APL# Language Description 2011-09-02

Introduction

The following pages describe the syntax of APL# using BNF-like notation.

'a' or "ab" Double quotes are used to quote single quotes.
* Zero or more instances of the preceding term.
? Zero or one .. .. ..
+ One or more .. .. ..
⍝ Comment, example or informal description.
Comments of the form "⍝ [...]" refer to following note [...].
.. Unicode sequence, such as 'A'..'Z' or '0'..'9'

Example:

DecimalNumber → '¯'? DecimalDigit+ ('.' DecimalDigit+)? ⍝ 0 99 ¯0.41
DecimalDigit → '0'..'9'

NB: The notation used in this document does not constitute BNF in the formal sense because the rules are ambiguous. For example, the fact that a name can reference an Array, a Function or an Operator, means that Name appears on the right of a number of rules:

ArrayAtom → Name
FuncAtom → Name
Mop → Name
...

This is another way of saying that we can't lexically parse: A B C

A note for the Dyalog 2011 Conference Edition

This document is a description of the **syntax** of APL#, which is a work-in-progress. Its primary use has been in resolving and recording language features during development. It is not really adequate as a tutorial though we hope that some of the definitions and examples may give enough of a hint for interested people to experiment. A more user-friendly introduction should follow in due course - JMS.

Unless otherwise stated, primitive functions and operators conform to the Extended APL Standard: **ISO/IEC 13751.**

Rule	Example	
ExprList → Expr? (Brk Expr?)* Brk → '⋄' | EndLine	⍝ 3 ⋄ {⍵} ⋄ ⍞	⍝ [X0]
Expr → ArrayExpr	⍝ 2+3	
| FuncExpr	⍝ +.×	
| OperExpr	⍝ ⍨⍣¯1	
| '(' ExprList ')'	⍝ (2 ⋄ 3)	⍝ [X1]
ArrayExpr → ArrayAtom	⍝ 88	
| Function ArrayExpr	⍝ ⍳6	
| ArrayAtom Function ArrayExpr	⍝ 2+⍳6	
| Mutation	⍝ v[0]←4	
| ArrayNaming	⍝ a←7	⍝ [A1]
| Nilad	⍝ System.DateTime.Now	
| NiladNaming	⍝ throw←⍤{.→?6 6}	⍝ [NN0]
| CtrlStruct	⍝ :If 1 ⋄ 6 ⋄ :End	
| Guard	⍝ 1 : 6	⍝ [CG0]
ArrayAtom → Name	⍝ vec	
| DecimalNumber | CharLiteral | String	⍝ "this"	
| Strand	⍝ 88(1+2)	
| Indexing	⍝ vec[1 2]	
| ArrayAtom '.' ArrayAtom	⍝ #.(⍳4)	
| ArrayAtom '⍤.' SimpleName	⍝ (# #)⍤.Rank	
| NewSpace		
| '⍬' | '#'		
| 'α' | 'ω' | 'αα' | 'ωω'		
| '(' ArrayExpr ')'		
ArrayNaming → NameStruct '←' ArrayExpr	⍝ ((a b)c)←(1 2)(3 4)	
NameStruct → Name | '(' NameStruct+ ')'	⍝ ((a b)c)	⍝ [N0]
Strand → ArrayAtom ArrayAtom+	⍝ ⍬ (2 3)	⍝ [A2]
Indexing → ArrayAtom Subscript	⍝ mat[;0][1 2]	
Mutation → Name Subscript '←' ArrayExpr	⍝ vec[]←0	
Subscript → '[' ArrayExpr? (';' ArrayExpr?)* ']'	⍝ [A;;A←3;]	⍝ [S0]
NewSpace → '[[' ExprList ']]'	⍝ [[a←4 ⋄ b←5]].(a b)	⍝ [NS0]

⍝ NewSpace creates and returns a new namespace in which the enclosed expression
⍝ has been evaluated. See the Examples below.

