's in the format string, our number string will be up to eight numbers or digits long.

The output of the `format` primitive is a string of the form "16#<the number>". We then strip off the leading three characters -- the substring "16#" -- to leave the hexadecimal number alone. There's still one problem. We've allocated space to display up to eight digits of the hexadecimal number. What if the number string needs fewer digits or characters? The `format` primitive will insert zeros in front of the number to fill up the rest of the number string. We'll use the Strip Leading Zeros, discussed below, to remove these padding zeros from the number string.

Notice that we leave the number in string format instead of converting it back to numerical format. This is because converting back to numerical format would revert the number to a decimal form. Besides, all we are doing with the number is displaying it, so there's no harm in leaving it in string format.

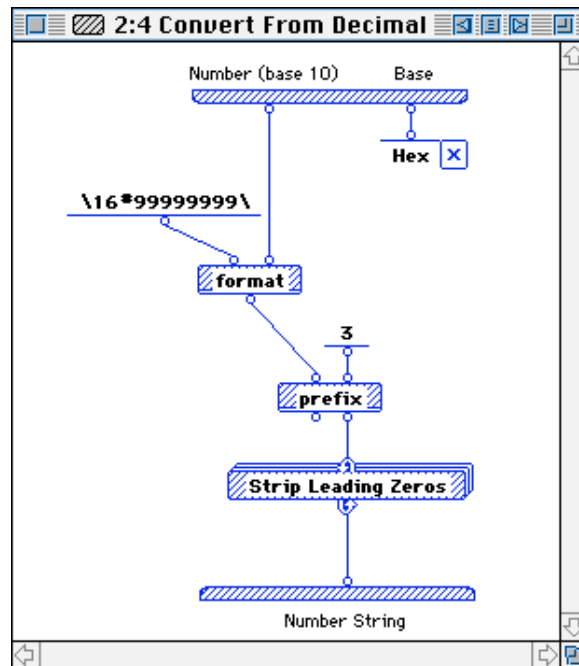
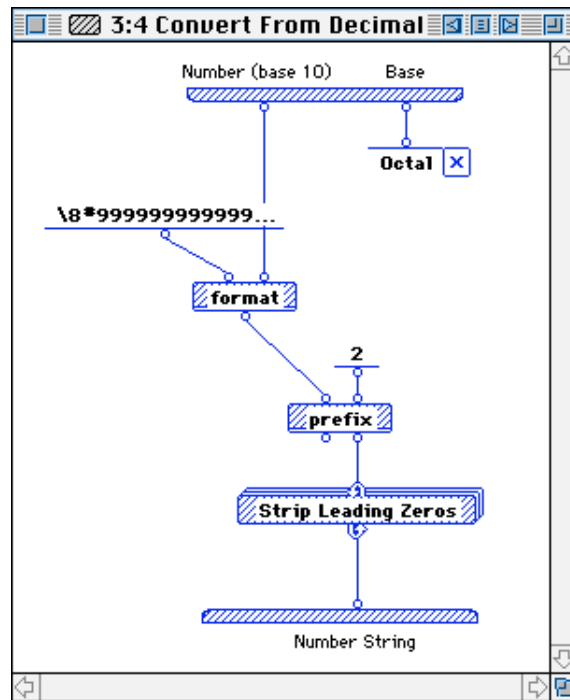


Figure 8.12: Second case of the Convert From Decimal method

Cases 3 and 4 of Convert From Decimal are similar. In the third case of the method for conversions to the octal format (see Figure 8.13), we begin the formatting string for the `format` primitive with “8#” to denote an octal string, and leave a little more room in the output number string for the longer converted octal number by placing more 9’s in the format string. Since the leading format code is “8#”, we need strip only two characters from the string with the `prefix` primitive.

**Figure 8.13: Third case of the Convert From Decimal method**

In the final case for conversions to the binary format (Figure 8.14), we begin the formatting string for the `format` primitive with “2#”, and place several 9’s in the format string.

