Scalar Functions
Summary

## 1. Introduction

This tutorial is for the reader who has little or no experience with programming in APL-like programming languages. The scalar functions exhibit many of the qualities that make A+ an effective programming language, while at the same time being generally familiar, so this is a good place to start.

As you will learn, many program fragments in A+ can be stated in a few expressions that do not include explicit control statements. This expressiveness sometimes depends on using A+ primitive functions in ways that are not apparent from their definitions. So, don't expect to see all or even most uses of the A+ primitive functions in these tutorials. The tutorials are meant to introduce you to the language and some of its uses, but when you're done many surprises will still lie ahead. Who would want it otherwise?

This tutorial is made up of textual descriptions and A+ examples. To execute the examples conveniently you should set up your Emacs environment to have two visible buffers, one holding the tutorial and the other an A+ session. If you are currently reading this in Emacs, simply press F4.

The examples consist of indented A+ expressions and their values. Sometimes the text refers to the values, and it is helpful to have them handy, but the main reason they are displayed is to help you become familiar with A+ in the Emacs environment. In the later tutorials the values usually do not appear in the text. You should evaluate the indented expressions, and compare the results in the A+ session with those in the text.

For example, when you see a display of the form

```
1 2+3 4
 4 6
```

the indented expression 1 2+3 4 is the one to bring into the A+ session for evaluation. The second expression, 4 6, is the result of executing the indented expression and should not be brought into the A+ session for evaluation.

To bring individual expressions from the tutorial into the A+ session, place the cursor on the expression and press F2. When an expression is brought into A+ in this way, it is automatically evaluated, and its value is displayed. The displayed value should be identical to the display in the tutorial.

If you need more help on running Emacs and A+, see the "Getting Started" tutorial. If you want to try your hand at devising your own A+ expressions, see the keyboard layout diagrams in Chapter 2 and Appendix B of the A+ Reference Manual.

## 2. Constants

### 2a. Numeric Constants

There are several preliminary things to discuss. First of all, a list of numeric constants can be entered on a line by separating the individual numbers with blanks, as in

```
3 4.5 7
 3 4.5 7
```

This is a list of the three numbers, 3, 4.5 and 7. Secondly, negative numbers are denoted by a raised minus sign. For example:

```
5+¯2
 3
```

Exponential notation can be used for fractional, or floating-point, numbers, as in

```
1.23e5
 123000
```

which is 1.23 times 10 to the 5, or 123000. Negative exponents are denoted with the usual minus sign, not the raised minus sign, so that

```
¯2000e-2
 ¯20
```

is negative 2000 times 10 to the negative 2, or negative 20.

2b. Character Constants

A single character is entered surrounded by quotation marks, and a list of characters is entered in exactly the same way. For example:

```
'axcdert'
 axcdert
```

Either single quotes or double quotes can be used, but they must both be the same. For example:

```
"axcdert"
axcdert
```

It is interesting to see how a quotation mark can be included in a list of characters. If a single quotation mark is to be included, surround the list with double quotes; if a double quote mark is to be included, surround the list single quotes. For example:

```
'He said, "Oh no!"'
He said, "Oh no!"

"I can't go."
I can't go.
```

The question then becomes, how are both quotation marks included in the list? The answer is that the one that is identical to the quotes surrounding the list must be doubled. For example:

```
'"I can''t go either," he said.'
"I can't go either," he said.
```

2c. Symbol Constants

A symbol constant is of the form backquote,characters. An example is `ThisIsASymbol. It provides a form that enables rapid comparison and is convenient for aggregation. A vector of symbols can be written simply by juxtaposing them, with or without blanks between them:

```
`s1 `s2`s3   `s4 `s5`s6
```

```
 `s1 `s2 `s3 `s4 `s5 `s6
```

2d. Nested Constants

In addition to numeric, character, and symbolic data, A+ has nested, or boxed, data. Nested data comes in lists, like numeric, character, and symbolic data, except that the elements are not restricted to individual numbers, characters, or symbols; they can be anything. For example:

```
(2 3.5 ¯7;'dfgtyuh')
<  2 3.5 ¯7
< dfgtyuh
```

This constant has two elements. The < in the display indicates that what follows is a boxed element. An easy way of producing nested arrays is to use strand notation, as in the example. It is of the form (...;...;...) and its value is a list, or vector. The successive elements of this list are the enclosures, or boxed forms, of the successive expressions between semicolons or between a semicolon and a left or right parenthesis.

