Chapter 4: The Syntax and Semantics of A+

The main purpose of this chapter is to describe the syntax of A+, but through a series of examples, rather than in a formal way. Consequently 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. In addition, some discussion of semantics has been included, but only where it seemed reasonable in order to complete a description. A brief discussion of well-formed expressions is presented at the end of this section, after all the rules for the components of expressions have been presented.

Names and Symbols

Primitive Function Symbols

A+ uses a mathematical symbol set to denote the functions that are native to the language, which are called *primitive functions*. This symbol set, part of 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. The symbols are listed in Table B-1, page 225.

Two of the symbols can be used alone, viz., `ß` and . If the execution of a function or operator has been suspended, they mean resume execution (with increased workspace size if necessary) and abandon execution, respectively; in the absence of a suspension, they are ignored. Instead of , a dollar sign ($) can be used. Inside a function definition, an expression can consist of the symbol `ß` alone, but it will be ignored, and the parser rejects alone as a token error.

User Names

User names fall into two categories, unqualified and qualified. An *unqualified name* is made up of alphanumeric (alphabetic and numeric) characters and underbars (_). The first character must be alphabetic. For example, `a`, `a1c`, and `a_1c` are unqualified names, but `3xy` and `_xy` are not. (Although underbar is currently permitted as the first character in user names, this manual has been written as if it were not, and you should consider this form reserved for system names and avoid it.) The identifying words in control statements (case, do, else, if, while) are reserved by A+ for that use; they cannot appear as user names, even in qualified names.

A *qualified 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 qualified user names. An unqualified name preceding the dot in a qualified name is the name of a *context*. If there is a dot but no preceding name, the context is the *root* context.

System Names

System function names are unqualified names preceded by an underbar, with no intervening spaces, `_argv` for instance. The use of system function names is reserved by A+.

The name of an object traditionally (and therefore in A+) called a system variable is an unqualified name preceded by a backquote, with no intervening spaces. For example, `‘rl` is the name of the system variable called Random Link. These objects cannot be dealt with directly in A+, but only through certain system and primitive functions and system commands, to which they act as parameters. As indicated in “Symbols and Symbol Constants”, page 27, they look just like symbols (and may be considered such). They are not, however, the symbol forms of names: A+ will not recognize `rl`, for instance, as having anything to do with `‘rl`; the quoted form `'rl'`, however, is recognized by system functions such as `_gsv`.

System Command Names

System command names begin with a dollar sign, followed immediately by an unqualified name, which is the name of the command. The name is sometimes followed by a space and then by a sequence of characters whose meaning is specific to the command, usually separated from the name by a space.

Comments

Comments can appear either alone on a line or to the right of an expression. A comment is indicated by the a symbol (usually called “lamp,” since it looks like a bulb filament and since comments illuminate code), and it and everything to its right on the line constitute the comment. For example:

```
     a+b          a  This is the A+ notation for addition.
```

Infix Notation and Ambi-valence

A+ is a mathematical notation, and as such uses infix notation for primitive functions with two arguments. In infix notation, 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 *ambi-valent* (i.e., it uses “both valences”). A+ has extended the idea of ambi-valence to most of its primitive functions. For example, just as `-b` denotes the negative of `b`, so `b` denotes the reciprocal of `b`.

Defined functions cannot be ambi-valent.

Functions with one argument are called *monadic,* and functions with two arguments are called *dyadic*. One often speaks of the *monadic use* or *dyadic use* of an ambi-valent primitive function symbol.

Syntactic Classes

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 (`¢`)—including `¢Inf`, which we will come to later—instead of the more conventional minus sign (`-`), although negative exponents in the exponential format are denoted by the conventional minus sign.

Exponential format is of the form `1.23e5`, meaning 1.23 times 10 to the power 5, `¢5e2`, meaning -500, and `1e-2`, meaning .01. Only numbers can appear around the `e`. The one following it must be an integer—no decimal point—and have a regular minus sign if negative: a high minus there elicits a parse error report. A negative number before the `e` must have a high minus: a regular minus is considered to lie outside the format.

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 .00003. `Inf` can appear in such a list. If you omit the blanks, A+ will give you a numeric vector, but probably not the one you intended. If a number is being parsed and a character is encountered that can’t be part of the number, then a new number is started if the character could begin a number. For instance,

`1e-3.5 40.358.62.7` is read by A+ as 0.001 0.5 40.358 0.62 0.7 .

