The Syntax of A+

Summary

1. Introduction

The purpose of this tutorial is to describe the syntax of A+ through a series of examples, rather than in a formal way. Some commonly understood terms are used without being formally defined. In particular, the phrase A+ expression, or simply expression, is taken to have the same general meaning it does in mathematics, namely, a well-formed sentence that produces a value. A brief discussion of well-formed expressions is presented at the end, after all the rules for the components of expressions have been presented.

Not all aspects of A+ syntax are discussed here; see the chapter on syntax in the A+ Reference Manual, and the Assignment tutorial.

Although this tutorial is primarily concerned with syntax, examples require some knowledge of their meaning. Each example will be fully explained, but comprehensive treatments of topics other than syntax are left to the other language tutorials.

The 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 Getting Started. If you want to try your hand at writing your own A+ expressions, see the keyboard layout diagrams in Appendix B of the A+ Language Manual.

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

2. Names and Symbols

One of the most basic things to know is how things are named. There are no exercises in this section, just information you will need later.

2a. Primitive Functions

A+ uses a mathematical symbol set to denote the functions that are native to the language, which are called primitive functions. This symbol set, which is the APL character set, consists of common mathematical symbols such as + and ×, commonly used punctuation symbols, and specialized symbols such as ↑ and ↓. In some cases it takes more than one symbol to represent a primitive function, as in +/, but the meaning can be deduced from the individual symbols.

2b. User Names

User names fall into two categories, unqualified and qualified. A valid, unqualified name is made up of alphanumeric (alphabetic or numeric) characters and underbars (_). The first character must be alphabetic. For example, a, a1c, and a_1c are valid unqualified names, but 3xy and _xy are not.

A valid qualified user name is either an unqualified user name preceded by a dot (.), or a pair of unqualified user names separated by a dot. In either case there are no intervening blanks. For example, .xw1 and w_2.r2_a are valid qualified user names.

2c. System Names

System names are unqualified names preceded by an underbar, with no intervening spaces. For example, _argv is a valid system name. The use of system names is reserved by A+.

2d. System Commands

System commands begin with a dollar sign, followed immediately by an unqualified name, which is the name of the command. The name is followed by a space, and then possibly by a sequence of characters whose meaning is specific to the command. For example $fns is a valid system command.

2e. Comments

Comments can appear on a line by themselves or to the right of any expression. They are indicated by the ⍝ symbol, and everything to the right of this symbol is the comment. For example:

```
2×3 ⍝ This is the A+ notation for multiplication.
```

3. Infix Notation and Ambivalence

A+ is a mathematical notation, and as such uses infix notation for functions with two arguments. That is, the symbol or user name for a function with two arguments appears between them. For example, a+b denotes addition, a-b subtraction, a×b multiplication, and a÷b division. In mathematics, the symbol - can also be used with one argument, as in -b, in which case it denotes negation. This is true in A+ as well. Because the symbol denotes two functions, one with one argument and the other with two, it is called ambivalent.

A+ has extended the idea of ambivalence to most of its primitive functions. For example, just as -b denotes the negative of b, ÷b denotes the reciprocal of b.

User defined functions cannot be ambivalent.

Functions with one argument are called monadic, and functions with two arguments are called dyadic. For a primitive function symbol, one often refers to its monadic use or dyadic use.

Ex 1. Execute each of the following using F2. After each one is executed, you will see the result displayed immediately below.

```
5÷2
```

```
÷2
```

A more interesting example, perhaps, is the primitive function denoted by the down arrow ↓(meta-u on the keyboard). The dyadic form is called Drop because it has the effect of dropping a specified number of elements from a list. For example, if x is the name of a variable containing the list of five characters a, b, c, d, and e, then 2↓x drops the first two characters from the list, leaving a list of the three characters c, d, and e. The monadic form of ↓ is called Print because its effect is to display its argument in the A+ session log. For example, execute the following:

```
5÷(↓2)
```

and you will see 2 displayed, followed by the result of the expression. The print primitive, like all primitives, produces a result, and that result is used in further execution. Unlike most primitives, it also has a side effect, which is the display of its argument in the session log.