## 3. Scalar Functions

For convenience, the scalar functions have been divided into categories that may be familiar to you. The arithmetic functions are Plus, Minus, Times and Divide. The relational functions are Equal to, Not equal to, Less than, Less than or Equal to, Greater than, and Greater Than or Equal to. The logical functions are And, Or, and Not. The elementary functions are Power, Log, and certain trigonometric and algebraic functions. Finally, there is a miscellaneous set. The names and symbols of the scalar functions are summarized in Table 1, at the end of the tutorial.

The expressive power of the scalar functions in A+ is largely due to useful variations provided by operators. "5. Operators" introduces Reduction, Scan, Outer Product and Inner Product. Table 2, at the end of the tutorial, tells which of these operators apply to which scalar functions.

## 4. Arithmetic Scalar Functions

The common arithmetic operations are examples of scalar functions in A+. They are Plus (+), Minus (-), Times (×), and Divide (÷). They

apply to individual numbers, as in:

```
3+4
 7
```

and they apply to lists in an element-by-element fashion. For example:

```
      2 5 ¯7 + 10
 12 15 3
```

In this example the three numbers, 2, 5, and ¯7, are added to 10. The result is a list of three numbers. The spaces around the symbol + are meant as an aid to readability and are not necessary.

A list of numbers can appear on the right of the operation symbol as well as the left, as in the following subtraction example:

```
      200 - 27 ¯34.56 1.521e2 129
 173 234.56 47.9 71
```

and there can be lists on both sides of the symbol:

```
      2 7 9 × 10 ¯3 4
 20 ¯21 36
```

When there are two lists, i.e., more than one element on each side, the number of elements must be the same on each side.

The operation of negation is denoted by - in mathematics, and in A+ as well. For example:

```
      - 3 6.7 ¯12 102 1e-3 ¯3.4e3
 ¯3 ¯6.7 12 ¯102 ¯0.001 3400
```

The use of one symbol to denote two operations, such as the - symbol for subtraction and negation, is carried over in A+ to almost all its primitive operations. For example, ÷ denotes both division and the taking of reciprocals, as in:

```
      2 ¯3 10 ÷ 5 6 5
 0.4 ¯0.5 2
```

and

```
  ÷ 5 10 12
```

```
 0.2 0.1 0.08333333333
```

The number or list of numbers that appears to the left of a symbol is called the left argument of the operation. That which appears to the right is called the right argument. The two together are referred to as the arguments of the operation.

When there are two arguments, the operation is said to be dyadic. When an operation has only argument, when nothing appears to the left of its symbol that could serve as the left argument, it is said to be monadic. Symbols, such as -, that have both a monadic and dyadic use are called ambivalent. Most primitive function symbols in A+ are ambivalent.

It is time to adopt some conventional A+ terminology. The common mathematical term for +, -, × and ÷ is operation. Operations are examples of the general mathematical notion of function, but the term operation is quite commonly used in mathematics when the symbol for the dyadic function appears between the arguments. However, in A+ we tend to use the more general term "function", even for the arithmetic operations. This may be due to the fact that A+ also has operators, another common mathematical concept, and the use of two closely related terms can be confusing. Whatever the reason, we will adopt the A+ convention, and from now on we will refer to the arithmetic operations as functions, and in particular as scalar functions.

Ex 1. A mistake that is often made concerning numeric constant lists such as

```
1 5 ¯12 21
```

is to attach functional significance to the spaces between items. Now that we have seen how scalar functions are evaluated, we can examine this common pitfall. For example, one might expect to be able to form the list consisting of 5 plus 23, 24 minus 13, and 10 times 6 simply by putting these expressions on a line separated by blanks, as in:

```
5+23 24-13 10×6
```

To understand how this expression is evaluated you need to know the following (see the Syntax tutorial for a general discussion):

```
    the first thing done in evaluating this expression is to form
       the constant 13 10;
    the next thing done is to form the constant 23 24;
    next, the function × is evaluated;
```

```
    then -;
    and finally +.
```

Use these rules and your knowledge of scalar functions to evaluate this expression by hand. Check your answer by bringing this expression into the A+ session for evaluation with F2.

