APL# Language Description

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.
..Unicode sequence, such as 'A'..'Z' or '0'..'9'

Example:

DecimalNumber → '−'? DecimalDigit+ ('.' DecimalDigit+)?
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**

ExprList → Expr? (Brk Expr?)*

Brk → '⋄' | EndLine

Expr → ArrayExpr
   | FuncExpr
   | OperExpr
   | '( ExprList )'

ArrayExpr → ArrayAtom
   | Function ArrayExpr
   | ArrayAtom Function ArrayExpr
   | Mutation
   | ArrayNaming
   | Nilad
   | NiladNaming
   | CtrlStruct
   | Guard

ArrayAtom → Name
   | DecimalNumber | CharLiteral | String
   | Strand
   | Indexing
   | ArrayAtom '.' ArrayAtom
   | ArrayAtom '::' SimpleName
   | NewSpace
   | 'Θ' | '#'
   | 'α' | 'ω' | 'αα' | 'ωω'
   | '(' ArrayExpr ')

ArrayNaming → NameStruct '←' ArrayExpr

NameStruct → Name | '(' NameStruct+ ')

Strand → ArrayAtom ArrayAtom+

Indexing → ArrayAtom Subscript

Mutation → Name Subscript '←' ArrayExpr

Subscript → [' ArrayExpr? ('; ArrayExpr?)* ']

NewSpace → ['[[' ExprList ']]'

**Example**

Example of ExprList:

```
A 3 ⋄ {⍵} ⋄ ⍠
```

Example of Brk:

```
A 2+3
```

Example of OperExpr:

```
A ⍴-1
```

Example of ArrayExpr:

```
A 2+3
```

Example of FuncExpr:

```
A ⍬ System.DateTime.Now
```

Example of OperExpr:

```
A :If 1 ⋄ 6 ⋄ :End
```

Example of ArrayAtom:

```
A vec
```

Example of Strand:

```
A #.(1+2)
```

Example of Indexing:

```
A #:.(1+2)
```

Example of Mutation:

```
A vec[1 2]
```

Example of Subscript:

```
A vec[1 2]
```

Example of NewSpace:

```
A vec[1 2]
```

A NewSpace creates and returns a new namespace in which the enclosed expression
A has been evaluated. See the Examples below.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function → FuncAtom &lt;br&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FuncExpr → Function</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FuncAtom → Name</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FuncDerived → LeftOperand MopExpr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FuncNaming → Name '←' FuncExpr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LeftOperand → ArrayAtom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RightOperand → ArrayAtom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2-train → ArrayAtom Function &lt;br&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3-train → ArrayAtom Function Function &lt;br&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Nilad → DefinedNilad</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NiladNaming → Name '+' '' Nilad</td>
<td></td>
</tr>
</tbody>
</table>
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

A <2
A *
A limit + ≡
A ((*)+)
A operator self-reference
A inverse + ≡-1
A ..
A ≡-1 A [M0]
A ..
A {⍺⍺}⍵→⍺⍺ ⍵
A #.{{f}⍵→f ⍵}
A TBD Example needed.
A B
A {⍺{f g}⍵→f ⍺ g ⍵} A (atop)
A #.B
A TBD Example needed.
Rule | Example
---|---
PrimFunc → '+' | '-' | '×' | '÷' | '*' | '⍟' | '⌊' | '⌈' | '|' | '!' | '○' | '<' | '≤' | '=' | '>' | '≥' | '≠' | '∧' | '∨' | '⍲' | '⍱' | '~' | '?' | '⍴' | '∊' | '↑' | '↓' | '⍪' | '⍋' | '⍒' | '⌽' | '⍉' | '⊖' | '∪' | '⊃' | '⊂' | '∩' | '⍎' | '⍕' | '⌷' | '⊣' | '⊢' | '≡' | '≢' | '¤' | '$' | Slash

PrimMop → '¨' | '⍨' | Slash

PrimDop → '⍠' | '∘' | '⍣' | '⍢'

Slash → '/' | '\' | '⍷'

A Where:

A Squad-indexing as in Dyalog
A ☆ New space. Analogous to ⎕NEW in Dyalog
A $ Converts between "string" and 'character vector'
A
A ô Rank
A ⍪ Variant
A × Power
A ⍳ Dual/Under

Guard → ArrayExpr ': ' Expr

Name → SimpleName
| '[]'
| ArrayAtom '. ' SimpleName

SimpleName → IdentifierChar (IdentifierChar | DecimalDigit)*

IdentifierChar: The Unicode Standard defines which chars may be used in identifiers.
Function and Operator Definition