Rule	Example
Function → FuncAtom | FuncDerived	⍝ {⍵+1} ⍝ ,∘{⍵+1}/ ⍝ [F0]
FuncExpr → Function | 2-train | 3-train | FuncNaming	⍝ [F1]
FuncAtom → Name | PrimFunc | DefinedFunc | 'αα' | 'ωω' | '∇' | ArrayAtom '.' FuncAtom | ArrayAtom '`.' SimpleName | '(' FuncExpr ')'	⍝ [F2] ⍝ #.⍳ ⍝ ns`.Method
FuncDerived → LeftOperand MopExpr	⍝ +/ +.×
FuncNaming → Name '←' FuncExpr	⍝ nxt ← 1+ ⍝ [F3]
LeftOperand → ArrayAtom | FuncAtom | FuncDerived	
RightOperand → ArrayAtom | FuncAtom	
2-train → ArrayAtom Function | Function Function	⍝ 1+ ⍝ left-arg currying ⍝ ⍴⍴
3-train → ArrayAtom Function Function | Function ArrayAtom Function | Function Function Function	⍝ 1⌈⍴ ⍝ ⊢0⊣ ⍝ +/÷⍴
Nilad → DefinedNilad | SystemNilad	⍝ {.→?2} TimeNow
NiladNaming → Name '←' '`' Nilad	⍝ t←`{.→TimeNow}

Rule

```
OperExpr ← MopExpr
         | DopAtom
         | OperNaming
         | '(' OperExpr ')'
         | '∇∇'

OperNaming ← Name '←' OperExpr

MopExpr ← MopAtom
        | DopAtom RightOperand

MopAtom ← Name
        | PrimMop
        | DefinedMop
        | ArrayAtom '.' MopExpr
        | ArrayAtom '`.' SimpleName

DopAtom ← Name
        | PrimDop
        | DefinedDop
        | ArrayAtom '.' DopAtom
        | ArrayAtom '`.' SimpleName
```

Example

```
⍝ ∘2
⍝ ∘
⍝ limit ← ⍣≡
⍝ ((∘)÷)
⍝ operator self-reference

⍝ inverse ← ⍣¯1

⍝ ⍤⍤
⍝ ⍣¯1                    ⍝ [M0]

⍝ ⍤⍤
⍝ {{αα}ω→αα ω}
⍝ #.{{f}ω ← f ω}
⍝ TBD Example needed.

⍝ ⌸
⍝ {α{f g}ω→f α g ω}      ⍝ (atop)
⍝ #.⌸
⍝ TBD Example needed.
```

Rule

Example

```
PrimFunc ← '+' | '-' | '×' | '÷' | '*' | '⍟' | '⌊' | '⌈' | '|' | '!'
         | '○' | '<' | '≤' | '=' | '>' | '≥' | '≠' | '∧' | '∨' | '⍲'
         | '⍱' | '~' | '?' | 'ρ' | 'ι' | '∈' | '↑' | '↓' | '⍪' | '⍋'
         | '⍒' | '⌽' | '⍉' | 'Θ' | '∪' | '⊃' | '⊂' | '∩' | '⍎' | '⍕'
         | '⌷' | '⍪' | '⊢' | '≡' | '≢' | '¤' | '$' | Slash

PrimMop ← '⍨' | '⍣' | Slash                                          ⍝ [M1]

PrimDop ← '⍤' | '⍠' | '∘' | '⍣' | '⍢' | '.'

Slash ← '/' | '\' | '⌿' | '⍀'                                        ⍝ [SL0]

⍝ Where:
⍝
⍝ ⌷ Squad-indexing as in Dyalog
⍝ ¤ New space. Analogous to ⎕NEW in Dyalog
⍝ $ Converts between "string" and 'character vector'
⍝
⍝ ⍤ Rank
⍝ ⍠ Variant
⍝ ⍣ Power
⍝ ⍢ Dual/Under

Guard ← ArrayExpr ':' Expr                                           ⍝ [G0]

Name ← SimpleName
     | '⎕'                                                           ⍝ ⎕←(A ⎕ B)←⎕ C D
     | ArrayAtom '.' SimpleName                                      ⍝ [[]].A←4

SimpleName ← IdentifierChar (IdentifierChar | DecimalDigit)*         ⍝ [NM0]

IdentifierChar: The Unicode Standard defines which chars may be used in identifiers.
```