The next common expectation is to be able to put each of the expressions 5 plus 23, 24 minus 13, and 10 times 6 in parentheses, and then put these three parenthesized expressions on a single line separated by blanks, as in:

```
(5+23) (24-13) (10×6)
```

But this is an invalid expresson, and an attempt to evaluate it will result in an error message:

```
PARSE +...  : var?
```

or, in Version 3,

```
⍎[parse] +... : var?
```

To put the results of individual expressions like these three together in a list, you must use the function designed for that purpose, which is called Catenate and is denoted by the comma. Use F2 to evaluate the following expression:

```
(5+23),(24-13),(10×6)
```

Ex 2. Write two expressions for a 5-element list whose first two elements are those of the list 14+4 8 and whose last three elements are those of the list 4 9 12-3 10 7.

## 5. Operators

In mathematics, operators are objects that act on functions to produce new functions. A+ has several operators that act on scalar functions. We will introduce them here for the arithmetic functions, and return to them later as more scalar functions are introduced.

### 5a. Reduction

The slash (/) denotes an A+ operator called reduction. For example,

+/ denotes + reduction, and ×/ denotes × reduction. The function symbol always appears to the left of the slash. The functions +/ and ×/ are called derived functions because they are derived from other functions, namely + and ×. A hint at how these derived functions are defined is that +/ is also called summation.

```
+/3 5 8 12
 28
3+5+8+12
 28
+/1 2 3 4 5 6 7 8 9 10
 55
1+2+3+4+5+6+7+8+9+10
 55
```

That is, +/ takes the sum of all elements in a list. The reduction operator applies uniformly to all functions whose symbols are permitted to the left of the slash (see Table 2), so you should suspect that × reduction takes the product of all elements in a list.

```
×/3 5 8 12
 1440
3×5×8×12
 1440
  ×/1 2 3 4 5 6 7 8 9 10
 3628800
  1×2×3×4×5×6×7×8×9×10
 3628800
```

5b. Outer Product

The symbols ∘. denote the operator called outer product. ∘.+ is called the outer product of + and ∘.× is called the outer product of ×. The outer product operator is sometimes called the table maker, and here is an example:

```
1 6 ¯2 3∘.+4 10 3
  5 11  4
 10 16  9
  2  8  1
  7 13  6
```

When you execute the above outer product expression you will see the four separate lines displayed below the expression. These four lines are the rows of a table. You will note that the numbers in these rows

line up vertically. The aligned vertical lists are the columns of the table. This table contains all combinations of sums of elements from the list on the left with elements from the list on the right. For example, the number at the third row, second column of the table is 8. By definition, it is the sum of the third element of the list on the left, ¯2, and the second element of the list on the right, 10. Pictorially:

```
 + │  4 10  3
───┼─────────
 1 │  5 11  4
 6 │ 10 16  9
¯2 │  2  8  1
 3 │  7 13  6
```

Like reduction, the outer product operator applies uniformly to all functions whose symbols are allowed to the right of the ∘. (see Table 2). For example, the tables produced by ∘.× contain all combinations of products, as in:

```
1 6 ¯2 3∘.×4 10 3
   4  10   3
  24  60  18
  ¯8 ¯20  ¯6
  12  30   9
```

5c. Scan

The backslash (\) denotes the A+ operator called scan. Scan is a companion of reduction: for example, while + reduction gives the sum of a list of numbers, + scan gives the running, or partial, sums:

```
+\1 3 ¯5 2 6
 1 4 ¯1 1 7
+/1
 1
+/1 3
 4
+/1 3 ¯5
 ¯1
+/1 3 ¯5 2
 1
+/1 3 ¯5 2 6
 7
```

Note that something new about reduction has come up. Namely, + reduction of a single number is that number. In fact, this is true for any reduction. This is a separate definition from the one for lists with at least two elements, because there aren't enough elements to put the function symbol between.

