Assignment

Summary



## 1. Introduction

The purpose of this tutorial is to familiarize the reader with the A+ primitive function called assignment, or specification. This primitive is the one to use for initializing and modifying the values of variables. Ordinary assignment is of the form a←b, where the term ordinary refers to the simple form of the construct to the left of the arrow. Selective assignment allows for more complicated constructs on the left, whereby subarrays of the values of variables are replaced.

It is possible to create functions with ordinary assignment when there is a function expression to the right of the assignment arrow. Function assignment is not a topic for this tutorial, however; see the

A+ Reference Manual. In addition, ordinary assignment has a special side effect when used in the left argument of a dyadic do statement.

This tutorial is made up of textual descriptions and A+ examples. 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.

To bring individual expressions from the tutorial into the A+ session, place the cursor on the expression and press F2; for function definitions place the cursor anywhere in the definition and press F3. It is assumed that the expressions and functions are brought into the A+ session when you first encounter them, unless there are explicit directions to the contrary.

If you need more help on running emacs and A+, see the Getting Started tutorial.

## 2. Ordinary Assignment

Ordinary assignment associates a value, i.e. an array, with a variable name. For example:

```
a←⍳2 3
and
c.b←'abcdef'
```

In the latter case the assignment is to the variable b in the context named c:

```
$cx c       ⍝ Enter the context named c
$vars
 b ⍝ b is in the context c
  $cx . ⍝ Return to the root context
  $vars
 a ⍝ a is in the root context
```

Ordinary assignment can also be expressed in the form (a)←b. When used inside a function definition, any appearance of a←b means that a will be a local variable, while the form (a)←b can be used to assign a value to the global variable a.

## 3. Strand Assignment

Several ordinary assignments can be incorporated into one by way of strand assignment. For example:

```
(a;c.b)←('ABC';10+⍳3 4)
```

Strand assignment is important when working with dependencies. See the discussion of commit and cancel for sets of dependencies in the section on Cyclic Dependencies in the A to A+ document.

## 4. Value and Value in Context Assignment

Symbols are convenient form for managing names in A+ applications, and there is a primitive function that evaluates symbols holding the names of global variables.

### 4a. Value

Value is the monadic primitive function denoted by % that takes a symbol holding a global variable name as its argument and produces the value of the global variable. For example, your A+ session should now be in the root context, and we can then continue the above example as follows:

```
%`a
ABC
%`c.b
 10 11 12 13
 14 15 16 17
 18 19 20 21
```

### 4b. Value in Context

Value in Context is the dyadic primitive denoted by %, which permits a symbol on the left representing the context of the variable on the right.

```
`%`a   ⍬ The empty symbol ` denotes the root context
ABC
`c%`b
 10 11 12 13
 14 15 16 17
 18 19 20 21
```

4c. Value and Value in Context Assignment

Both Value and Value in Context can be used on the left side of assignment.

```
(%‘a)←⍎a
a
CBA
(‘c%‘b)←10×%‘c.b
c.b
 100 110 120 130
 140 150 160 170
 180 190 200 210
```

Both Value and Value in Context assignment are important when working with callback functions. See the A to A+ document for discussions of callback functions.

5. Bracket Index Selective Assignment

Just as bracket indexing can be used to select subarrays, it can also be used on the left side of the assignment arrow to specify subarrays. Here are a series of examples applying bracket index selective assignment to a variable x. The examples have been chosen to make it fairly easy to look at x after each assignment to verify that you understand the change that took place.

```
  x←⍳3 5 6
x[0;0;0]←50 ⍝ Specify the element at coordinates (0,0,0)
x[0;1]←100+⍳6 ⍝ Specify the rank one cell at coordinates (0,1)
x[0;1;]←200+⍳6 ⍝ Specify the same rank one cell in a different way
x[2]←1000+x[2] ⍝ Specify the second item
  x[2;;]←1000+x[2] ⍝ Specify the same item in a different way
```

Any bracket index expression that can be used to select a subarray can be used to replace that subarray. The replacement array must have the same shape as the indexed subarray, or it must have one element. In the latter case, the one element in the replacement array replaces every element of the indexed subarray.

```
x←⍳3 5 6 ⍝ Restart
⍴x[1 2;;2+⍳2 2]
 2 5 2 2
x[1 2;;2+⍳2 2]←100+x[1 2;;2+⍳2 2]
```

```
x[1 2;;2+⍳2 2]←1000
```

5a. Repeated Indices

What about repeated elements in the bracket index expression? Let's look at a simple example:

```
y←⍳10
y[2 2 2]← 30 40 50
```

The question is: element y[2] is associated with three distinct elements for replacement: 30, 40, and 50. What is the value of y[2] after the above assignment? The answer is: the one with the highest index in the ravel of the right argument.

```
y[2]
  50
```

Another example:

```
y[2 2⍴3 3 3 3 ]←2 2⍴ 100 200 300 400
  y[3]
 400
```

Finally, note that Value, Value in Context, and Ravel can be used with bracket indexed selective assignment.

```
z←⍳5 6
(,%`z)[4 5 6]←100 200 300
```

5b. Replace All: a[]←b

This is a special form of bracket index selective assignment for replacing all the elements in the variable named on the left of the assignment arrow. Either the shape of the array on the right must be identical to the shape of the value of the variable on the left, or the array on the right must have one element. In the latter case, every element of the array on the left is replaced with the single element on the right. For example:

```
a←⍳3 4
a[]←?3 4⍴100
or
a[]←5
```

```
ρa
 3 4
```

The differences between a←b and a[]←b are:

a←b copies b into a new (replacement) copy of a, but a[]←b copies b into the existing a;

a←b enforces no (prior) compatibility restraints on a and b, but a[]←b is valid only if a and b have compatible types, and either identical shapes or b has one element.