Function and Operator Definition

```
DefinedFunc ← '{' (Larg? Rarg '←')? ExprList '}'                    ⍝ [DF0]

DefinedMop ← '{' Larg? '{' Land '}' Rarg '←' ExprList '}'           ⍝ [DM0]

DefinedDop ← '{' Larg? '{' Land Rand '}' Rarg '←' ExprList '}'

DefinedNilad ← '{' '.'  '←' ExprList '}'

Larg ← NameStruct | 'α'
Rarg ← NameStruct | 'ω'
Land ← NameStruct | 'αα'
Rand ← NameStruct | 'ωω'
```

Control Structures

```
CtrlStruct → IfStruct | WhileStruct | RepeatStruct                A [CO]
           | ForStruct | SelectStruct | WithStruct
           | Label | GoToStruct
           | ThrowStruct | TryCatchStruct
           | ContinueStruct | LeaveStruct
           | PropertyStruct | ReturnStruct

IfStruct → ':If' ArrayExpr Brk
            AndOr?
                ExprList Brk
         ( ':ElseIf' ArrayExpr  Brk
               AndOr?
                ExprList Brk      )*
         ( ':Else' Brk
                ExprList Brk      )?
         ( ':End' | ':EndIf')

WhileStruct → ':While' ArrayExpr Brk
               AndOr?
                   ExprList Brk
             (':End' | ':EndWhile' | Until)

RepeatStruct → ':Repeat' Brk
                    ExprList Brk
              (':End' | ':EndRepeat' | Until)

Until → ':Until' ArrayExpr Brk
         AndOr?

AndOr → (':AndIf' ArrayExpr Brk )+
      | (':OrIf'  ArrayExpr Brk )+

ForStruct → ':For' NameStruct (':In' | ':InEach') ArrayExpr Brk
                 ExprList Brk
           (':End' | ':EndFor')

SelectStruct → ':Select' ArrayExpr Brk
           (  (':Case' | ':CaseList') ArrayExpr Brk
                    ExprList Brk        )*
           (   ':Else' Brk
                     ExprList Brk        )?
              (':End' | ':EndSelect')

WithStruct → ':With' ArrayExpr Brk
                    ExprList Brk
              (':End' | ':EndWith')
```

```
TryCatchStruct → ':Try' Brk
                     ExprList Brk
                ( ':Catch' ArrayExpr Brk                    ⍝ :Catch RankError
                       ExprList Brk       )*
                ( ':Finally' Brk
                       ExprList Brk       )?
                 (':End' | ':EndTryCatch')

PropertyStruct → ':Property' Name Brk
                ( ':Get' Function  Brk)?
                ( ':Set' Function  Brk)?
                ( ':End' | ':EndProperty')

ThrowStruct → ':Throw' ArrayExpr                            ⍝ :Throw RankError

ReturnStruct → ':Return' ArrayExpr                          ⍝ :Return 2+3

Label → ':Label' Name Brk                                   ⍝ :Label Loop     [L0]

GoToStruct → ':GoTo' Name                                   ⍝ :GoTo Loop      [G0]

DecimalNumber → '¯'? DecimalDigit* ('.' DecimalDigit+)? (Exponent)?     ⍝ 0 99 ¯0.41E¯7
Exponent → ('E' | 'e') '¯'? DecimalDigit+
DecimalDigit → '0'..'9'

CharLiteral → "'" UnicodeCharacter* "'"                     ⍝ 1=≡'charvec'
String      → '"' UnicodeCharacter* '"'                     ⍝ 0=≡"string"

EndLine → Comment? '\r'? '\n'
Comment → '⍝' PrintableChar*                                ⍝ like this

PrintableChar: See Unicode Standard.
```

Trains

Train definitions are under review. 3-trains follow J's definitions, extended with the (f B h) case, which is handy for constant function: ⊢0⊣ and (Array Function)* sequences such as (32 + 1.8 ×).

