APL# Core language

Presented at Dyalog 2011 on 3rd October 2011

The core language described in this document is partially implemented in APL#/Silverlight 0.3.0.627 that will be made available to attendees at Dyalog 2011. The interpreter is built primarily for testing the language design, and as such has been engineered for internal flexibility rather than performance. As a result the interpreter should only be used on small arrays of data (up to a couple of hundred items), when experimenting with the language. It is expected that a new version of the interpreter will be made available approximately every 3-6 months during the next year. These updates will implement more language features, and have better tool support (for example debugging support), as well as improving the performance of the parts of the language that have stabilised.

Please note that some of the language features described in this document are not available in the 0.3.0.627 build. These will be noted in the relevant sections of the document. The final page provides a summary of the features that are implemented in 0.3.0.627.

If you would like to be informed when an updated interpreter is available, or have any thoughts on the design outlined here, please let us know by emailing aplsharp@dyalog.com.

Introduction

Dyalog APL is already a fast and efficient APL system which we will continue to support and improve for the foreseeable future, so one question which is often asked is why are we working on APL# as well, rather than concentrating purely on Dyalog APL?

To answer that question we can consider the different strengths of Dyalog APL and APL#. Firstly Dyalog APL is a full rich APL dialect with a very fast implementation and comprehensive tool support (code editing, debugging, GUI frameworks etc.) that runs on a number of platforms. It is a proven APL platform for high performance computing, and fast program development. As a result it is an ideal tool for APL centred development.

APL# is being designed as a lightweight APL engine that can be used on different development platforms in systems where very tight integration with other programming languages or frameworks is needed. We aim to provide good tool support for code writing & debugging, but expect that the tools provided by 3rd parties (such as platform vendors) would be used, in conjunction with APL# and other languages, for specialised application areas such as user interface design.

So whilst APL# will be a different APL dialect to Dyalog APL they will be highly compatible at the source code level, share tooling and be able to interoperate at the binary level both when run on the same platform (such as Windows with .Net) and across platforms, such as an APL# Silverlight application calling a Dyalog APL web service hosted on an AIX server.

Using the two complementary APL systems APL developers will be able to write systems that run in many more places, work on a range of device devices, and developer platforms, and interoperate with code written in other languages, better than would be possible with either tool on it's own.

The rest of this document describes the current design of the core APL# language. It does not cover how the core language maps to particular platforms, such as Silverlight or WinRT (Windows 8 Metro). Nor does it cover tooling.

Some of the major aims of the APL# language design are:

- Keep the language small, simple and composable.
    - Don't introduce any more concepts than absolutely necessary for clear concise general purpose code.
    - Keep the expressiveness of traditional APL dialects.
    - The language should be simple to learn for new users.
- Familiar APL.
    - Writing APL# code should be familiar to anyone with previous APL experience.
- Flexible and platform agnostic.
    - The core language must assume as little as possible about the underlying platform.

Modern software development platforms

Before looking at the details of the APL# language, it is important to consider the types of platform that it is being designed for. An increasing trend in computer platform design, particularly on Microsoft platforms, is to provide platform APIs (Application Programming Interfaces) in a format that can be consumed equally easily, and naturally, by many different computer languages. When writing applications for these platforms developers can choose the language and tools that they can best express themselves or their problem in, yet still have access to all of the richness of the platform. It then becomes easy to split an application into different sections and use different languages for each area, with the different parts of the application able to communicate easily. For example the user interface could be written in one language, and the business logic or domain model in another.

This trend comes in response to traditional architectures, such as Win32 or UNIX where the platform APIs favour one language, usually C, and different language implementations tend to develop libraries of functions that can't be shared between languages without a lot of complexity.

APL# is being designed as a modern, simple and elegant APL dialect that can be used on language agnostic platforms such as .NET and WinRT without paying a heavy semantic or performance penalty when dealing with the underlying platforms, or parts of an application written in other languages.