## 5d. Inner Product

The fourth and last operator to be introduced here is inner product, denoted by the dot (.). Unlike the other operators, inner product applies to a pair of scalar functions. For example, the inner product of + and × is denoted by +.×. The function +.× is dyadic, and when applied to lists, as in a+.×b, is equivalent to +/a×b. For example:

```
34 ¯1 5 12+.×3 14 10 ¯5
 78
```

and

```
34 ¯1 5 12×3 14 10 ¯5
 102 ¯14 50 ¯60
  +/ 102 ¯14 50 ¯60
 78
```

Inner products also apply to tables, and when +.× is applied to tables it is equivalent to the ordinary matrix product in linear algebra.

## 6. Selection and Assignment

Variables can be assigned values by placing their names to the left of the assignment arrow and the values to the right. For example:

```
a←12 ¯23.921 83214      ⍝ No value is displayed for assignment.
```

Enter the name alone on a line, press the Return key, and the value will be displayed:

```
a
 12 ¯23.921 83214
```

Once a variable has a value individual elements can be replaced. For example:

```
a[1]←1024.7             ⍝ No value is displayed for indexed assignment.
```

```
  a

 12 1024.7 83214
```

Individual elements can be selected as well:

```
a[1]

 1024.7

  a[2 1]

 83214 1024.7 ⍝ Elements can be selected in any order.
```

Note that for the purposes of selection and assignment, the first element is element number 0, the second element is element number 1, and so on. These numbers are called the indices of the elements. For example, in the above list a:

12 is the element at index 0
1024.7 is the element at index 1, and
83214 is the element at index 2.

Note that not only can elements be replaced, but the entire value can also be replaced:

```
a←'x'                    ⍝ No value is displayed for assignment.
  a
x
```

## 7. Relational Scalar Functions

The mathematical relations (=, ≠, <, ≤, >, ≥) are ordinary scalar functions in A+. They apply to lists of numbers in the same way as the arithmetic functions. They return boolean results, i.e., the numbers 0 and 1, where 1 indicates that the relation holds, and 0 that it does not. For example:

```
3 10<7 ¯4
 1 0
```

The value 1 0 expresses the fact that 3 is less than 7 but 10 is not less than ¯4. Equal to and Not equal to apply to lists of characters and nested lists as well as lists of numbers. For example:

```
'a'='xcade'
 0 0 1 0 0
'a'≠'xcade'
 1 1 0 1 1
(1 2 3;'abcdef')=(1 2 3;'fed')
 1 0
```

The advantage of the relations' being functions is that they can then be used in ordinary, mathematical-like expressions. For example, to evaluate the sum of the positive numbers in a list named w:

```
w←1 2 ¯4 12 0 ¯7 14 2 ¯3
+/(w>0)×w
 31
```

Here is how it works. The subexpression w>0 produces a list of 1's and 0's; the 1's appear wherever the elements of w are positive. Then since 1 times any number is that number and 0 times any number is 0, an element of (w>0)×w is equal to the corresponding element of w when the latter is positive, and 0 when it is 0 or negative. Since 0's do not contribute to a sum, +/(w>0)×w is the sum of the positive elements of w.

```
w>0
 1 1 0 1 0 0 1 1 0
(w>0)×w
 1 2 0 12 0 0 14 2 0
  +/(w>0)×w
 31
```

Note that the last expression is equivalent to the following inner product:

```
(w>0)+.×w
 31
```

The outer product operator also applies to relational functions as well as the arithmetic ones. For example, we can answer the following question: how many times does each character in a list occur in that list? The answer for:

```
x←'ancahac'
```

is 3 1 2 3 1 3 2. That is, reading the list of characters from left to right, there are three a's, one n, two c's, three a's again, one h, three a's again, and two c's again. (The matter of removing

repetitions from the list of character counts is discussed in the "Simple Arrays" tutorial).