Ex 2. What do you think the result of ↓x is? Describe it in terms of x.

4. Syntactic Classes

4a. Numeric Constants

Individual numbers can be expressed in the usual integer, decimal, and exponential formats, with one exception: negative number constants begin with a "high minus" sign (¯) instead of the more conventional minus sign (-). Negative exponents in the exponential format are denoted by the conventional minus sign.

It is also possible to express a list of numbers as a constant, simply by separating the individual numbers by one or more blank spaces. For example:

```
1.23 ¯7 45 3e-5
```

is a numeric constant with four numbers: 1.23, negative 7, 45, and 0.00003.

Ex 2. Most likely you are familiar with numeric formats, and by the end of this tutorial you should be experimenting with expressions of your own creation, so we will use numeric constants to illustrate how to deal with ill-formed expressions.

The high minus sign is not used for exponents. Execute the following to see a parse error message:

```
1e¯2
```

Ex 3. Constants can have more than one element, as illustrated above. As a single number, 1.2.3 is ill-formed, but A+ parses this sequence as if it were a list of numbers. Execute the following and explain what you see:

```
1.2.3
```

Ex 4. Constants can be put inside parentheses, which does not effect their value, but gives us a way to illustrate syntax errors. Execute the following:

```
2.109)
```

You will see a syntax error message saying that the right parenthesis has no matching left. Now execute

```
(2.109
```

You will now see a *. The display of a * by the A+ in circumstances like these indicates suspended execution. The reason that this expression results in suspended execution instead of a syntax error is that it is viewed by the A+ process as incomplete. More characters could have been appended on its right side to form a complete expression, which is not true of the first expression, 2.109). Select the A+ buffer, and the keyboard cursor should then be positioned to the right of the *. Enter the closing right parenthesis and press the Return key. You will see 2.109 displayed, just as if you had entered the syntactically correct expression (2.109) all on one line.

Select the tutorial buffer to continue.

The A+ language processor accepts expressions that occupy more than one line. However, expressions cannot be broken in the middle of

names, or numeric constants, or primitive functions that require more than one character, and their must be a reason for A+ to expect a continuation, such as open punctuation.

Ex 6. This exercise is a variation of the last one. Execute the expression:

```
(2.109
```

Once again you will see a *. Select the A+ buffer and enter (instead of). Press the Return key. You will now see two *'s. There are two points to be made here. First, the number of *'s indicates the level of suspension. It now takes two actions to clear the suspended execution, e.g. two closing parentheses. Second, suppose entering the second (was a mistake, and you simply want to clean things up and start over. To do this you should enter a right pointing arrow (meta-] on the keyboard) next to the two *'s, and press the Return key. Do that, and then select the tutorial buffer to continue.

4b. Character Constants

A character constant is expressed as a list of characters surrounded by a pair of single quote marks or a pair of double quote marks. In order to include the surrounding quote mark in the list of characters, it must be doubled. For example, both 'abc''d' and "abc'd" are constant expressions for the list of characters abc'd.

Ex 5. Execute each of the following to see how ' and " are handled:

```
'Aed"ss'
```

```
"Aed'ss"
```

The following will cause errors:

```
'Aed'ss'
```

```
'Aed' 'ss'
```

Explain the error reports. Clear any suspended executions, and return to the tutorial buffer.

Ex 6. What do you think happens if you break an A+ expression in the middle of a character constant?. Execute the expression:

```
’abcd
```

and you will see the suspension indicator. To the right of it enter:

```
* 2345’
```

The result will now be displayed. Explain what you see. For that purpose, note that the symbol # applied monadically to a list of characters yields the number of characters in the list. For example:

```
#’sdTvw’
 5
```

Repeat the above example using # as follows:

```
  #’abcd
* 2345’
 9
```

Explain the result.