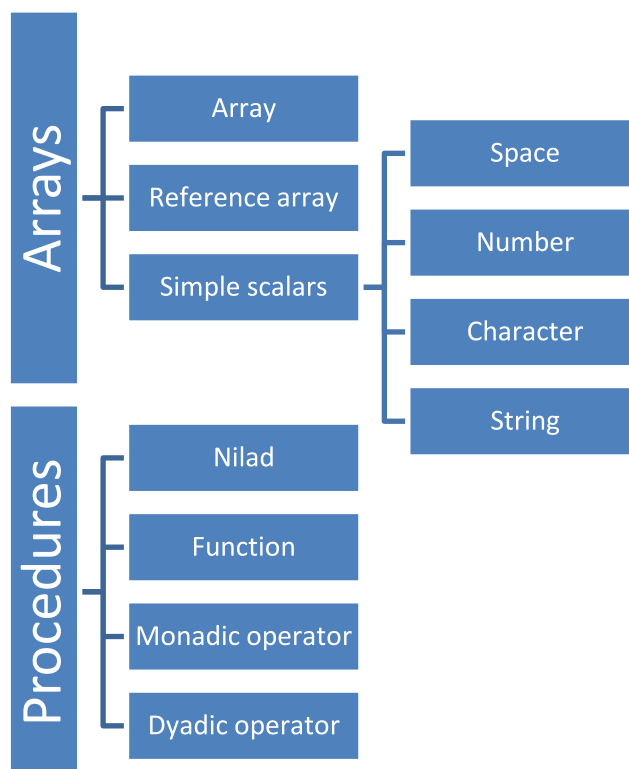
Core APL#

The core APL# language consists of the type system and syntax that are used in APL# applications. These will be implemented on various flavours of APL# - for example APL#/Silverlight will implement the core language on Silverlight, and provide interoperability with other code on the platform, including the Silverlight .net framework. Similarly APL#/WinRT will implement the core language and provide interoperability with the WinRT (Metro) runtime. Code written using only the features in the Core APL# language will run unchanged in any APL# interpreter, and in many cases will be portable to Dyalog APL with little or no change.

APL# Type system

Types are runtime objects that represent things that can be named and manipulated by APL# programs. On multi-language platforms they can also be passed directly to, and used by, code written in the other languages.

All of types in the APL# type system are either arrays or procedures. In any particular implementation of APL# on a platform, the native types of that platform will be exposed by presenting the native types as the APL# type where there is a very strong correspondence between the two (for example a .net string and an APL# string), or by adding platform specific types as simple scalar array types.



Arrays

Arrays are data structures that contain named members, members can be arrays or procedures. All arrays contain at least 3 members, Shape Items and TypicalItem.

The Shape member is a vector (one dimensional array) whose length determines the number of axis of the array, and whose items determine the length of each of the dimensions of the array. Once the array has been created the Shape is read only.

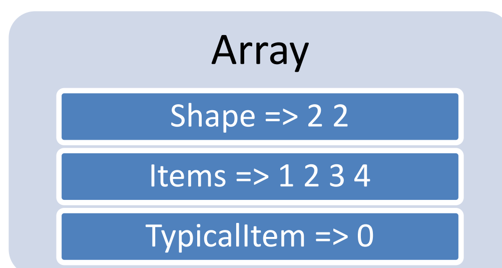
The Items member is a vector containing the items in the various cells of the array, in ravel order.

The TypicalItem is an array that provides a default array item for primitive functions such as ↑ (Take) that may require an element to pad the result array that matches the type of element of the array.

For example the array created by the statement:

```
2 2 ρ 1  2  3  4
```

Would look like:



The following sections describe the various types of array in the APL# type system.

Array

An array is a general purpose container for other arrays. An array will only have the default 3 members Shape, Items and TypicalItem as described above.