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. For a quote mark of the same kind as the surrounding quote marks to be included in a 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`. There is, however, a distinction between the two kinds of quotation marks.

Within single quotes (`'`) the C escape sequences and indeed any `\`c are not treated in any way, but left as is.

In strings contained within double quotes (`"`) these sequences and `\c` are treated as follows:

`\n` is replaced by a newline character;
`\`o, `\`oo, and `\`ooo (each o a digit) are replaced by a character (see below); and
the other sequences *simply have the leading backslash removed*.

These sequences and their translations are (where parenthesis indicates that A+ does not perform the substitution that the parenthesized term implies):

Table 4-1: Double-Quote Translations

Name	String	Translation	Comment
newline	`\n`	newline character	
(horizontal tab)	`\t`	`t`	for tab use `"\11"`
(vertical tab)	`\v`	`v`	
(backspace)	`\b`	`b`	for backspace use `"\10"`
(carriage return)	`\r`	`r`	for carriage return use `"\15"`
(formfeed)	`\f`	`f`	for formfeed use `"\14"`
(audible alert)	`\a`	`a`	
backslash	`\\`	`\`	
question mark	`\?`	`?`	
single quote	`\'`	`'`	
double quote	`\"`	`"`	
octal number	`\ooo`	a character	see below
(hex number)	`\x`*hh*	`x`*hh*	
(any other char)	`\`c	c	

Thus `"\?\\"` is equal to `'?\'` and `"\r\t"` is equal to `'rt'`; `\"` prevents the double quote from ending a string within double quotes, and `\\` allows literal inclusion of `\` in a translated string in double quotes.

The translation of an octal sequence— which is of *variable length* and could be shown as `\`[[o]o]o—is best understood as occurring in three steps. First, the digits to be translated are found: there is at least one (else this would not be an octal sequence) and at most three, but the end of the string and any nondigit character also act as terminators. Second, the string of digits is taken as an octal number and is translated to a decimal number. Any 8 and 9 digits are accepted as 10 octal and 11 octal, and any overflow is ignored, since only the 256 residue is used. Third, the ASCII character corresponding to that number is found. If the string being translated is `digits`, the translation is

`‘char'8´10 10 10¸ digits` where `1/( digits)/3` and `'digits` is `‘char`.

The foregoing implies these equivalences:
`"\99"ß "\121"     "\6a"ß "\006a"ß "\06",'a'     "\123456"ß "\123",'456'`

Symbols and Symbol Constants

A symbol is a backquote (`‘`) followed immediately by a character string made up of alphanumeric characters, underscores (_), and dots (`.`). Symbol constants can be thought of as character-based counterparts to numeric constants, aggregating several characters into a single symbol. Just as `1 2.34 12e3 3e5` is a list of four numbers, so `‘a.s ‘12 ‘b‘w_3` is a list of four symbols. A backquote alone represents the empty symbol.

A user name, like `balance`, can be put in symbol form by placing a backquote before it, as in `‘balance`. A user name in symbolic form is always taken to refer to a global object (see “Scope of Names”, page 177), never a local object. If it has no dot in it, it refers to a global object in the current context.

System variable names, like `‘rl`, are in the form of symbols. Unlike backquoted user names, they are not decomposable. If `var` is a user name, then `‘var` is recognized by A+ in certain situations as referring to the same object. A+ sees no relation, however, between `rl` and the system variable `‘rl`.

The Null

The Null is a special constant that can be formed as follows: `()`. It is neither numeric nor character, but has a special type, null. It is an empty vector, i.e., its rank is 1 and the length of its only axis is 0.

Variables

Variables are data objects that are named. They receive their values through Assignment, or Specification, which is denoted by the left-pointing arrow (ß). For example, the expression

```
      abcß1 2 3
```

assigns the three-element list consisting of 1, 2, and 3 to the variable named `abc`. Any user name can serve as a variable name. For more on assignment, see “Assignment, or Specification”, page 31.

Functions and Function Call Expressions

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. It may also include unnecessary parentheses and blanks; if it does not, we will call it *irredundant*. 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 be function expressions (see “Function Expressions”, page 29). For example, `f{9.98;.0775;«}`and `f{59;125;g}`, where `g` is a defined function, are valid function call expressions.

A function with no arguments, or parameters,—which must be a defined or system, not a primitive, function—is said to be *niladic*. The only valid irredundant function call expression for a niladic function `f` is `f{}`.