Monadic 2-train Ag implements left-argument currying:

```
    (1+)2
3
```

In the following, f, g and h are functions; A and B are arrays:

2-trains

```
 (f g)ω ← f g ω                    ⍝ fg        cf:  f∘g
⍺(f g)ω ← f ⍺ g ω                  ⍝ fg        cf: J's "atop"

 (A g)ω ← A g ω                    ⍝ Ag        left-arg currying
⍺(A g)ω ← SyntaxError              ⍝ Ag
```

3-trains

```
 (f g h)ω ← (  f ω)g(  h ω)        ⍝ fgh monadic fork
⍺(f g h)ω ← (⍺ f ω)g(⍺ h ω)        ⍝ fgh dyadic fork

 (A g h)ω ← A g(  h ω)             ⍝ Agh cf J's:  (N V V)ω
⍺(A g h)ω ← A g(⍺ h ω)             ⍝ Agh cf J's: ⍺(N V V)ω

 (f B h)ω ← f(B h ω)               ⍝ fBh              (V N V)ω
⍺(f B h)ω ← ⍺ f(B h ω)             ⍝ fBh             ⍺(V N V)ω
```

Examples

Here is a (nonsensical) APL# operator, which shows a medley of features:

```
myop ← {(larg0 larg1) {f g} rarg →        ⍝ Dyadic operator
      lucky←`{.→?2}                       ⍝ named nilad
      □←:If lucky ◇ f larg0               ⍝ nilad "lucky" evaluated
        :Else     ◇ g rarg ◇ :EndIf
      [[                                  ⍝ new space returned, in which:
            a←(lucky : f larg0 ◇ g rarg)  ⍝⍝ name-scope? See below
            x←{ [[a←⍵]] }a                ⍝ new space x, in which: x.a ≡ a
            y←[[
                  n←f g lucky             ⍝⍝ name-scope? See below
            ]]
            (x.y y.x)←y x                 ⍝ cross-references.
      ]]
}

⍝⍝ In the ⍝⍝ lines above, it seems intuitive (to JMS) that names (lucky,
⍝⍝ f, g, larg1, rarg) should be resolved _lexically_ within the capsule.
```

Structured naming of arguments:

```
      { ((a b)c) → a(b c) }               ⍝ right rotation of binary tree ⍵

      { (ll x){αα}(y rr) →                ⍝ Turing machine
            x=STOP : ll x (y rr)          ⍝ stop instruction: tape triple
            X←αα x                        ⍝ update of cell under read head
            (ll X)y ∇∇ rr                 ⍝ rewrite cell and skip right one frame.
      }
```

NewSpace structure contains temporary variables:

```
      [[                ⍝ unnamed temp space for calculation:
          tax←0.1
          nett ← +/37.20 42.50 25.30
          all ← nett × 1 tax, tax+1
      ]].all
105 10.5 115.5
```

Control Structures return results

```
      (
          n←27
          :Repeat
              n = :if 2|n ◇ 1+3×n
                  :Else     n÷2
                  :EndIf
          :Until n=1
      ), ' is the answer!'
1 is the answer!
```

Guard used in parenthesized expressions

```
      n←1 ◇ (
          2|n : 1+3×n
                n÷2
      ), ' is the result '
4 is the result
```

Connecting spaces

Unlike in Dyalog, APL# spaces do not have an inherent parent-child relationship. They are just free-floating but can be connected to each other via names.

```
      (b.a a.b) ← (a b) ← [[]] (⍎'')
      b.a a.b ≡ a b
1
```