Arrays are passed by value. In other words when an array is passed to a function, or named, the function or name gets a copy of the array. For example:

```
      A ← 1  2  3
      B ← A
      A[0] ← 99
      A
99  2  3
      B
1  2  3
```

Note: Arrays are the familiar arrays used in other APL systems. This describes the semantics of how arrays work. Implementations of APL# will be written to avoid this copying when not necessary, as is the case for all commercial APL systems.

Reference array

It is sometimes useful to be able to share an updatable array between various parts of a program so that its contents may be modified from various places. In Dyalog this can be accomplished using memory mapped files.

Reference arrays have exactly the same members as arrays, the only difference is that naming, or passing the array to a function, does not cause a copy of the array to be created.

Note: The syntax shown here is provisional and likely to change.

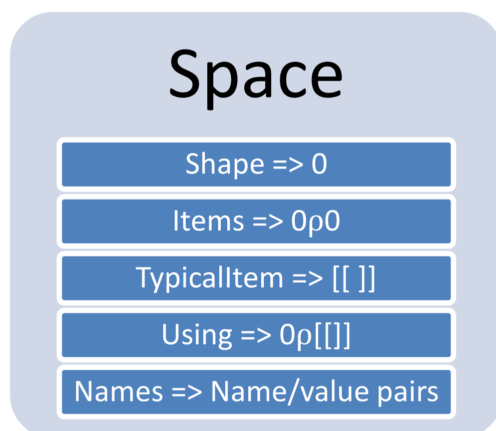
Reference arrays can only be created by using the new Reference primitive (see later) all other primitive functions return standard arrays, regardless of whether they were called with a reference array or a normal array.

```
      A ← ⍈ 1 2 3       ⍝ A names a reference array containing 1 2 3
      B ← A             ⍝ B names the same reference array as A
      C ← ⊢ A           ⍝ C names a new array containing 1 2 3
      A[0] ← 99
      A
99 2 3
      B
99 2 3
      C
1 2 3
```

Note: reference arrays are not implemented in build 0.3.0.627.

Space

A space is a simple scalar (Rank 0, Depth 0 array) that contains names. The primitive constant # refers to a single shared (or global) space that can be used to share named arrays and procedures between different parts of the code.



Note: Unlike Dyalog namespaces APL# spaces don't have an explicit parent/child hierarchy.

```
      S2←[[        ⍝ Create a new space
           x←10    ⍝ set local x in the space
           y←20    ⍝ set local y in the space
      ]]
      S2.(x + y) ⍝ Evaluate an expression in a space
30
```

All APL# code runs within a space. If a script, or the tools that run it, don't specify a particular space then the global space is used.

In addition to the standard members Space contains a Using member which is an array of spaces used to resolve names if a name cannot be found in the space itself.

Assigning an array or procedure to a name that is unused within a space creates a new local value. When a name is referred to it is resolved by looking in the following locations:

1. The space
2. Each item of the Using member (see below) in turn
3. The global space - #

For example:

```
      S1←[[x←99]]
      S2←[[ ]]
      S2`.Using←S1
      S2.(x)
99
```

Note: `. is used to access the members of an array, rather than it's elements. For more information see the Escape section later in this document.

Note: A syntax to refer to the current space will be provided, but this has not yet been agreed.

Note: The `. syntax is not implemented in build 0.3.0.627.

Number

A number is a simple scalar that represents any real number. It is expected that complex numbers will be added in a future version of APL#.

The internal representation or representations of numbers are implementation dependent, but implementations must give the illusion that only one, number representation is being used, at least as effectively as traditional APL systems.

Therefore any implementation that uses binary floating point numbers internally must use the standard APL tolerant comparison operations, as defined in the draft Extended ISO APL standard.

Character

A character is a simple scalar that represents any single Unicode character.

String

A string is a simple scalar that represents an ordered sequence of Unicode characters.