4c. Symbol Constants

A symbol is a backquote (‘) followed immediately by a character string made up of the alphabetic characters, underscores (_), and dots (.). A symbol constant can be thought of as a character-based counterpart to numeric constants. Just as 1 2.34 12e3 is a list of three numbers, ‘a.s ‘12 ‘w_3 is a list of three symbols.

4d. The Null

The Null is a special constant formed as follows: (). It is neither numeric nor character, but has a special type reserved for it alone.

4e. Variables

Variables are named data objects. They receive their values through assignment, or specification, which is denoted by the left-pointing arrow (←). For example, the expression

```
abc←1 2 3
```

assigns the three-element list consisting of 1, 2, and 3 to the variable named abc. Any valid user name can serve as a variable name.

For more on assignment, see the Assignment tutorial.

4f. Functions

Functions take zero or more arguments and return results. A sequence of characters that constitutes a valid reference to a function will be called a function call expression. That is, a function call expression includes a function symbol or name together with all its arguments and all necessary punctuation. In general, the arguments of a function are data objects, which may appear in function call expressions as variable names, constants, or expressions that require evaluation. In addition, for the various forms of function call expressions using braces, arguments can also be functions.

A function with no parameters - which must be a user defined function - is said to be niladic. The valid function call expression for a niladic function f is f{}.

Functions with one argument can be either primitive or user defined. The valid function call expressions for a function f with one argument a are f a and f{a}. In the form f a, the space is required if, when it is omitted, the result would be a valid name, as plus 2.3.

Functions with two arguments can also be either primitive or user defined. The valid function call expressions for a function g with two arguments a and b are a g b and g{a;b}. a is called the left argument and b is called the right argument. The rule for required spaces in the dyadic form a g b is the same as for the monadic form f a.

Functions with more than two arguments must be user defined. The valid function call expression for a function of more than two arguments a, b, ..., c is f{a;b;⍝;c}.

For function call expressions that use braces and contain at least two arguments, any of the positions between neighboring semicolons, or between the left brace and the first semicolon, or between the last semicolon and the right brace, can be left blank. For example, each of the following is a valid function call expression: f{a;}, f{;b}, f{;a;b}, f{;;b}, etc. However, if f is monadic then f{} is not valid,

because f{} is a niladic function call expression. When an argument position is legitimately left blank, A+ assumes that the argument is the Null.

The number of arguments of a function is called its valence. The valence of a user defined function is fixed by the form of its definition.

Ex 7. Use F2 to define the following dyadic function:

```
a f b:a-b
```

and then evaluate the following function call expressions:

```
2 f 5
```

```
f{2;5}
```

(Function definitions are discussed in Defined Functions.) Explain the meaning of

```
-{2;5}
```

and then execute it for verification.

Ex 8. Define the following function:

```
g{a;b;c}:(a;b;c)
```

As will be explained later, the result of this function is a data aggregate with three elements, which are the arguments to the function. For example, execute:

```
g{1;2;3}
```

and you will see displayed three lines, with < 1, < 2, and < 3. The symbol < indicates that the data being displayed is part of an aggregate. Now execute:

```
g{;2;3}
g{1;;3}
```

and you will see that wherever an argument is omitted, the corresponding output line is < followed by blanks. This indicates that the omitted arguments are taken to be the Null (however, the same display line could represent other things as well, such as a blank

list of characters.)

4g. Operators and Derived Functions

There are two formal, primitive operators in A+, known as Rank and Each. By a formal operator we mean an operator in the mathematical sense, i.e. a function that takes a function as an operand, or produces a function as a result, or both. The resulting function is called a derived function.

The Each operator is denoted by the dieresis, ¨. For a given function f, the function derived from the Each operator is denoted by f¨. The function f can be either monadic or dyadic, in which case so is f¨.