We can use the outer product of = to compare every character of x to every other character.

```
x∘.=x
 1 0 0 1 0 1 0
 0 1 0 0 0 0 0
 0 0 1 0 0 0 1
 1 0 0 1 0 1 0
 0 0 0 0 1 0 0
 1 0 0 1 0 1 0
 0 0 1 0 0 0 1
```

The following display shows how the elements of x are related to the elements of the outer product result.

```
= │ a n c a h a c
── ──────────────
a │ 1 0 0 1 0 1 0
n │ 0 1 0 0 0 0 0
c │ 0 0 1 0 0 0 1
a │ 1 0 0 1 0 1 0
h │ 0 0 0 0 1 0 0
a │ 1 0 0 1 0 1 0
c │ 0 0 1 0 0 0 1
```

If we look down any column of the outer product table, we see 1's whenever a character in the list x matches the character at the head of the column. If we sum the 1's in that column, we get the number characters in x that match the character at the head of the column. Of course the sum of the 1's in a column is simply the sum of the column because the other elements are 0. Therefore, + reduction of the columns gives us the answer:

```
+/x∘.=x
 3 1 2 3 1 3 2
```

You may have noticed that the table is symmetric. That is, we could just as well have summed the rows of the table to get the result. However, reduction applies to the columns by default. We will see how to apply reduction to the rows of a table in the "Simple Arrays" tutorial.

## 8. Logical Scalar Functions

The logical functions are And and Or, and are denoted by ∧ and ∨. They apply to boolean values and produce boolean values. They are defined as follows:

```
0∧0 equals 0 0∨0 equals 0
0∧1 equals 1∧0 equals 0 0∨1 equals 1∨0 equals 1
1∧1 equals 1 1∨1 equals 1
```

We say that the result of ∧ is "true" (i.e., 1) if both its arguments are true; the result is "false" if at least one argument is false. Similarly, the result of ∨ is true if at least one argument is true; the result is false if both arguments are false.

The logical functions are often used to form compound relational expressions. For example, the following expression tests whether or not one of the two numbers c and b is greater than 10 (first giving c and b values so that the test can be evaluated):

```
c←5
  b←11
(c>10)∨(b>10)
 1
```

If you wanted to know whether or not both c and b are greater than 10, the test expression would be:

```
(c>10)∧(b>10)
 0
```

Note: the way to read A+ expressions is from left to right. For example, the above expressions is "c greater than 10 and b greater than 10."

If either c or b (or both) are lists the situation is more complicated because the value of the expression will be a list of boolean values stating whether the individual elements of c and b satisfy the test. For example:

```
c←5 21 ¯7 50
b←2 ¯10 20 17
(c>10)∨(b>10)
 0 1 1 1
(c>10)∧(b>10)
```

```
 0 0 0 1
```

Consequently, depending on the problem at hand, you will usually want to know if all the individual relations hold , or if at least one does. However, many other conditions can arise, such as knowing that at least two of the individual relations hold. Using only what we know now about A+, we can determine the number of individual relations that hold by summing the boolean lists (the sum of a boolean list is the number of 1's in the list, which is the number of individual relations that hold).

To sum the individual relations in the above expressions:

```
+/(c>10)∨(b>10)
 3
+/(c>10)∧(b>10)
 1
```

To test whether or not at least one of the individual relations holds:

```
1≤+/(c>10)∨(b>10)
 1
1≤+/(c>10)∧(b>10)
 1
```

(Reading the last expression from left to right: "1 less than or equal to the sum of the relation: c greater than 10 and b greater than 10.")

To test whether or not at least two of the individual relations hold:

```
2≤+/(c>10)∨(b>10)
 1
2≤+/(c>10)∧(b>10)
 0
```

Testing whether or not all the individual relations hold can be done in the same general way as above, but it's a little trickier. Besides, it's time to introduce another way to do all these tests (see "8a. Reduction Revisited".)

Before continuing with the reduction examples, there is a third logical scalar function to be introduced here, which is the monadic function called Not, or logical negation, and denoted by ~. The value of ~0 is 1 and that of ~x for any nonzero integer x is 0. Both ∧ and ∨ also apply to all integers, not just 0 and 1, but the extended definitions are different in Versions 2 and 3. For Version 3 they are

analogous to Not. For Version 2 see the March 1994 A+ Reference Manual.