Error

It is likely that the error handling constructs will require a specific simple scalar type to hold the information about, and identity of errors. If so this will be defined along with the error handling code. (See later.)

Note: Error is not implemented in build 0.3.0.627.

Procedures

Nilad

A Nilad represents some code that takes no arguments and is evaluated as soon as it appears, unless it is escaped.

```
      x←10
      y←{.→x)  ⍝ The procedure is evaluated, and the result is named
y
      z←`{.→x} ⍝ The procedure is named z and evaluation is deferred
      x←20
      y
10
      z     ⍝ The procedure referred to by z is now evaluated
20
```

Note: The procedure declaration and ` syntax will be detailed later in this document.

Function

A Function represents a block of code that takes one or two arrays and returns an array.

Note: Currently all APL# functions are ambivalent, that is any function `F` can be called either monadically:

```
      F 1
```

Or dyadically:

```
      1 F 2
```

This is currently under review, and monadic user defined functions may be added back into the language. See user defined procedures later.

Monadic operator

A monadic operator represents a block of code that takes one function or array as an operand and returns a function.

Dyadic operator

A dyadic operator represents a block of code that takes one function or array as an operand and returns a function.

Note: operators always return functions, never Nilads. However this may change if monadic user defined functions are added back into the language. See user defined functions later in this document.

APL# Syntax

An APL# script is a Unicode string containing valid APL# syntax. Typically this is either from a text file, or code entered into a 6-space prompt. Scripts can be evaluated in any space, but the default is to evaluate them in the global space.

Scripts can contain the following syntax:

Names

Any sequence of the characters A-Z,a-z,0-9,_,- that starts with either _ or a letter.

Constants

Numeric and character constants are declared using the same syntax as Dyalog APL. Additionally string constants can be declared using double quotes.

```
C←'Hello, world'    ⍝ Character vector
S←"Hello, world"    ⍝ String
N←99                ⍝ Number
A←1.0 ¯2.5e100 1.2  ⍝ Numeric array
```

Primitive constants

Currently ⍬ (a 0 length numeric vector) and # (the global space) are the only available primitive constants.

Note: The constant # is not implemented in build 0.3.0.627, as it is not useful without #.x and #`.x which are also not implemented.

Primitive functions and operators

APL# will provide a full set of APL primitive functions as described in the table below. Unless otherwise noted these will match the behaviour of Dyalog APL. The implemented column indicates the primitives that are implemented in the conference version of APL#.

Note: APL# does not support variable index origin, comparison tolerance or migration level. When comparing APL# to Dyalog it can be considered to have fixed ⎕IO=0, ⎕CT=1e¯14 and ⎕ML=3.

	Monadic	Dyadic	Implemented in build 0.3.0.627
+	Conjugate	Plus	✓
−	Negate	Minus	✓
×	Direction	Multiply	✓
÷	Reciprocal	Divide	✓
=		Equal	✓
≠		Not equal	✓
<		Less than	✓
≤		Less than or equal	✓
>		Greater than	✓
≥		Greater than or equal	✓
⊣	Same	Left	✓
⊢	Same	Right	✓
≡	Depth	Match	✓
≢		Not match	✓
~	Not		✓
⍳	Index generator	Index of	✓
∨		Or/GCD	✓
∧		And/LCM	✓
⍱		Nor	✓
⍲		Nand	✓
⍴	Shape	Reshape	✓
⍨	Commute/Duplicate		✓
|	Magnitude	Residue	✓
⌊	Floor	Minimum	✓
⌈	Ceiling	Maximum	✓
¨	Each		✓
,	Ravel	Join	✓
$	String		
?	Roll	Deal	
⍕	Format		
⍎	Execute		
¤	New space	New	
⍠	Variant		
↑	First	Take	✓
↓	Split	Drop	✓
*	Exponential	Power	✓
⍟	Natural logarithm	Logarithm	✓
⊃	Mix	Pick	✓