Names in spaces in functions have lexical scope

All function-local names are visible from within NewSpace or :With structures. This is in contrast to Dyalog, where local _namespaces_ are not visible.

```
      'A'{
          var←⍺ ⋄ fun←, ⋄ spc←[[c←⍵]]     ⍝ names local to function

          ⍬ ← 'B', :With spc              ⍝ within space,
              var fun spc.c               ⍝ all outer names are visible
          :EndWith

          ⍬ ← 'C', [[                     ⍝ within a NewSpace,
              y ← var fun spc.c           ⍝ all outer names are visible
          ]].y

          ⍬ ← 'H', spc.(                  ⍝ within space,
              var ← 'O'                   ⍝ space-global name shadows outer.
              var fun spc.n               ⍝ other outer names are visible
          )
      }'T'
BAT
CAT
HOT
```

Lexical Scope Considerations

APL# enjoys the same lexical scope conventions as Dyalog, except for a couple of corrections, marked “!!”.

```
      which←{
          scope←'lexical'
          sub←{scope}   ⍝ which “scope” is in view?
          {
              scope←'dynamic'
              sub ⍵
          }⍵
      }
      which'scope'
lexical
```

Similarly, if the name-referencing function is an operand to an inner operator:

```
      which←{
          scope←'lexical'
          {scope}{{⍺⍺}⍵ →           ⍝ which “scope” is in view?
              scope←'dynamic'
              ⍺⍺ ⍵                  ⍝ operand evaluated in definition-,
          }⍵                        ⍝ rather than reference- context.
      }
      which'scope'
lexical
```

Note that, unlike Dyalog, this convention extends to primitive operands:

```
      which←{
          scope←'lexical'
          ⍺⍺{{⍺⍺}⍵ →                  ⍝ where is ⍺⍺ evaluated?
              scope←'dynamic'
              ⍺⍺ ⍵                    ⍝ operand evaluated in definition-,
          }⍵                          ⍝ rather than in reference- context.
      }
      which'scope'                    ⍝ this differs from Dyalog !!
lexical
```

A related issue is the conflict between lexical scope rules and “dotted” expressions:

```
      {
          ns←[[where←'ns']]
          where← 'fn'
          ⎕←'a: ', where
          ⎕←'b: ', ns.where
          ⎕←'c: ', ns.(where)
          ⎕←'d: ', [[v←where]].v
          ⎕←'e: ', ns.{where}⍬
          ⎕←'f: ', ns.[[v←where]].v
      }⍬
a: fn
b: ns
c: ns
d: fn
e: ns
f: ns
```

Lexical scope inside :With control structure

```
      ns←{a →
          :With ⍎''
              b←a ⍝ outer name “a” is visible.
          :End
      }88
      ns.b
88
      ns.a
ValueError
```

Two-by-Two Binding Strength Table

The BNF-like notation presented here does not make explicit the strengths with which adjacent tokens are bound. The BNF could be extended but the additional complexity (combinatorial explosion) of doing so might swamp the whole thing.

An example of the problem is to define the relative bindings strengths of:

```
      ArrayAtom ArrayAtom                ⍝  ⍬ ⍬
      ArrayAtom Subscript                ⍝  ⍬[]
      '.' ArrayAtom                      ⍝  #.(⍬)
      Dop RightOperand                   ⍝  ,∘4
```

... in order to determine where the parser will parenthesize the following expressions:

```
                        ⍝ Dyalog:
      A B.C         ⍝ A(B.C)
      A B[]         ⍝ (A B)[]
      #.B[]         ⍝ (#.B)[]
      ,∘⍬ ⍬         ⍝ ,∘(⍬ ⍬)
      ,∘#.⍳         ⍝ ,∘(#.⍳)
      ,∘⍬[⍬]              ⍝ (,∘⍬)[⍬]
```

J.D.Bunda and J.A.Gerth published a method for describing the relative binding strengths of APL tokens. See: “APL two by two-syntax analysis by pairwise reduction” Proceedings of the international conference on APL, 1984.

