Combinatory Logic and Combinators in Array Languages

Conor Hoekstra
conorhoekstra@gmail.com
Toronto Metropolitan University
Toronto, ON, Canada

Abstract

The array language paradigm began in the 1960s when Ken Iverson created APL. After spending over 30 years working on APL, he moved on to his second array language J, a successor to APL which embeds a significant subset of combinatory logic in it. This paper will look at the existence of combinators in the modern array languages Dyalog APL, J and BQN and how the support for them differs between these languages. The paper will also discuss how combinators can be a powerful language feature as they exist in modern array languages.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: combinatory logic, combinators, array languages, apl, j, bqn

ACM Reference Format:
Conor Hoekstra. 2022. Combinatory Logic and Combinators in Array Languages. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '22), June 13, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3520306.3534504

1 Introduction

The goal of this paper is to provide a brief history of both array languages and combinatory logic and then to more importantly show how a significant subset of combinatory logic exists in modern array languages in the form of combinators and why this is an extremely useful language feature.

Section 2 will briefly introduce array languages and how they differ from other programming languages. Section 3 will cover a brief history of array languages. Section 4 will cover a brief history of combinatory logic focusing on the papers and literature that first introduced individual combinators. Section 5 will enumerate the hierarchy of combinators and how certain combinators are specializations of other combinators. Section 6 will discuss the evolution of combinators in array languages and specifically how "trains" differ in modern array languages. Section 7 will enumerate each of the combinators from combinatory logic as they exist in each of Dyalog APL, J and BQN. Section 8 will discuss what makes combinators in modern array languages so powerful and expressive. Section 9 will list several additional examples. Finally, in Section 10 a summary of what has been discussed and shown in this paper will be provided.

ARRAY '22, June 13, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9269-3/22/06…$15.00
https://doi.org/10.1145/3520306.3534504

2 A Brief Introduction to Array Languages

The three main array languages that will be discussed in this paper are: Dyalog APL (1983), J (1990) and BQN (2020). These languages are notably different from other programming languages because APL and BQN use Unicode symbols for primitives and J uses ASCII symbols, digraphs, and trigraphs. Many have asserted that this makes the languages unreadable but that is the equivalent of saying Chinese is unreadable just because it doesn't use the Latin alphabet. As long as you have the requisite knowledge to read the symbols, not only is it extremely readable but it can also be extremely expressive and powerful. For example, the following snippet of APL code can be used to generate the first 10 odd numbers:

```
      ⍝ numbers from 1 to 10
      ⍳10
1 2 3 4 5 6 7 8 9 10
      ⍝ multiply by 2
      2×⍳10
2 4 6 8 10 12 14 16 18 20
      ⍝ add negative 1
      ¯1+2×⍳10
1 3 5 7 9 11 13 15 17 19
```

Other examples that demonstrate the expressivity and power of APL are examples using outer product. The following snippet of APL code can be used to create a multiplication table:

```
      ⍝ multiply outer product
      ∘.×⍨⍳10
 1  2  3  4  5  6  7  8  9  10
 2  4  6  8 10 12 14 16 18  20
 3  6  9 12 15 18 21 24 27  30
 4  8 12 16 20 24 28 32 36  40
 5 10 15 20 25 30 35 40 45  50
 6 12 18 24 30 36 42 48 54  60
 7 14 21 28 35 42 49 56 63  70
 8 16 24 32 40 48 56 64 72  80
 9 18 27 36 45 54 63 72 81  90
10 20 30 40 50 60 70 80 90 100
```

And finally, one of my personal favorite examples is a solution to the "maximum consecutive ones" problem.

```
      vec ← 1 1 0 1 1 1 0 0 0 1
      ⍝ split (partition) on zeroes
      ⊆⍨vec
┌─────┬─────┬─┐
│1 1│1 1 1│1│
└─────┴─────┴─┘
      ⍝ size of each sublist
      ≢¨⊆⍨vec
2 3 1
      ⍝ max reduction
      ⌈/≢¨⊆⍨vec
3
```

The final maximum consecutive ones APL solution can be translated for those who don't read APL:

```
reduce(max, map(length, W(partition, vec)))
```

As demonstrated by these three short examples, array languages provide the user with a combination of expressivity and terseness not found in many other programming languages. Language features common across array languages that enable this are:

- A rich set of fundamental builtin primitives in the form of:
  - functions
  - operators (higher order functions)
  - combinators
- The fact that spelling primitives with one glyph or two ASCII characters enables a *flexibility* not found in any other paradigm
- Rank polymorphism (primitives work on different rank arrays)

3 Brief History of Array Languages

Kenneth Iverson started working on Iverson Notation in 1957 while at Harvard. He later published his book A Programming Language in 1962[22]. This booked used Iverson Notation as a notation for expressing algorithms. It wasn't until 1966 when Iverson Notation would be "upgraded" to a programming language and rebranded as APL (which is the acronym for A Programming Language).