## 8a. Reduction Revisited

The reduction operator applies to both ∧ and ∨. Like plus and times, ∧ reduction can be defined by putting ∧ between the consecutive elements of a list, as in:

```
∧/1 1 0 1
 0
  1∧1∧0∧1
 0
```

∨ reduction is similar:

```
∨/1 1 0 1
 1
1∨1∨0∨1
 1
```

In order to understand ∧ and ∨ reduction, consider evaluating + reduction by placing the + symbol between all pairs of consecutive elements of a list, as follows:

```
a+b+c+d+e+f+g
```

Compute the sum any pair of consecutive elements, and then replace the + symbol and the two summands with the result. (Any pairs can be used because all the functions that are permitted in reductions are associative.) For example, let h be d+e, so the above expression becomes:

```
a+b+c+h+f+g
```

We now have a list of the same form as the original, but with one fewer element and one fewer + symbol. If the process is repeated, the same thing happens, and eventually all that is left is a single element and no + symbols. That single element is the result of the + reduction.

The same procedure works for evaluating ∧ reduction and ∨ reduction. Using it, we can verify the following rule:

The result of ∧ reduction is 1 when all the elements in the list are 1, and only then.

To verify this rule, consider:

```
1∧1∧1∧1∧1∧1∧1∧1
 1
```

Evaluate any one of the ∧'s and you get a 1. Replace the 1∧1 you evaluated with that result and you will have the same expression as before, only with one less element and one less ∧. Continue evaluating and eventually you will get to 1.

Now put a 0 anywhere in the original list:

```
1∧1∧1∧1∧0∧1∧1∧1
 0
```

No matter what order you evaluate the individual ∧'s, eventually you must come to either 1∧0 or 0∧1. Either one evaluates to 0, and the 0 replaces either 1∧0 or 0∧1. Either way, there is still a 0 in the list, and eventually you will get to a result of 0.

As an exercise, use this line of argument to verify the following rule:

The result of ∨ reduction is 0 when all the elements in the list are 0, and only then.

Using these rules, we can see how two of the above tests can be rewritten. Namely, the rule for ∨ reduction says that the result of ∨ reduction is 1 if any elements in the list are 1. Consequently, a test of whether or not at least one of the individual relations holds is:

```
∨/(c>10)∨(b>10)
 1
```

Analogously, a test of whether or not all the individual relations hold is:

```
∧/(c>10)∨(b>10)
 0
```

8b. Scan Revisited

Now that we understand ∧ reduction and ∨ reduction, let's look at ∧

scan and ∨ scan. Starting with ∧ scan, consider the following example:

```
∧\1 1 1 0 1 1 0 1
 1 1 1 0 0 0 0 0
```

To understand this result, look at the individual ∧ reductions that make it up:

```
∧/1
 1
∧/1 1
 1
∧/1 1 1
 1
∧/1 1 1 0
 0
∧/1 1 1 0 1
 0
∧/1 1 1 0 1 1
 0
∧/1 1 1 0 1 1 0
 0
∧/1 1 1 0 1 1 0 1
 0
```

Reading the elements of the scan result from the left, all the corresponding elements of the ∧ scan are 1 until the first 0 in the argument is encountered. Once the first 0 is encountered in the argument list, all elements in the scan will be 0 from that point on, no matter what else occurs.

As an example of the use of ∧ scan, and based on the above interpretation, if x is a boolean list, then +/∧\x is the index of the first 0. For example:

```
x←1 1 1 0 1 1 0 1
+/∧\x
 3
x[0 1 2 3]
 1 1 1 0
```

Consequently, if x represents a list of relations, then +/∧\x is the index of the first relation (reading the list from the left) that fails to hold. For example:

```
      m←5 21 ¯7 50 8
      p←2 ¯10 20 17 12
p<m
 1 1 0 1 0
∧\p<m
 1 1 0 0 0
+/∧\p<m
 2
```

The description of ∨ scan is analogous:

Reading the elements of the list from the left, all the corresponding elements of the ∨ scan are 0 until the first 1 is encountered. Once the first 1 is encountered in the argument list, all elements in the scan will be 1 from that point on, no matter what else occurs.