The Rank operator is denoted by the at symbol, @. Unlike the Each operator, the Rank operator has both a function argument and a data argument. For a given function f and data value a, the function derived from the Rank operator is denoted by f@a. f can be either monadic or dyadic, in which case so is f@a.

Ex 9. The Rank and Each operators modify their function argument to produce some variant of that function. For example, use F2 to execute:

```
(2;3)+4
```

You should see the error message +: type, which in this case means that + does not apply to data aggregates. Following the message is a line with a *, indicating suspended execution. Clear the suspension and return to the tutorial. Now use F2 to execute:

```
(2;3)+¨4
```

Explain the result you see. What do you think the following expressions produce? Evaluate them to confirm your guesses.

```
(2;3)+¨4 5
(2;3)+¨(4;5)
```

Reduction, Scan, Outer Product, and Inner Product are not operators, strictly speaking: they do not accept all functions as operands. The ones they do accept are shown in Table 2-2. Because these character sequences look so much like derived functions, ohowever, we will use the term operator to include these four as well as the primitive Each and Rank operators and user defined operators.

Ex 10. Many of the symbols in should be familiar, but some may not be. For example, ⌊ (meta-d on the keyboard) and ⌈ (meta-s) denote the Minimum and Maximum functions, respectively, when used dyadically. Execute the following expressions:

```
3⌊5
3⌈5
⌊/1 2 3 4 5
⌈/1 2 3 4 5
+/1 2 3 4 5
```

Explain how the functions ⌈/, ⌊/, and +/ are variants of the functions ⌈, ⌊, and +. Feel free to experiment with other arguments. Remember, if you make error and execution is suspended, enter the right arrow (meta-]) to get out of it.

5. Defined Functions

A function definition consists of a function header, followed by a colon, followed by either an expression, or an expression block, which is a series of expressions separated by semicolons and enclosed in braces and represents a sequence of statements to be executed.

Function headers take the same forms as function call expressions (see Functions above), except that no argument may be omitted. A function header has the monadic form, dyadic form, or general form. The monadic form is the function name followed by the argument name, with the two names separated by at least one space. For example, if the function name is correlate then

```
correlate a:{...}
```

is a function definition with the monadic form of the header.

The dyadic form of function header is the function name with one argument name on each side, with the names separated by at least one blank. For example:

```
a correlate b:{...}
```

is a function definition with the dyadic form of the header.

The third form of function header is the general form, which is the function followed by a left brace, followed by a list of argument

names separated by semicolons, and terminated with a left brace. For example:

correlate{a;b;c}:{...}

is a function definition with the general form of the header. In this example the function has three arguments.

A function with one argument can be defined with either the monadic form of function header, or the general form, and analogously, functions with two arguments can be defined with either the dyadic form or general for. Regardless of which way they are defined, they can be called either way.

Ex 8 provides an example of a defined function. The result of that function is the value of the (a;b;c).

6. Dependencies

A dependency definition consists of a name (the name of the dependency), followed by a colon, followed by either an A+ expression, or an expression block.

7. Bracket Indexing

A+ data objects are arrays, and bracket indexing is a way to select subarrays. Bracket indexing uses special syntax, whose form is

x[a;b;⍳;c]

where x represents a variable name and a, b,⍳,c denote expressions. The space between the left bracket and the first semicolon, between successive semicolons, and between the last semicolon and the right bracket, can be empty.

Ex 16. This exercise takes us into the subject matter of the other language tutorials, but it is interesting to see what it means to leave the spaces in the bracket index expression empty. Execute the following:

⍳3 4

and you will see a matrix with three rows and four columns, populated by the numbers 0 through 11. Execute each of the following and

explain what you see:

```
(⍳3 4)[0;0]
(⍳3 4)[2;3]
(⍳3 4)[1;1 3]
(⍳3 4)[1;3 1]
(⍳3 4)[1;]
(⍳3 4)[;2]
(⍳3 4)[;]
```

8. Strands

Aggregate data objects can be formed by separating the individual data objects with semicolons and surrounding the collection of data objects and semicolons with a pair of parentheses. For example:

```
(a;b;...;c)
```

where a,b,...,c denote expressions. Any of these expressions can be function expressions.