There would be many variants of APL, starting with APL\360 which was used to design the IBM/360. Other variants developed over the years would include APL PLUS, IBM APL2, Sharp APL, Dyalog APL and GNU APL to name a few. For a comprehensive list of APL variants/dialects see the APL Wiki[4][6].

After evolving for over 30 years, APL would have two main child languages: J and K. Kenneth Iverson worked on J along with Roger Hui, one of Iverson's protégés. Iverson's other protégé, Arthur Whitney would go on to work on K. K4 (one of the dialects of K) would be sold to First Derivatives for $100 million and be "wordified" (words would be added to go along with the ASCII symbols). The "wordified" version was called Q. It is also worth noting that before Whitney went off to create K, he was involved in the creation of J when he wrote the J Incunabulum[20]. For a full list of K dialects, visit the APL wiki[3].

Two of the most recent array languages that have been developed are I and BQN created by Marshall Lochbaum. Note that I is not actively developed and was an experimental language that Lochbaum does not recommend for use. BQN on the other hand is very actively developed and is a more *functional* array language with the richest set of combinators of any of APL, J or K. See Table 1 for a list of significant array languages and when they were introduced.

Table 1. Array languages timeline.

Language	Year
APL	1966
Dyalog APL 1.0	1983
J	1990
K	1994
Q	2003
I	2012
BQN	2020
Dyalog APL 18.0	2020

Array languages influencing other array languages is very common. See Figure 1 for a dependency graph of how array languages influenced each other.

For a more detailed array language influence graph, visit the APL Wiki[2].

4 Brief History of Combinatory Logic

4.1 Moses Schönfinkel, 1924

The first paper to introduce combinators was by Moses Schönfinkel in his 1924 "Über die Bausteine der mathematischen Logik"[29]. The German title translates to "On the building blocks of mathematical logic"[30]. In this paper, Schönfinkel introduces the I, C, S, T and Z combinators.

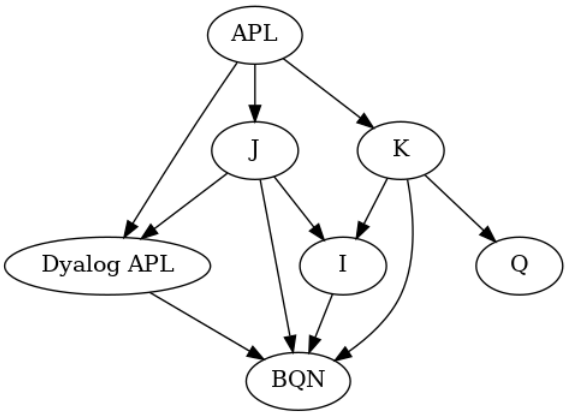

Figure 1. Array language influences.

The C, Z and T combinators would later be respectively renamed to K, B and C by Haskell Curry[12]. For an in depth background on the origins of Schönfinkel, see "Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel"[36].

4.2 Haskell Curry, 1929

Haskell Curry would independently discover combinators in the late 1920s. He came across Schönfinkel's paper before publishing his first paper on the topic in 1929, "An Analysis of Logical Substitution"[12]. In this paper, Curry reintroduces S, K, I, B and C and goes on to also introduce W. Note that although the word "combinatory" shows up twice in this paper, the terms *combinator* and *combinatory logic* do not.

4.3 Haskell Curry, 1930

In Curry's dissertation "Grundlagen der Kombinatorischen Logik"[13] (The Foundations of Combinatory Logic), he would introduce the B_n combinator (and therefore implicitly the B_1 combinator). It was also in Curry's dissertation that the terms *combinator* and *combinatory logic* would be introduced.

4.4 Haskell Curry, 1931

After publishing his dissertation[13] in 1930, Curry would quickly publish "The Universal Quantifier in Combinatory Logic"[14] in 1931 in which he would introduce the Φ combinator and more generally the Φ_n combinator as well as the Ψ combinator. In this paper he would refer to the Φ combinator as the *formalizing combinator*.