Functions with one argument, *monadic* functions, can be primitive, defined, or system. The valid irredundant function call expressions for a function `f` with one argument `a` are `f a` and `f{a}`. In the form `f a`, the blank is required only if `f` followed by some initial part of `a` would form a valid name.

Dyadic functions can also be primitive, defined, or system. The valid irredundant function call expressions for a function `f` with two arguments `a` and `b` are `a f b` and `f{a;b}`, where `a` is called the *left argument* and `b` the *right argument*. In the infix form, each blank is required only if its absence could cause a name to be extended, and if the left argument is itself an infix expression it must be parenthesized.

Functions with more than two arguments must be defined or system, not primitive, functions. The only valid irredundant function call expression for a function of more than two arguments `a`, `b`, ... , `c` is `f{a;b;...;c}`.

In functional expressions that use braces, any position adjacent to a semicolon can be left blank. For example, each of the following is a valid functional expression: `f{a;}`, `f{;b}`, `f{;}`, `f{;a;b}`, `f{;;b}`. However, if `f` is monadic then `f{}` is not valid because `f{}` is reserved for niladic function call expressions. When an argument position is legitimately left blank, A+ assumes that the argument is the Null.

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

Table 4-4, page 34, summarizes the function call expressions discussed here.

Operators and Derived Functions

There are three primitive formal operators in A+, known as Apply, Each, and Rank. 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 Apply and Each operators are both denoted by the dieresis, `¡`. For a function `f`, the function derived from the Each operator is denoted by `f¡`. The function `f` can be either monadic or dyadic, and `f¡` has the same valence as `f`. For a given function scalar `g`, where `g` is equal to `<{f}`, the function derived from the Apply operator is denoted by `g¡`. The function `f` can be either monadic or dyadic, and `g¡` has the same valence as `f`.

The Rank operator is denoted by the *at* symbol, `@`. Unlike Each, the Rank operator has both a function argument and a data argument. For a function `f` and data value `a`, the function derived from the Rank operator is denoted by `f@a`. This derived function has the same valence as `f`, which can be either monadic or dyadic.

A+ permits defined operators. As with primitive operators, only infix notation is allowed for operator and operands. Like Each, the operand of a monadic defined operator is to the left of the operator name. For example, if the operator is `opm` then `+opm` is the derived function for `+`. For a dyadic defined operator, one operand is on the left of the operator name and the other is on the right, like the Rank operator. For example, if the operator is `dyop` then `+dyop«` denotes the derived function for `+` and `«`. A dyadic defined operator can have a data right operand: see the note following Table 4-5, page 35. See also “Operator Syntax”, page 178.

Unlike a primitive operator, the valence of a function derived from a defined operator is not determined by the valence of the function operands, but, like a defined function, by the form of the operator definition.

There are five other symbols (`.˚/\fi`) that can appear with certain primitive function symbols, the resulting sequences representing functions. Their syntax might suggest that these symbols represent operators; however, not all primitive function symbols can be used in these sequences, and neither can defined function names. Consequently it would be misleading to think of them as formal operators, so we have simply listed all the sequences that are allowed. It is often convenient, however, to speak loosely of these sequences as representing derived functions, and of the five symbols in question as representing operators, namely, Inner Product, Outer Product (`˚.`), Reduce, Scan, and Bitwise.

From now on, the general terms *operator* and *derived function* will include Apply, Each, Rank, defined operators, their derived functions, and the “operators” and “derived functions” in Table 4-2.

Table 4-2: Special Character Sequences (Quasi-Operators)

“Operator” Name	“Derived” Functions
Bitwise	`'fi` (Cast and Or) `^fi ~fi <fi /fi =fi ƒfi >fi ¤fi`
Inner Product	`+.«   ˜.+   .+`

Outer Product	`°.+ °.- °.« °.  °.* °.˜ °.`
	`°.˝ °.< °.> °./ °.ƒ °.= °.¤`

Reduction	`+/  «/  ^/  '/  ˜/   /`
Scan	`+\  «\  ^\  '\  ˜\   \`

Operator call expressions should be understood in terms of derived functions and function call expressions. Namely, an operator symbol and its function operands, or in the case of the Rank operator, its function operand to its left and its data object operand immediately to its right, form a derived function. A derived function is syntactically like any other function, and so can be used in the function position of any function call expression, as in `f@a{c;d}` and `b f@a c` . See Table 4-3 for a summary; it shows both irredundant expressions and expressions in which the derived functions are parenthesized. As in function call expressions, the blanks are not required in some instances and the left argument may need to be in parenthesis; moreover, a constant data operand and a constant right argument may require punctuation to separate them.