See Ex 8.

9. Precedence Rules

The precedence rules in A+ are simple:

all functions have equal precedence, whether primitive, defined, or derived

    all operators have equal precedence

    operators have higher precedence than functions

the formation of numeric constants has higher precedence than operators.

Ex 11. Execute the following:

```
1 2+3 4
```

The result indicates that the constant with the two numbers 1 and 2, and the constant with the two numbers 3 and 4, are formed before + is applied. Do you see how this is related to the above rules?

10. Right-to-Left Order of Execution

The way to read A+ expressions is from left to right, like English. For the most part we also read mathematical notation from left to right, although not strictly because the notation is two-dimensional. To illustrate reading A+ expressions from left to right, consider the following examples.

```
a+b+c
```

Read as: "a plus the result of b plus c."

```
x-÷y
```

Read as: "x minus the reciprocal of y."

As you can see, reading from left to right in the suggested style implies that execution takes place right to left. In the first example, to say "a plus the result of b plus c" means that b+c must be formed first, and then added to a. And in the second example, to say "x minus the reciprocal of y" means that ÷y must be formed before it is subtracted from x.

Of course, reading from left to right is not necessarily associated with execution from right to left. For example, the expression a÷b+c is read left to right in conventional mathematical notation as well as A+, but the order of evaluation is different in the two; in mathematics a divided by b is formed and added to c, while in A+, a is divided by b+c. The order of execution is controlled by the relative precedence of the functions, or operations. In mathematics, divide has higher precedence than plus, which means that in a÷b+c, divide is evaluated before plus.

Another way to say that A+ expressions execute from right to left is that A+ has long right scope and short left scope. For example, consider:

```
a+b-c÷e×f
```

The arguments of the minus function are b on the left (short scope) and c÷e×f on the right (long scope.) The left argument is found by starting at the - symbol and moving to the left until the smallest possible complete subexpresson is found. In this example it is simply the name b. If the first non-blank character to the left of the

symbol had been a right parenthesis, then the left argument would have included everything to the left of the right parenthesis, up to the matching left parenthesis. For example, the left argument of minus in a+(x÷b)-c÷e×f is x÷b.

The right argument is found by starting at the - symbol and moving to the right, all the way to the end of the expression, or until a semicolon is encountered, or until a right parenthesis, brace, or bracket is encountered whose matching left partner is to the left of the symbol. In the above example the right argument of minus is everything to the right. If the case of a+b-(c÷e)×f, the right argument is also everything to the right. However, for a+(b-c÷e)×f, the right argument is c÷e.

11. Control Statements

11a. Case Statement

The form of a case statement is the word case, followed by an expression in parentheses, followed by one of two special expression sequences. The placement of semicolons must be as illustrated below. The point of the specification in the examples is that A+ control statements are actually compound expressions with results.

```
x←case (a) {0;"The case is 0";
1;"The case is 1";
"The default case"
}

x←case (a) {0;"The case is 0";
1;"The case is 1";
  }
```

These expression blocks are of the form

```
{case-expression0; value-expression0;
 case-expression1; value-expression1;
  .
.
.
}
```

In both of the above instances, the case statement is evaluated by first evaluating the expression in parentheses. The value of that expression is compared to the value of case-expression0. If they

match, value-expression0 is evaluated and its value is the result of the case statement. If they do not match, the value of the expression in parentheses is compared to the value of case-expression1. If they match, value-expression1 is evaluated and its value is the result of the case statement. This pattern continues until the case-expression, value-expression pairs are exhausted. At that point the case statement either has one remaining expression (the first example above) or none. If there is one, it is evaluated and its value is the result of the case statement. If there is none, the result of the case statement is the Null.