	Monadic	Dyadic	Implemented in build 0.3.0.627
!	Factorial	Binomial	✓
○	PiTimes	Circle	✓
⍪	Table	Join first	✓
/	Reduce (operator)	Replicate (function)	✓
⌿	Reduce first (operator)	Replicate first (function)	✓
\	Scan (operator)	Expand(function)	✓
⍀	Scan first (operator)	Expand first (function)	✓
.		Inner product	✓
∘.g		Outer product	✓
∘		Compose	✓
⍋	Character grade up	Grade up	
⍒	Character grade down	Grade down	
⌽	Reverse	Rotate	✓
⊖	Reverse first	Rotate first	✓
⍉	Transpose	Transpose	✓
⌹	Matrix inverse	Matrix division	
∪	Unique		✓
∊	Enlist	Member of	✓
⍣		Power operator	✓
⍤		Rank operator	✓
⊥		Base value	✓
⊤		Representation	✓
⊂	Enclose	Partition	✓
⌷		Squad indexing	✓
⍈	Reference		

Note: $ is a primitive function for converting character arrays to strings. Format can be used to convert a string to a character vector.

Note: The rank operator is modelled on that in J.

Note: The reference function is provisional. See the reference array section above for more information.

Name scope separator

APL# uses . syntax, similar to Dyalog, to evaluate code within a space. When given a space on its left and either a parenthesised expression, a name or an assignment on it's right, the . evaluates the expression or name within the space. For example:

```
      x←10
      s ← [[ ]]
      s.x←x       ⍝ Evaluate x in the outer space, and
                  ⍝ assign it's value to the name x within s
      s.(y←x)     ⍝ Evaluate (y←x) within s
      s.y         ⍝ Resolve the name y within s
10
```

As with Dyalog APL the . is applied pervasively to any array on its left. For example:

```
      spaces← [[x←10]] [[x←20]] [[x←30]]
      spaces.x
10 20 30
```

Note: The name scope separator is not implemented in build 0.3.0.627.

Escape

The escape character ` can be used to temporarily prevent evaluation of a procedure, or to access the members of an array using . for example:

```
      x←10
      n←`{x}                 ⍝ Name a nilad by delaying its evaluation
      x←20
      n                      ⍝ Call the nilad
20
      a←1 2 3
      a`.Shape               ⍝ Call the Shape member of the array
3
      m←2 2⍴1 2 3 4
      m`.Items               ⍝ Call the items member of the array
1 2 3 4
      M
1 2
3 4
```

Note: Escape is not implemented in build 0.3.0.627.

Naming

APL# supports assignment to individual names or parenthesized name strands.

```
     a ← 1 2 3          ⍝ Name the array 1 2 3
     (a b c)←1 2 3      ⍝ a←1 ⋄ b←2 ⋄ c←3
     x y z←1 2 3        ⍝ names the array z, value error on y
Value error: y
```

Selective specification (see Dyalog APL) will be added later, but is not currently part of the APL# language.

Indexing

```
     (a←1 2 3)[0 2]     ⍝ Retrieve items from an array
1 3
     a←1 2 3
     a[2]←9            ⍝ Replace elements in a named array
     a
1 2 9
     m←2 2⍴1 2 3 4
     m[0;0]←9          ⍝ Replace elements in a matrix
     m
9 2
3 4
```

Expression seperators

Within a script APL# expressions are separated by any of the following:

- `⍝comment \n`
- `\n`
- `\r`
- `⋄`

Where `\n` is a newline character and `\r` is a carriage return character.

Precedence specification

Parts of APL# expressions may be parenthesised to control binding. For example:

```
      1 - 2 - 3
2
      (1 - 2) - 3
¯4
```

Note that any part of an expression may be parenthesised, in common with Dyalog APL.