Table 4-3: Operator Call Expressions

Operator Valence	Forms for Derived Function Having Monadic Valence		Forms for Derived Function Having Dyadic Valence	
monadic	`(f op)a`	`f op a`	`a(f op)b`	`a f op b`
	`(f op){a}`	`f op{a}`	`(f op){a;b}`	`f op{a;b}`
dyadic	`(f op g)a`	`f op g a`	`a(f op g)b`	`a f op g b`
	`(f op g){a}`	`f op g{a}`	`(f op g){a;b}`	`f op g{a;b}`

Function Expressions

The function arguments of operators are function expressions. The simplest function expressions are the names of defined functions and the symbols for primitive functions *other than Assignment and Bracket Indexing*. Any formulation of a derived function is also a function expression (see “Operators and Derived Functions”, page 28).

Function expressions are limited to infix notation, since operators are limited to it.

A function expression can be enclosed in parentheses. For example, `a(f@1)b` is equivalent to `a f@1 b`. Moreover, a function expression is a valid function argument to a formal operator, and therefore quite complicated function expressions can be built. For example, `+/` is a function expression, and therefore so are the following: `+/¡`, `+/¡¡`, and `+/¡@a`. See “Scope Rules for Function Expressions”, page 32.

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 or an expression in parenthesis, `a`, `b`, ..., `c` denote expressions, and the number of semicolons is at most one less than the rank of the array being indexed. (The form `x[]` is, however, allowed for scalars.) 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. If there are no semicolons, the space between the left and right brackets can be empty. Inserting semicolons immediately to the left of the right bracket does not change the meaning of the entire expression, as long as the maximum allowable number of semicolons is not exceeded. The form `[a;b;...;c]` is an *index group*. See “Sequences of Expressions”, page 32, and “Bracket Indexing”, page 57.

Expression Group

An expression group is a sequence of expressions contained in a pair of braces in which the expressions are separated by semicolons, where there is not a function expression immediately preceding it (except perhaps for spaces), so it is not a set of arguments for a function. Any of the expressions can be null, consisting of zero or more blanks. For example:

```
{a;b;...;c;}
```

and

```
{a;b;...;c}
```

are expression groups, where `a`, `b`, ... ,`c` denote expressions. See .

Expression Result and Expression Group Result

The result of an expression is the result of the last function executed in the expression, whether primitive, defined, or derived. See “Well-Formed Expressions”, page 35.

The result of an expression group is the result of the last expression executed. It is possible that the last expression in the group may not be the last one executed—indeed, may not be executed at all; see the “Result” section, page 89.

Strands

Aggregate data objects (nested arrays) can be formed by separating the individual data objects by semicolons and surrounding the result with a pair of parentheses. For example:

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

where `a`, `b`, ... , `c` denote expressions. Any of these expressions can be function expressions. There must be at least one semicolon. See “Sequences of Expressions”, page 32.

Function Scalars

The above strand notation produces objects with at least two elements. One-element aggregates of data can be formed with the primitive function Enclose (page 66), denoted by `<`. A one-element object such as

```
<{a}
```

where `a` is a *function expression*, is called a *function scalar*.

The symbol `¡`, used also for the Each operator, serves as the Apply operator when the operand (argument) of the operator is a function scalar. For example, `a(<{ })¡b` is `a b`.

Assignment, or Specification

The Assignment primitive, denoted by ß, is used to associate a name with a value. For example:

```
aß1
fß+
```

assigns the value 1 to the name `a` and the function Add to the name `f`. The name to the left of the assignment arrow is assigned the value of the expression to the right. If that expression is a function expression, the name to which it is assigned represents a function—not the name of a function, but a function itself. Otherwise it represents a variable.

A series of names can be associated with a series of values, using strand notation; for example,

```
(a;b;c)ß(1 2 3;3 4 7;'txt')
```

Ordinary Assignment can also be expressed as `(a)ßb`. Any appearance of `aßb` inside a function or operator definition means that `a` will be a local variable, if `a` is an unqualified name. The form `(a)ßb` can be used to assign a value to the global variable `a`, provided that `aß...` doesn't appear elsewhere in the definition. If both `aß...` and `(a)ß...` appear, they are equivalent: the latter has no special significance.