However, a[]←b is the most efficient way to replace an entire mapped file.

5c. Append: a[,]←b

This is another special form of bracket index selective assignment for appending items onto the end of an array. For example:

```
a←⍳2 5
a[,]←10 20 30 40 50
a
  0  1  2  3  4
  5  6  7  8  9
 10 20 30 40 50
```

Execution of a[,]←b can be more efficient then a←a,b because if there is enough unused space in the storage area allocated to a, then b can simply be copied into the area at the end of a. This form of specification is the most efficient way to update a mapped file.

6. Choose-Pick Selective Assignment

The basic rule for bracket index selective assignment applies to Choose-Pick selective assignment as well: if the arrays i and p are such that i#p⊃x selects a subarray from x, then:

```
⍝ (i#p⊃x)←a
```

will replace that subarray. Since # denotes the primitive Choose function and ⊃ denotes the primitive Pick function, this is called Choose-Pick selective assignment.

Value, Value in Context, and Ravel can be used with Choose-Pick assignment, as in:

```
⍎ (i#,p⊃‘c%‘x)←a
```

Examples:

```
x←(1 2 3;⍳5 6;'abced')  A three element boxed vector
1 2#2⊃x bc
(1 2#2⊃x)←'XY'
(1 2;3 4 5)#1⊃x
  9 10 1
 15 16 17
((1 2;3 4 5)#1⊃x)←100+?2 3⍴10
```

Examine x to see the changes. Another example:

```
(1 6 17 25#,1⊃x)←1000 2000 3000 4000
```

Examine x now and you will see that the last selective assignment did not replace a rectangle subarray of 1⊃x, but instead modified elements scattered throughout x. This type of scattered assignment is most efficiently done to the ravel of the target array; if the array is not raveled, then Choose can only address these elements one at a time, as in:

```
      ((0;1);(1;0);(2;5);(4;1))#¨<1⊃x
<  1000
<  2000
<  3000
<  4000
```

Note the <1⊃x is just 1#x, but the former construct was used to make the comparison with the earlier examples easier.

It is left to you to make examples using Value and Value in Context.

## 7. Primitive Functions in Selective Assignment Expressions

Some A+ primitive functions can be used in expressions on the left of the assignment arrow. They are: Take, Drop, Replicate, Expand, Ravel and Item Ravel, Reshape, Rotate and Reverse, and Transpose (monadic and dyadic.) Defined functions can also be used, but in the same ways as these primitives, and therefore are not discussed here. See the A+ Reference Manual.

For example:

```
a←4 5⍴'abcdefghijklmnopqrst'
(2↑a)←2 5⍴'ABCDEFGHIJ'
  a
ABCDE
FGHIJ
klmno
pqrst
(5↑a)←5 5⍴'abcdefghijKLMNOPQRSTUVWXY'
a
Ybcde
fghij
KLMNO
PQRST
```

The last result gives us a clue to the definition of this form of selective assignment. Let

```
(f a)←b
```

represent the selective assignment for the monadic primitive functions listed above, where a represents a variable name. Then the definition of the assignment is as follows: first evaluate

```
  i←f ⍳⍴a
```

to get an array i, and then do the bracket index selective assignment

```
(,a)[i]←b
```

Why does this work? The reasons are these:

    the elements of ⍳⍴a are exactly all indices of ,a;

    ∧/(,f ⍳⍴a)∈,⍳⍴a has the value 1.

Make sure you understand these reasons.

Applying the definition to the last example above, evaluate 5↑⍳⍴a to get the index array i:

```
i←5↑⍳⍴a
i
  0  1  2  3  4
```

```
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
 0  0  0  0  0
```

Then evaluate

```
(,a)[i]←5 5⍴'abcdefghijKLMNOPQRSTUVWXY'
```

to get the result in the example. The fact that a[0;0] is 'Y' is explained by the rule for repeated elements in i (see Repeated Indices.)

The definition for the dyadic primitives is similar. In this case the right argument a, which must be a name, is the target of the assignment and the left argument x is any conformable array for which x f a is valid. Then

```
(x f a)←b
```

is evaluated by

```
i←x f ⍳⍴a

(,a)[i]←b
```

Note that the primitive functions listed above all have the property that the elements of the intermediate index array i are always a subset of ⍳⍴a, which means that (,a)[i]←b can never fail because of an index error (see reasons 1) and 2) above.)

Value, Value in Context, Pick and Ravel can all be used with this form of selective assignment. See the A+ Reference Manual.