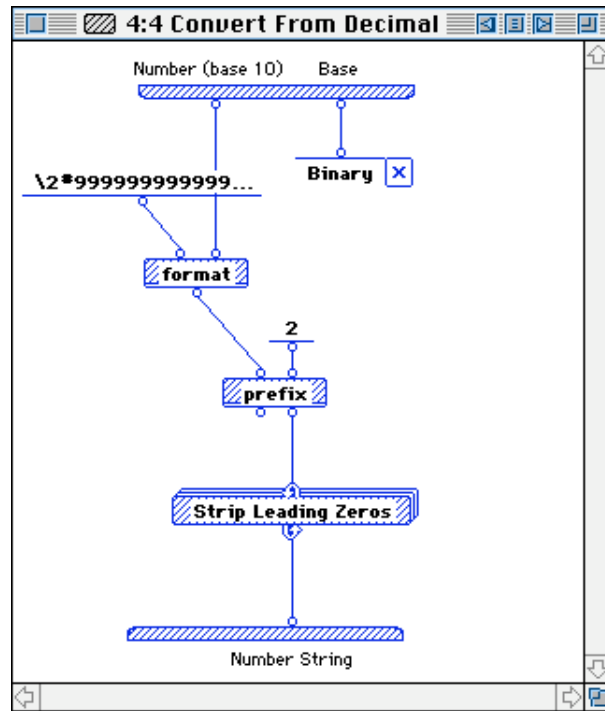


Figure 8.14: Fourth case of the Convert From Decimal method

The last method of the programmer's calculator program removes the leading zeros that the `format` primitive places in front of short numbers. The `Strip Leading Zeros` method is called as a *repeat* by the `Convert From Decimal` method. This repeating method first checks if there are any leading zeros in the number. The `"in"` primitive checks for zeros in the string and returns the position of the zero in the string. If the returned position is 1, the first character in the number string was a zero. It then uses the `prefix` primitive to remove that zero, and proceeds to the next iteration of the repeat. If a leading zero was not present, the number string is returned unchanged and the repeat is exited.

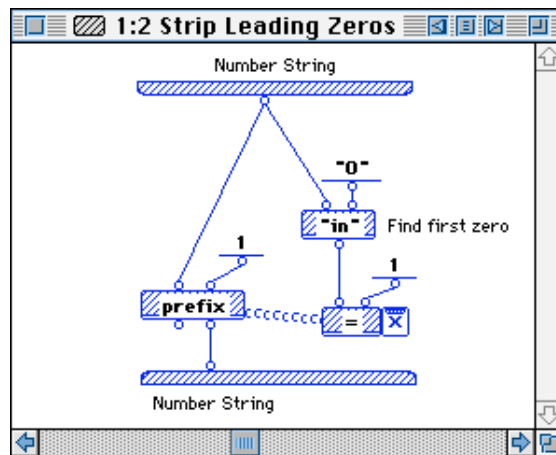


Figure 8.15: First case of the Strip Leading Zeros method

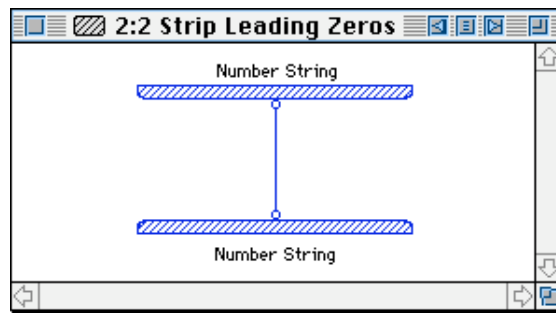


Figure 8.16: Second case of the Strip Leading Zeros method

Summary

In this chapter, we combined the knowledge we gathered in the previous chapters to create a useful Prograph program that converts numbers from one number base to another. We used structured program design (general methods containing more and more specific methods), program flow control, strings and lists.

- **Matches** were used to check the number base of the input and output number bases and execute the appropriate code for each number base.
- A **repeat** loop was used to remove leading zeros from the final output number (in string form) by searching the leading character of the string for a zero, removing it, then checking the remaining string again.
- **Lists** were used to remove spaces from strings. Strings were converted to lists of numbers, spaces were partitioned from the lists, then the lists were converted back to strings. The special conversion primitives `to-ascii` and `from-ascii` helped here.
- **Strings** were used to hold sequences of digits (and special characters used in hexadecimal numbers). Special string conversion primitives (`from-string`, `format`) converted numbers to and from string format.