In particular, if x is a boolean list the +/~∨\x is the index of the first 1 in the list. For example, using m and p above:

```
m+p
 7 11 13 67 20
20<m+p
 0 0 0 1 0
∨\20<m+p
 0 0 0 1 1
~∨\20<m+p
 1 1 1 0 0
+/~∨\0<m+p
 3
```

## 9. Elementary Algebraic and Transcendental Scalar Functions

A+ has the full set of elementary functions from standard calculus courses. The power function is denoted by the symbol *, and the logarithm by ⍟. For example:

```
2*0 1 2 3 4 5
 1 2 4 8 16 32
2⍟1 2 4 8 16 32
 0 1 2 3 4 5
```

a⍟b is the "base-a logarithm of b". The monadic function ⍟x is the natural logarithm function, and the monadic function *x is "e to the power x." For example:

```
*0 1 2 3
 1 2.718281828 7.389056099
⍟*0 1 2 3
 0 1 2 3
```

(Reading the last expression from left to right: "the natural logarithm [pause] of e to the powers 0 1 2 3".)

The circular functions are denoted by ○ and represent the elementary algebraic and trigonometric functions. The left argument is an integer between ¯7 and 7 that identifies a particular function, and the right argument is the argument of that function. For example, 0○x is (1-x*2)*0.5, i.e., "the square root of 1 minus x squared", and 1○x is sin x. A full list can be found in the A+ Reference Manual.

## 10. Miscellaneous Scalar Functions

### 10a. Absolute Value and Residue

The vertical bar (|) denotes the monadic scalar function called Absolute Value and the dyadic scalar function called Residue, or Remainder. For example:

```
|10 ¯2 0
 10 2 0
```

The absolute value of a positive number equals that number; the absolute value of 0 is 0; the absolute value of a negative number equals ¯1 times that number.

Turning to the dyadic function:

```
2|3 4 5 6 7
 1 0 1 0 1
```

The result of x|y is the remainder of y divided by x, or in the common mathematical term, y mod x. As the above example shows, 2|y is 1 if y is an odd integer and 0 if y is even. The result of 2|y is boolean. So V/ and +/ can be applied to the results, with the following interpretations:

```
V/2|3 4 5 6 7
 1
```

which means that there is at least one odd integer in the list, and:

```
+/2|3 4 5 6 7
 3
```

which means that there are three odd integers in the list.

10b. Sign

Sign, or signum, is the name for the monadic use of ×. For example:

```
×10 ¯2 0
 1 ¯1 0
```

That is, ×x for positive x is 1 and negative x is ¯1; ×0 is 0. Often, Sign is used in conjunction with Absolute Value. For example, consider fractional powers of negative numbers, which normally do not have results that are real numbers. They sometimes do, however, as in the cube root of ¯8, which is ¯2. The cube root function can be written in A+ as x*1÷3 or x*÷3, which is x to the one-third power. If this expression is applied to a negative number, a domain error results, because of approximations in the computer arithmetic.

```
¯8*÷3
 *: domain
*
```

The * you see on the last line of your A+ session indicates that execution is suspended because of an error. Enter the right pointing arrow to clear the suspension, as follows:

```
→
```

Using Sign and Absolute Value, it is easy to write a cube root function which for negative numbers like ¯8 produces the correct result: namely, (×x)×(|x)*÷3, which is Sign of x times the cube root of the Absolute Value of x. For example:

```
x←8 ¯8 0
(|x)*÷3
 2 2 0
×x
 1 ¯1 0
(×x)×(|x)*÷3
 2 ¯2 0
```

## 10c. Floor and Minimun

The monadic use of the symbol ⌊ is called Floor, and the dyadic use is called Minimum. Floor produces the integer part of its argument. For example:

```
⌊ 2 7.56 ¯12.34213
 2 7 ¯13
```

Floor is useful in separating numbers into their integer and fractional parts for subsequent analysis. It is also useful for rounding and truncation. For example, numbers can be truncated to two decimal digits by the function (⌊100×y)÷100:

```
y←89.1 91.20123 93.8975
100×y
 8910 9120.123 9389.75
⌊100×y
 8910 9120 9389
(⌊100×y)÷100
 89.1 91.2 93.89
```

Numbers can be rounded to the nearest integer by the function ⌊0.5+y. Using the above y:

```
0.5+y
 89.6 91.70123 94.3975
⌊0.5+y
 89 91 94
```