4.5 Haskell Curry, 1931-1948

Following 1931, Haskell Curry would go on to publish many papers on the topic of combinatory logic, sometimes referred to as the *theory of combinators*, including: "Some Additions to the Theory of Combinators"[15] in 1932, "A Revision of the Fundamental Rules of Combinatory Logic"[16] in 1941, and "The Combinatory Foundations of Mathematical Logic"[17] in 1942. John Rosser would remark in his review[28] of Curry's "The Combinatory Foundations of Mathematical Logic"[17] that "it does present an accurate and lucid account of the major accomplishments of combinatory logic to date, together with an estimate of what may be expected of the subject in the future." That "future" would come several years later when Curry published "A Simplification of the Theory of Combinators"[18] in 1948 which was a summary and simplification of his work on combinatory logic up until that point.

4.6 Haskell Curry, 1958

Haskell Curry's research would culminate in his 1958 seminal text Combinatory Logic: Volume I[19]. In this text he would go on to name the five combinators I, C, W, B and K the **elementary combinators** and give each of them special names (see Table 2).

Table 2. The elementary combinators.

Combinator	Elementary Name
I	Elementary Identificator
C	Elementary Permutator
W	Elementary Duplicator
B	Elementary Compositor
K	Elementary Cancellator

4.7 David Turner, 1979

David Turner worked on three different programming languages (SASL[32], KRC[34] and Miranda[35]) that were built on top of combinators and used combinator graph reduction compilation techniques. In his research on combinators and graph reduction, he would reintroduce the Φ combinator under a different name: the S′ combinator[33].

4.8 Raymond Smullyan, 1985

Raymond Smullyan would extend the list of existing combinators in his logic puzzle book[31]. Among the combinators introduced were the D (Dove), D_2 (Dovekie), E (Eagle) and Ê (Bald Eagle) combinators.

5 Combinator Specializations

The list of combinators that will be covered in section 7 and the corresponding lambda expressions are listed in Table 3.

Table 3. Combinators and lambda expressions.

Combinator	Lambda Expression
I	λa.a
K	λab.a
W	λab.abb
C	λabc.acb
B	λabc.a(bc)
S	λabc.ac(bc)
D	λabcd.ab(cd)
B_1	λabcd.a(bcd)
Ψ	λabcd.a(bc)(bd)
Φ	λabcd.a(bd)(cd)
D_2	λabcde.a(bd)(ce)
E	λabcde.ab(cde)
Φ_1	λabcde.a(bde)(cde)
Ê	λabcdefg.a(bde)(cfg)

Many combinators can just be thought of as specializations of other combinators. The most obvious specializations are:

- Φ is a specialization of D_2 with $d = e$
- D is a specialization of D_2 with $b = I$
- Ψ is a specialization of D_2 with $b = c$
- S is a specialization of Φ with $b = I$
- S is a specialization of D with $b = d$
- W is a specialization of S with $b = I$
- W is a specialization of Ψ with $b = I$ and $c = d$

A visualization of the specializations/dependencies of the above combinators is shown in Figure 2.

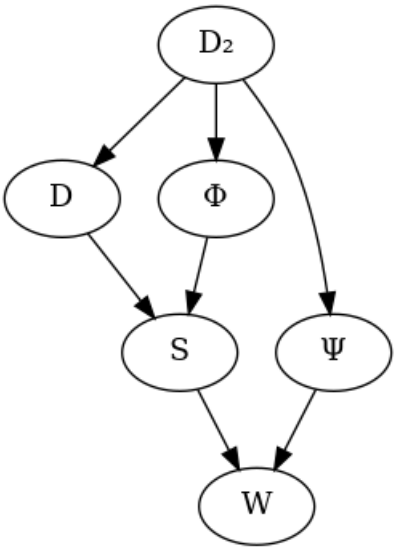

Figure 2. D_2 combinator hierarchy.

The E-family of combinators also are specializations of each other:

- E is a specialization of Ê with $b = K$
- Φ_1 is a specialization of Ê with $d = f$ and $e = g$

6 Evolution of Combinatory Logic in Array Languages

6.1 Phrasal Forms, 1989

In 1989, Kenneth Iverson and Eugene McDonnell wrote a paper entitled Phrasal Forms[26]. It was in this paper where they showed how the Φ and S combinators could be spelled in APL. They would be introduced into J as core parts of language. Unfortunately, save for this 1989 paper, there are no other documented references to combinatory logic and combinators in array language literature. Kenneth Iverson renamed combinators to *trains*, renamed the S combinator in J to the *hook* and renamed the Φ combinator in J to the *fork*. These are also referred to more generally as 2-trains and 3-trains. They are also sometimes referred to as "invisible modifiers"[27] because they are formed through juxtaposition of functions as opposed to other combinators that are spelled with glyphs.