DefinedFunc → '{' (Larg? Rarg '→')? ExprList '}' A [DF0]

DefinedMop → '{' Larg? '{' Land '}' Rarg '→' ExprList '}' A [DM0]

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

DefinedNilad → '{' '.' '→' ExprList '}'

Larg → NameStruct | '⍺'
Rarg → NameStruct | '⍵'
Land → NameStruct | '⍺⍺'
Rand → NameStruct | '⍵⍵'
Control Structures

\[ \text{CtrlStruct} \rightarrow \text{IfStruct} \mid \text{WhileStruct} \mid \text{RepeatStruct} \mid \text{ForStruct} \mid \text{SelectStruct} \mid \text{WithStruct} \mid \text{Label} \mid \text{GoToStruct} \mid \text{ThrowStruct} \mid \text{TryCatchStruct} \mid \text{ContinueStruct} \mid \text{LeaveStruct} \mid \text{PropertyStruct} \mid \text{ReturnStruct} \]

\[ \text{IfStruct} \rightarrow ':If' \text{ExprBrk} \\text{AndOr?’} \text{ExprListBrk} \]
\[ ( ':ElseIf' \text{ExprBrk} \text{AndOr?’} \text{ExprListBrk} )^* \]
\[ ( ':Else' \text{ExprListBrk} )? \]
\[ ( ':End' | ':EndIf' ) \]

\[ \text{WhileStruct} \rightarrow ':While' \text{ExprBrk} \\text{AndOr?’} \text{ExprListBrk} \]
\[ ( ':End' | ':EndWhile' | Until ) \]

\[ \text{RepeatStruct} \rightarrow ':Repeat' \text{ExprBrk} \]
\[ ':End' | ':EndRepeat' | Until ) \]

\[ \text{Until} \rightarrow ':Until' \text{ExprBrk} \text{AndOr?’} \]
\[ ( ':And' | ':Or' )^+ \]
\[ ( ':End' | ':EndFor' ) \]

\[ \text{AndOr?’} \rightarrow ( ':AndIf' \text{ExprBrk} )^+ \]
\[ ( ':OrIf' \text{ExprBrk} )^+ \]

\[ \text{ForStruct} \rightarrow ':For' \text{NameStruct} ( ':In' | ':InEach' ) \text{ExprBrk} \]
\[ ':End' | ':EndFor' ) \]

\[ \text{SelectStruct} \rightarrow ':Select' \text{ExprBrk} \]
\[ ( ( ':Case' | ':CaseList' ) \text{ExprBrk} \text{ExprListBrk} )^* \]
\[ ( ':Else' \text{ExprListBrk} )? \]
\[ ( ':End' | ':EndSelect' ) \]

\[ \text{WithStruct} \rightarrow ':With' \text{ExprBrk} \]
\[ ':End' | ':EndWith' ) \]
TryCatchStruct → ':Try' Brk
  ExprList Brk
  ( ':Catch' ArrayExpr Brk 
    ExprList Brk
  )* 
  ( ':Finally' Brk
    ExprList Brk
  )*? 
  ( ':End' | ':EndTryCatch' )

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

ThrowStruct → ':Throw' ArrayExpr
  A :Throw RankError

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

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

GoToStruct → ':GoTo' Name Brk
  A :GoTo Loop [G0]

DecimalNumber → '-'? DecimalDigit* ('. ' DecimalDigit+)? (Exponent)?
  A 0 99 ~0.41E7

Exponent → ('E' | 'e') '-'? DecimalDigit+

DecimalDigit → '0'..'9'

CharLiteral → "" UnicodeCharacter* ""
  A 1≡≡'charvec'

String → "" UnicodeCharacter* ""
  A 0≡≡"string"

EndLine → Comment? ''? '
'

Comment → 'A' PrintableChar*
  A 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
α(f g)ω → f α g ω

(A g)ω → A g ω
α(A g)ω → SyntaxError

3-trains

(f g h)ω → ( f ω)g( h ω)
α(f g h)ω → (α f ω)g(α h ω)

(A g h)ω → A g( h ω)
α(A g h)ω → A g(α h ω)

(f B h)ω → f(B h ω)
α(f B h)ω → α f(B h ω)
Examples

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

```
myop ← {(larg0 larg1) {f g} rarg →
  ⍝ Dyadic operator
  lucky←'.,+2?'
  ⍝ nilad
  Else    ⋄ g rarg ⋄ :EndIf
  [ [  ⍝ new space returned, in which:
    a←(lucky : f larg0 ⋄ g rarg
    x←[ [[a+ω]] ]a
    y←[[
        n+f g lucky
    ]] (x,y y x)←y x
    ]])
```