Minimum is dyadic and yields the lesser value of its left and right arguments. For example:

```
5⌊6
 5
5 12 ¯4 23 ⌊ 6 10 3 ¯8
 5 10 ¯4 ¯8
```

Minimum can be used with Reduction and Scan. ⌊ reduction yields the smallest element in a list, as in:

```
⌊/6 10 3 ¯8
 ¯8
```

and ⌊ scan yields the running minumum, in the same way as + scan

yields running sums:

```
⌊\6 10 3 ¯8
 6 6 3 ¯8
```

The last example suggests that we can now create test expressions such as one that determines the elements of a list that are less than or equal to all the elements that precede them:

```
z←6 10 3 8 3 5 2
⌊\z
 6 6 3 3 3 3 2
z=⌊\z
 1 0 1 0 1 0 1
```

The result indicates that for this value of z, the elements with indices 0, 2, 4, and 6 pass the test.

## 10d. Ceiling and Maximum

The monadic use of the symbol ⌈ is called Ceiling, and the dyadic use is called Maximum. Ceiling and Maximum are companions of Floor and Minimum. For example:

```
⌈ 2 7.56 ¯12.34213
 2 8 ¯12
5 12 ¯4 23 ⌈ 6 10 3 ¯8
 6 12 3 23
```

## 10e. Roll

The monadic use of the symbol ? denotes a scalar function called Roll that produces random numbers. If the argument is a single number it must be a positive integer, and the result will be a nonnegative random integer that is less than the argument. For example:

```
?100
 53
```

Actually, although care has been taken so the displayed value is the same as you will see when you execute this line, it is possible that they are different. If we both started this tutorial in a new A+ session and this is the first evaluation of either monadic or dyadic ?, then the results should be the same.

Repeat the above evaluation and you are likely to get a different answer:

```
?100
 14
```

Since Roll is a scalar function, random sequences can be generated by applying roll to a list:

```
?10 100 1000 100 10
 7 2 631 85 0
```

Roll is used in the "Simple Arrays" tutorial to generate test data. The dyadic use of ? is called Deal. Deal, which is not a scalar function, yields sequences of distinct random numbers. For example:

```
  20?100
 11 89 72 81 15 44 62 42 43 21 20 63 90 36 31 17 56 79 59 53
```

## 11. Tables

Two summary tables appear below. The first summarizes the scalar functions according to their symbols, names, and classification. The second summarizes the operators.

More on scalar functions can be found in the "Simple Arrays" tutorial.

Table 1: Scalar Functions

Class: A = Arithmetic, E = Elementary, M = Miscellaneous, R = Relational, L = Logical

| Class | Symbol | Dyadic Name | Monadic Name | Class | Symbol | Dyadic Name | Monadic Name |
|---|---|---|---|---|---|---|---|
| A | + | Add | Identity | R | = | Equal to | |
| A | - | Subtract | Negate | R | ≠ | Not Equal to | |
| A | × | Multiply | Sign | R | < | Less than | |
| A | ÷ | Divide | Reciprocal | R | ≤ | Less than or Equal to | |
| E | ⋆ | Power | Exponential | R | > | Greater than | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E | ⊛ | Log | Natural Log | R | ≥ | Greater than or Equal to | |
| E | ○ | Circle fns | Pi tImes | L | ∧ | And | |
| M | ⌈ | Max | Ceiling | L | ∨ | Or | |
| M | ⌊ | Min | Floor | L | ~ | | Not |
| M | \| | Residue | Absolute value | | | | |
| M | ? | | Roll | | | | |
| M | ∪ | Combine Symbols | | | | | |

Table 2: Operators on Scalar Functions

| Operator | Forms |
|---|---|
| Reduction | +/ ×/ ∧/ ∨/ ⌊/ ⌈/ |
| Scan | +\ ×\ ∧\ ∨\ ⌊\ ⌈\ |
| Outer Product | ∘.+ ∘.- ∘.× ∘.÷ ∘.\| ∘.⌊ ∘.⌈ |
| | ∘.= ∘.≠ ∘.< ∘.≤ ∘.> ∘.≥ |
| Inner Product | +.× ⌊.+ ⌈.+ |