The following table shows for each class of token:

DOT	Space-referencing dot. For example the ‘.’ in: #.⍳
DX	“Dotted” array. For example, the ‘#.’ in #.V
A	Array.
MF	Function bound with left argument. ⍝ [MF0]
F	Ambi-valent function.
MOP	Monadic operator or dyadic operator bound with a right operand.
DOP	Dyadic operator.
IDX	An indexing or Axis ‘[‘ token, as in V[0...
/	One of the four slash tokens: / ⌿ \ ⍀.

Where there is a binding, the table shows the binding strength and the classification of the resulting bound pair.

	DOT	A	DX	MF	F	MOP	DOP	IDX	/
DOT									
A	8 DX	6 A			2 MF	4 F		4 A	2 MF
DX		7 A		7 MF	7 F	7 MOP	7 DOP		7 /
MF		1 A							
F		1 A				4 F		4 F	4 /
MOP									
DOP		5 MOP			5 MOP				5 MOP
IDX									
/						3 MOP			

TBD: This table is an **extract** from the one used by the APL# parser. In particular, it does not include function trains.

Notes

[X0] An ExprList is a (possibly empty) list of Brk-separated expressions.
[X1] The value of an ExprList is the value of its last-executed Expr.
Note that an ExprList may be empty. In this case, if the context requires a result, a NoResult error will be generated.
[A0] ArrayAtom: is tightly bound; suitable for an operand or a left-argument or strand-element.
[A1] ArrayExpr: is loosely bound; suitable for a right argument.
[NN0] The result of a NiladNaming is the evaluation of the Nilad: 0≡A←⍤{→0}
[CG0] Control Structures and Guards can return only Array values.
[N0] For example: ((a b)c)
[F0] Function: binds tighter than its right-argument array expression.
[F1] FuncExpr: is loosely-bound; suitable for a function-naming.
[T0] Train.
[F2] FuncAtom: is tightly bound; suitable for a left operand.
[F3] FuncNaming: allows f ← g ← + . Ditto OperNaming for operators.
[S0] Subscripts require that adjacent ArrayExpr are separated by semicolons.
[A2] Strand (Array) binds tighter than Train (Expression): 1 2 + → (1 2)+
All functions are syntactically ambivalent; functions not defined for a particular valence generate their own error when called.
[M0] This gives us right-operand currying, eg: inv←⍣¯1
[M1] Tokens / \ ⌿ ⍀ appear as both functions and operators, as in Dyalog.
[NS0] NewSpace: [[ExprList]] returns a reference to a new space in which ExprList has been evaluated.
[C0] CtrlStruct: is an Expression and so evaluates to an Array.
[L0] Label names must be unique within a procedure but do not clash with (and so can be duplicates of) the names of objects, such as arrays or functions.
[G0] The :GoTo construct cannot pass control across a {..} boundary. In other words, it may not be be used to branch into or out of a procedure.
[SL0] The four slash tokens / \ ⌿ ⍀ continue to be “schizophrenic” as in Dyalog APL. With an array to their left, they are functions; with a function to their left, they are operators. For example, in: (1 2/¨3 4) the vector 1 2 is an argument to function /¨, rather than an operand to operator /. In this case, ¨ distributes its function operand between corresponding pairs of arguments: (1 2/¨3 4) ↔ (1/3)(2/4)
[G0] The guard syntax allows (g0 : g1 : true ⋄ false). This should be interpreted as a shorthand for: (g0 : (g1 : true ⋄ false) ⋄ false) or:

```
      DefinedFunc:If g0 ⋄ :AndIf g1 ⋄ true ⋄ :Else ⋄ false ⋄ :Endif
```

In other words, Guard-colons following the first one are treated as AndIfs.
[NM0] Unicode char, which can be used as an identifier. See The Unicode Standard.
[DF0] The function “signature” is optional for ease of importing D-functions from Dyalog.
[DM0] The operator signature is mandatory and so must be added to D-operators imported from Dyalog.

TBD: Selective specification