11b. Do Statement

The monadic form of the do statement is the word do, followed by an expression or expression block.

The dyadic form is like the monadic form, except that a valid left argument expression appears to the left of the word do. There are two special forms recognized for the left argument. For example, evaluate each of the following:

```
n←10
x←n do ↓n
n
```

The specification of n is simply to get the example going. The point is that when the do statement is evaluated, n already has a value. The do statement prints the value of n each time it is evaluated. You might have expected to see a series of 10's, but you saw 0 through 9. The rule is that when the left argument is simply a variable name with an integer value, say k, that variable is successively given the values 0, 1,⍳,k-1 for the successive evaluations of the expression on the right. Finally, evaluating the last statement in the above sequence shows that n once again has its value (10) from before evaluation of the do statement.

Basically the same behavior occurs when the left side of the do statement is a simple specification. For example:

```
x←(n←10) do ↓n
n
```

No other form of the left argument has this effect. For example:

```
n←20
```

x←(n-15) do ↓n

11c. If Statement

The form of an if statement is the word if, followed by an expression in parentheses, followed by another expression or an expression block.

11d. If-Else Statement

The form of an if-else statement is the word if, followed by an expression in parentheses1, followed by another expression or expression block, followed by the word else, followed by another expression or an expression block.

11e. While Statement

The form of a while statement is the word while, followed by an expression in parentheses1, followed by another expression or an expression block.

12. Execution Stack References

Execution stack references are &, &0, &1, etc. The symbol & can be used in a function definition to refer to that function. For example, a factorial function can be recursively defined in either of the two following ways:

fact{n}: if (n>0) n×fact{n-1} else 1

fact{n}: if (n>0) n×&{n-1} else 1

When execution is suspended the objects on the execution stack can be referenced by &0 (top of stack), &1, etc. See the Dealing with Errors tutorial.

13. Well-Formed Expressions

Basically, a well-formed expression is one that takes one of the forms described above, and in which all of the constituents are well-formed. The potential for complicated expressions is due to the fact that every one of these basic forms produces a result and can therefore be

used as a constituent in other forms. In this regard A+ is very much like mathematical notation.

The concept of the principal subexpression of an expression is useful for analysis. As execution of an expression proceeds in the manner described in Right-to-left Order of Execution, one can imagine that parts of the expression are executed and replaced with their results, and then some remaining parts are executed using these results, and are replaced with their results, and so on. Ultimately the execution comes to the last expression to be executed, which is called the principal subexpression. Once executed, its value is the value of the expression. If the principal subexpression is a function call expression or operator call expression, the function or derived function is called the principal function.

For example, the principal subexpression of (a+b÷c-d)*10×n is x*y, where x is the result of a+b÷c-d and y is the result of 10×n. The power function * is the principal function. As a second example, the principal expression of (x+y;x-y) is (w;z), where w is the result x+y and z is the result of x-y. In this case we do not refer to a principal function.

Knowing the principal subexpression often reveals the thrust of a complicated expression. Mathematical notation gives visual clues that usually point the reader directly to the principal subexpression. There are clues in A+ as well, but they are based largely on experience.

Ex 12. In each row of Table 3-1, an expression is given together with its principal function or expression. Make sure you understand each case.

Table 3-1: Well-Formed Expressions

Expression	Principal Function or Principal Expression
a+b-c×d	+
(a+b)-c×d	-
f÷×w	÷
(x-y)[a*2]	w[z]
(⊃+/¨w-a)/z	/
⊃+/¨w-a	⊃
+/¨w-a	+/¨

```
| (a+.×b)↓a∘.+b |↓                                                    |
| a∘.+b         |∘.+                                                  |
|f{a;g×a;x-y*2} |f{a;t;s}                                             |
-----------------------------------------------------------------------
```