However APL# also allows expression lists to be parenthesised, including guards and control structures. Parenthesised expression lists may only return arrays, and unnamed values within the parenthesised expression list are not echoed to the console. For example:

```
      x←20
      b←10 + (a←10    ⍝ Parentheses do not create a new scope,
                      ⍝ so a is created in the containing scope
              99      ⍝ Not echoed to the console
              a+x)    ⍝ return a+x from the parentheses
     a
10
   b
40
```

Space specification

A new space can be defined by wrapping an expression list (possibly empty) in double brackets:

```
   X←10
   S←[[Y←X]]
   S.Y
10
```

When a new space is created it’s Using statement is set to the containing space, to bring the containing space into scope. After the initialisation expression list has been evaluated, if the space’s Using statement has not been assigned to then the Using statement is cleared. This gives the initialisation code for the new space access to members of the enclosing space during construction, and a chance to capture a reference to the enclosing space if required.

Note: space specification is not implemented in build 0.3.0.627, as it is of little use without “a.b” which is also not implemented.

Control structures

APL# will contain a set of control flow control structures that behave as per Dyalog APL, including:

:If :While :Repeat :For :Select :Else :ElseIf :AndIf :OrIf :Until :Case :CaseList :End :Leave :Continue

Note: Control structures are not implemented in build 0.3.0.627.

Goto

APL# does not support the branching arrow, or implied line numbers. Instead the :Label and :Goto control structures are used. Note that the names of labels are completely separate from the names of local values.

```
      F←{ x←⍵
          :Label start
          x←x+1
          x > 20 :  :Goto end
          :Goto start
          :Label end
          x
        }
      F 2
21
```

Note: Unlike Dyalog D-Fns APL# functions can contain a mixture of guards and control structures.
Note: Unlike traditional APL "`X⊃start end`" is not valid as it uses label names in an expression. Instead this should be written as "`X : :Goto end ⋄ :Goto start`".

Note: :Goto and :Label not implemented in build 0.3.0.627.

Guards

```
      Condition : Expression
```

Guards are a lightweight control flow syntax. First the Condition is evaluated, and if the result of that matches 1 then the Expression is evaluated and it's result immediately returned as the result of the enclosing function or script.

```
      F←{⍵>10 : "Big" ⋄ "Little"}
      F 2
Little
      F 100
Big
```

Error handling

Error handling is likely to be performed using control structures and an Error type with an exception based error handling model. But this has not been designed yet, and therefore the final mechanism may be completely different. It is certain, however, that the design team will aim to consolidate and simplify the functionality of all the error handling mechanisms in Dyalog APL.

Note: error handling is not implemented in build 0.3.0.627.

Procedure specification

User defined procedures (or A-Fns) are an evolution of D-Fns, combining the best of D-Fns and Traditional Functions from Dyalog APL.

A procedure definition begins with a { optionally followed by a header, then an expression list and finally a } . Like D-Fns all A-Fns are declared anonymously, and may be named using the naming arrow ←.

Unlike D-Fns A-Fns return the result of the last expression in their body, rather than that of the first unnamed expression:

```
    f←{α ⋄ ω}
    1 f 2      ⍝ Dyalog
1
    1 f 2      ⍝ APL#
2
```

When a user procedure is called a new space is created to run the body of the procedure in. This space is initialised with various special read only names that can be used to access the arguments and operands of the procedure, the procedure itself, and the space in which the procedure was defined. Also the Using member of the space is set to the space in which the procedure was defined.

Additionally if any arguments or operands of the procedure are named in the header these are initialised with the values passed in to the procedure, as illustrated by the operator below

```
⍝ User operator as defined in an APL# script:
{a {f g} (b c) →
  ((a b) f c) g (c f (a b))
}

{a {f g} (b c) →
⍝========= Initialised first
  α             ⍝ left argument
  ω             ⍝ right argument
  αα            ⍝ left operand
  ωω            ⍝ right operand
  ∇             ⍝ This (derived) function
  ∇∇            ⍝ This operator
  ##            ⍝ Lexical parent space
  ∇`.Using←##   ⍝ Initial using gives lexical name resolution.
⍝========== Then named arguments and operands are initialised
  a← α
  f←αα
  g←ωω
  (b c)←ω
⍝========== Now the procedure body is evaluated

  ((a b) f c) g (c f (a b))
}
```

Note: Like traditional APLs when defining an operator, you provide the definition of the derived function.