Assignment behaves somewhat like a dyadic function, in that it has a result, namely, the right argument. The left argument expression is syntactically limited to certain forms. See Table 7-2, page 92, for a summary of Selective Assignment target expressions, which are additional to those in ordinary assignment.

Assignment, in any form, cannot be the operand of an operator.

Precedence Rules

Precedence rules describe a hierarchy in the syntactic elements of a language that determines how these elements are grouped for execution in an expression. For example, in mathematics « has higher precedence than +, which means that « is evaluated before +. For example, in the mathematical expression a«b+c, the subexpression a«b is grouped for execution, and the result is added to c.

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.

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.

```
b+c+d      ª Read as: "b plus the result of c plus d."
x- y       ª 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 "`b` plus the result of `c` plus `d`" means that `c+d` must be formed first, and then added to `b`. 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`.

To be sure, reading from left to right is not necessarily associated with execution from right to left. For example, the expression b c+d 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 b divided by c is formed and added to d, and consequently the expression is read as “b divided by c, [pause] plus d,” while in A+, b is divided by c+d. The order of execution is controlled by the relative precedence of the functions, or operations. In mathematics, division has higher precedence than addition, so that in b c+d, division is performed before addition.

Another way to say that A+ expressions are executed from right to left is that functions have long right scope and short left scope. For example, consider:

```
a+b-c e«f
```

The arguments of the subtraction 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 subtraction symbol and moving to the left until the smallest possible complete subexpression is found. In this example it is simply the name `b`. If the first nonblank character to the left of the symbol had been a right parenthesis, then the left argument would have included everything to the left up to the matching left parenthesis. For example, the left argument of subtraction in `a+(x b)-c e«f` is `x b`.

The right argument is found by starting at the function symbol and moving to the right, all the way to the end of the expression; or until a semicolon is encountered at the same level of parenthesization, bracketing, or braces; 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 subtraction is everything to its right. If the case of `a+b-(c e)«f`, the right argument is also everything to its right. However, for `a+(b-c e)«f`, the right argument is `c e`.

Scope Rules for Function Expressions

Interestingly enough, the scope rules for function expressions are the mirror image of those for ordinary expressions. Namely, operators have long scope to the left and short scope to the right. For example, `+/¡@a` is equivalent to `((+/)¡)@a`, and if `dyop` is a dyadic defined operator, `+dyop ¡` is equivalent to `(+dyop )¡`, not `+dyop( ¡)`.

Sequences of Expressions

Index groups, expression groups, and strands are forms for sequences of expressions separated by semicolons. The expressions in an expression group are executed in the order suggested for reading, from left to right, like successive statements in a function. Index groups and strands, however, fall within other expressions and are executed right to left. For example, if the variable `a` has the value 2 and the strand