AA In the AA lines above, it seems intuitive (to JMS) that names (lucky, AA 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 ω
{ (ll x){⍺⍺}(y rr) →
  x=STOP : ll x (y rr)
  X←⍺⍺ x
  (ll X)y ⋄ rr
}                           ⍝ Turing machine
```

NewSpace structure contains temporary variables:

```
[[
  A 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) \div (a \ b) \div \[[[\] \] \] \] \text{('')}\]
\[b. a \ a. b \equiv a \ b\]

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.

\[\text{'}A\text{'}{\\\n\begin{array}{l}
\text{var} \sim \text{⍺} \ \text{⋄ fun} \sim , \ \text{⋄ spc} \sim [[[c} \ \sim \omega]] \text{A names local to function}
\text{\[\text{'}B\text{'}{, :With spc} \text{A within space,}
\text{\var fun spc.c} \text{A all outer names are visible}
\text{:EndWith}
\text{\[\text{'}C\text{'}{, [[[y} \sim \text{var fun spc.c} \text{A all outer names are visible}}
\text{\]}y\}
\text{\[\text{'}H\text{'}{, spc.}{\\\n\begin{array}{l}
\text{\var} \sim \text{'}O\text{'} \text{A space-global name shadows outer.}
\text{\var fun spc.n} \text{A other outer names are visible}}
\text{\]}\}
\text{'}T\text{'}\]
\text{BAT}
\text{CAT}
\text{HOT}\]

Lexical Scope Considerations

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

\[\text{which}\sim{\\\n\begin{array}{l}
\text{scope} \sim \text{'}lexical'\text{A which "scope" is in view?}
\{\text{scope} \sim \text{'}dynamic'\text{A operand evaluated in definition-,}
\text{\sub} \sim \omega \text{A rather than reference-context.}}
\}
\text{which'scope}'\text{lexical}\]

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

\[\text{which}\sim{\\\n\begin{array}{l}
\text{scope} \sim \text{'}lexical'\text{A which "scope" is in view?}
\{\text{scope}\{\text{⍺⍺} \sim \omega \text{A operand evaluated in definition-,}
\text{\text{⍺⍺}} \sim \text{'}dynamic'\text{A rather than reference-context.}}
\}
\text{which'scope}'\text{lexical}\]
Note that, unlike Dyalog, this convention extends to primitive operands:

```plaintext
which←{  
  scope←'lexical'  A where is ⍪ evaluated?  
  ⍪{(⍺⍺)⍵ ↄ  
    scope←'dynamic'  A operand evaluated in definition-,  
      ⍺⍺ ⍵  A rather than in reference- context.  
  )⍵  
  which'scope'  A this differs from Dyalog !!
limit
```

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

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

a: fn  
b: ns  
c: ns  
d: fn  
e: ns  
f: ns
```

**Lexical scope inside :With control structure**

```plaintext
ns←{  
  a →  
    :With ⍬''  
      b←a A outer name “a” is visible.  
  :End
}θ

ns.b  
θ

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 ~ A θ θ
- ArrayAtom Subscript ~ θ[
- '.' ArrayAtom ~ θ(θ)
- Dop RightOperand ~ ,*4

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

- Dyalog:
  - A B.C ~ A A(B.C)
  - A B[] ~ A (A B)[]
  - #.B[] ~ A (#.B)[]
  - ',*θ ~ A ,*θ[θ)
  - ',#.t ~ A ,*(#.t)
  - ',*θ[θ) ~ A (,*θ)[θ]


The following table shows for each class of token:

- DOT Space-referencing dot. For example the ‘.’ in: #.t
- DX “Dotted” array. For example, the ‘#.’ in #.V
- A Array.
- MF Function bound with left argument. ~ A [MFO]
- 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.

<table>
<thead>
<tr>
<th>DOT</th>
<th>A</th>
<th>DX</th>
<th>MF</th>
<th>F</th>
<th>MOP</th>
<th>DOP</th>
<th>IDX</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>8</td>
<td>DX</td>
<td>6</td>
<td>A</td>
<td>2</td>
<td>MF</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>DX</td>
<td>7</td>
<td>A</td>
<td>7</td>
<td>MF</td>
<td>7</td>
<td>F</td>
<td>7</td>
<td>MOP</td>
</tr>
<tr>
<td>MF</td>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>A</td>
<td></td>
<td>4</td>
<td>F</td>
<td></td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>MOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOP</td>
<td>5</td>
<td>MOP</td>
<td>5</td>
<td>MOP</td>
<td>5</td>
<td>MOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.


[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 rightOperand 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 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