Note: If the header is omitted it defaults to {⍺ ⍵ → …} Therefore a header must be provided to define either an operator or a nilad, unlike D-Fns.

The type of procedure specified by the user defined procedure is determined by the optional header. Valid headers are of the following forms where a f g and b are each either names or name strands, or the appropriate special name ⍺ ⍵ ⍺⍺ or ⍵⍵.

```
{.→...}         ⍝ Nilad
{b→...}         ⍝ function
{a b→...}       ⍝ function
{{f} →...}      ⍝ Monadic operator, derived function
{{f} b→...}     ⍝ Monadic operator, derived function
{a {f} b→...}   ⍝ Monadic operator, derived function
{{f} →...}      ⍝ Dyadic operator, derived function
{{f} b→...}     ⍝ Dyadic operator, derived function
{a {f} b→...}   ⍝ Dyadic operator, derived function
```

When a function is called monadicaly ⍺ is set to the default value of ⊣. For a nilad both ⍺ and ⍵ default to ⊣.

If separate Monadic and Dyadic functions are reinstated in the APL# core language then the possible headers will become:

```
{→...}            ⍝ Nilad
{b→...}           ⍝ Monadic function
{a b→...}         ⍝ Dyadic function
{{f} →...}        ⍝ Monadic operator, derived nilad
{{f} b→...}       ⍝ Monadic operator, derived monadic function
{a {f} b→...}     ⍝ Monadic operator, derived dyadic function
{{f} →...}        ⍝ Dyadic operator, derived nilad
{{f} b→...}       ⍝ Dyadic operator, derived monadic function
{a {f} b→...}     ⍝ Dyadic operator, derived dyadic function
```

For more information on A-Fns, including the reasoning behind using ⊣ as the default value for a missing argument see the "Unifying T-Fns and D-Fns in APL#" paper presented at the APL 2010 conference in Berlin, and available from http://www.aplsharp.com/ .

Binding strength

APL# follows the same binding rules as Dyalog APL, with the exception of strand assignment, which binds stronger than arrays in APL#, but not in Dyalog:

```
      (a b c) ← 1 2 3  ⍝ Strand assignment in Both Dyalog & APL#
      a b c ← 10 20 30

⍝ In Dyalog
      a
10
      b
20
      c
30


⍝ In APL#
      a
1
      b
2
      c
10 20 30
```

For more details on APL# binding strengths see AplSharpDescription.pdf .

Summary of features implemented in V0.3.0.627

Syntax

- Constants
    - Numeric
    - Character
    - String
- Names
- Naming
- Procedure definition
    - Nilads
    - Ambivalent functions
    - Monadic operators
    - Dyadic operators
- Primitives – most functions and operators
  + − × ÷ = ≠ < ≤ > ≥ ⊣ ⊢ ≡ ≢ ~ ⍳
  ∨ ∧ ⍱ ⍲ ⍴ ⍨ | ⌊ ⌈ ¨ , ↑ ↓ * ⍟ ⊃
  ! ○ ⍪ / ⌿ \ ⍀ . ∘.g ∘ ⌽ ⊖ ⍉ ∪ ∊
  ⍣ ⍤ ⊥ ⊤ ⊂ ⌷
- Fixed values instead of system variables, equivalent to
  (⎕IO ⎕CT ⎕ML)←1 1e¯14 3
- ⍬
- Indexing
- Indexed assignment
- Statement separators, including comments
- Parenthesised expressions and expression lists
- Guards