```
bß(aß5;a«a)
```

is executed, the value in the second element of `b` will be 4, proving that the assignment `aß5` happened after the multiplication `a«a`. (A Strand Assignment, however, like an expression group, is executed left to right, after its righthand argument has been evaluated in the usual way.)

To improve readability in source files, sequences of expressions are often broken at the semicolons and continued on the next physical line. Note that in such cases for expression groups the left to right order of execution for the expressions within a sequence becomes a natural top to bottom order.

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 defined in either of the 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 referred to by `&0` (top of the stack), `&1`, and so on. These objects can be examined and respecified, and execution resumed (ß). The left to right order of arguments generally corresponds to increasing stack numbers.

In the definition of a dependency `a`, the symbol `&` refers to that definition but `a` always denotes the (stored) value of `a`, whereas in the definition of a function `f`, both `&` and `f` denote the definition of `f`.

Control Statements

For the interpretation of these control statements, see “Control Statements”, page 120. The words case, do, else, if, and while are reserved by A+; they cannot be employed as user names.

Case Statement

The form of a case statement is the word case, followed by an expression in parentheses, followed by an expression group. When case followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group.

Do Statement

There are two do statements, which together have the same syntax as an ambi-valent primitive function (with the word do in place of the function symbol). Both the monadic and dyadic forms have an expression or expression group to the right of the word do. The dyadic form also has an expression to the left which would serve as the left argument if the word do were the name of a dyadic function. In the absence of pending punctuation, if do is entered alone on a line, it is taken to be complete, and echoed by A+, and if it is preceded by an expression but followed by nothing, a parse error is reported.

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 group. When if followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group.

If-Else Statement

The form of an if-else statement is the word if, followed by an expression in parenthesis, followed by an expression or expression group, followed by the word else, followed by another expression or expression group. When an if-else is entered, if there is nothing following the else, a parse error is reported in the absence of pending punctuation.

While Statement

The form of a while statement is the word while, followed by an expression in parentheses, followed by another expression or an expression group. When while followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group. If the expression in parenthesis is valid and nonzero, it is necessary to interrupt execution (by **Control-c Control-c**) before anything else can be done.

Function Definitions

A function definition consists of a function header, followed by a colon, followed by the function body, which is either an A+ expression or an expression group.

Function headers take the same forms as functional expressions (see “Functions and Function Call Expressions”, page 27), except that only names can appear and none can 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 name followed by a left brace, followed by a list of from zero to nine argument names separated by semicolons, and terminated by a right 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. Names must appear in all positions of the argument list — no position can be left empty. (In a niladic function definition no argument position is left empty; there just is no argument position.)

A function with one argument can be defined with either the monadic form of function header or the general form and a function with two arguments can be defined with either the dyadic form or the general form. In a reference to the function, either form (of the correct valence) can be used, no matter how it was defined.

The number of arguments of a defined function is nine or fewer. See Table 4-4 for a summary of function header formats.

Function Result

The result of a defined function is the result of the expression or expression group that forms the function body. The result can be used in the same ways as the result of a primitive function.

Table 4-4: Function Call Expressions and Function Header Formats

Valence	Forms
niladic	`f{}`
monadic	`f a` or `f{a}`
dyadic	`a f b` or `f{a;b}` (A position next to a semicolon can be empty for calls)
general	`f{a;b;...;c}` (A position next to a semicolon can be empty for calls)

Operator Definitions

An operator definition consists of an operator header, followed by a colon, followed by the body of the definition, either an A+ expression or an expression group. The header must be in infix, not general, form.

An operator can be monadic or dyadic, depending on whether it has one argument or two, and the derived function can also be monadic or dyadic. Consequently there are four forms for the header. See Table 4-5 for a summary of operator header formats.

Note the parentheses in the forms in “Operator Header Formats”. While parentheses are not necessary in operator call expressions, they are necessary in operator definition headers to specify the function expression part. Compare with “Operator Call Expressions”, page 29.

Operator Result

The result of a defined operator, which is strictly speaking the result of the derived function, is the result of

Table 4-5: Operator Header Formats

Operator Valence	Monadic Derived Function	Dyadic Derived Function
monadic	`(f op)a`	`a(f op)b`
dyadic	`(f op h)a`	`a(f op h)b`

the expression or expression group that forms the body of the definition. The result can be used in the same ways as the result of a primitive operator.

NOTE: In the dyadic form, if the right operand is the letter `g`, then it must be a function; otherwise, it must be data unless every occurrence in the body of the operator syntactically requires it to be a function.

Dependency Definitions

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 group. An itemwise dependency has the same form except that the name is followed by `[i]` where `i` can be any unqualified user name (except the name of the dependency).

Dependency Result

The result of a dependency is either a value that was assigned to the name, or the result of the expression or expression group that forms the definition, or, for itemwise dependencies, a combination of the two—see "Dependencies", page 186. The results of dependencies are referenced in the same way that values of variables are referenced, simply by their names.

Well-Formed Expressions

A well-formed expression is one of the basic forms described above, in which all of the constituent expressions are well formed. The potential for complicated expressions arises from the fact that every one of these basic forms produces a result and can therefore be used as a constituent in other forms, except that the right arrow ( ) can only appear alone and the left arrow (ß) must appear alone unless it has an expression to its right. In this building of expressions from simpler ones 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", page 31, one can imagine that parts of the expression are executed and replaced by their results, and then other parts are executed using these results, and are replaced by their results, and so on. Ultimately the execution comes to the last expression to be executed, which is called the principal subexpression. Once it is executed, its value is the value of the expression. If the principal subexpression is a function call expression or operator call expression, that 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 `x+y` and `z` is `x-y`. In this case we do not refer to a principal function; the last thing done in executing the expression is what is implied by the strand notation—enclosing `w` and `z` and catenating them.