In Phrasal Forms, there is a reference to John Backus' 1978 Turing Award paper[8] where it mentions that the *construction form* is equivalent to a fork in APL or J where the binary operation is catenate (`,`). For instance: `[+,-]` in FP is functionally the same as `(+,-)` in APL or J. It is also interesting to note the similarity between the title of the paper *Phrasal Forms* and the name John Backus used for his *combining forms* as it seems like Iverson took inspiration from Backus here. Note that *Phrasal Forms* was not just the name of the paper, but the initial name for what would end up being called *trains*. The fork from APL and J as well as *infix notation*[9] from FP are both the Φ and the Φ_1 combinator.

6.2 Hook Conjunction?, 2006

It is definitely worth noting the difference in choice of two-trains in APL, BQN and J.

Table 4. 2 and 3-trains in APL, BQN and J.

Year	Language	2-Train	3-Train
1990	J	S and D	Φ and Φ_1
2014	Dyalog APL	B and B_1	Φ and Φ_1
2020	BQN	B and B_1	Φ and Φ_1

When J was created in 1990, that was the first time an array language added combinators in the form of trains. It would be 24 years before Dyalog APL added trains in 2014 when Dyalog APL 14.0 was released. The reason Dyalog APL chose to replace the monadic and dyadic 2-trains with B and B_1 is because the implementer of J, Roger Hui, considered it a mistake as he points out in his 2006 essay "Hook Conjunction?"[21]. The main reason was that using S as the 2-train resulted in special parsing rules. BQN would also go on to choose B and B_1 for the monadic and dyadic 2-trains.

6.3 Dyalog APL and BQN

Dyalog APL added combinators in the following years[5] and versions seen in Table 5.

Table 5. History of combinators in Dyalog APL.

Year	Version	Combinator	Spelling
1983	1.0	B, D, C	∘⍨
2003	10.0	W	⍨
2013	13.0	K	⊢
2014	14.0	B, B_1, Φ, $Φ_1$	trains
2020	18.0	B, B_1, Ψ, K	⍥⍤⍨

BQN is still pre-1.0 but has added the richest set of combinators of any array language and arguably any programming language. For a comprehensive list of combinators and their bird names, see Marshall Lochbaum's *BQN Birds*[25].

7 Combinators in APL, BQN & J

In this section, APL specifically refers to Dyalog APL version 18.0. Furthermore, see Table 6 for definitions of the terms used for arity (number of function arguments) in this section.

Table 6. Arity terms.

Arity	Greek Term	Latin Term
1	Monadic	Unary
2	Dyadic	Binary

It is more common to use the terms *monadic* and *dyadic* when referring to functions when working with array languages. However, in this paper *unary* and *binary* are used when referring to functions in general and *monadic* and *dyadic* are used when referring to specific APL, BQN or J functions or trains.

Lastly, in the Python code in this section, `f, g, h` refer to functions and `x, y, z` refer to arguments.

7.1 I

The I combinator is known as *identity* in most languages. It is `id` in Haskell. It is a unary function that returns the argument that is passed in. It was first introduced in Schönfinkel 1924[29]. It was introduced in Curry and Feys[19] as the **Elementary Identificator**.

A potential implementation in Python:

```
def i(x):
    return x
```

7.2 K

The K combinator is known under many names in different languages. It is `const` in Haskell. It is a binary function that returns the first argument that is passed in. It was first introduced in Schönfinkel 1924[29]. Note that Schönfinkel originally called this combinator C but it was later renamed to K by Curry[12]. It was introduced in Curry and Feys[19] as the **Elementary Cancellator**. Smullyan nicknamed it Kestrel in his logic puzzle book To Mock a Mockingbird[31].

Table 7. I combinator.

Language	Name	Symbol
APL	Same	⊢
APL	Constant	⍨
BQN	Identity	⊢
BQN	Constant	˙
J	Same	]

Table 8. K combinator.

Language	Name	Symbol
APL	Right	⊢
BQN	Right	⊢
J	Right	]

A potential implementation in Python:

```
def k(x, y):
    return x
```

7.3 S

The S combinator is known under a couple of different names. It is known as `ap` or `<*>` in Haskell, as the *hook* in both J and I[23] and as *monadic after* in BQN. It is the 2-train in J. It is the composition of a binary and unary function such that the binary function takes two arguments: the right one after having the unary function applied to it and the left one as the original argument. It was first introduced in Schönfinkel 1924[29]. Smullyan nicknamed it Starling[31].

Table 9. S combinator.

Language	Name	Symbol
APL	-	-
BQN	(Monadic) After	⍥
J	(Monadic) Hook	2-train

Although APL does not have the S combinator, the same effect can be achieved by combining the D and W combinators: `f∘g⍨`.

Potential implementations in Python:

```
# Includes application
def s(f, g, x):
    return f(x, g(x))
```

```
# Excludes application
def s(f, g):
    return lambda x: f(x, g(x))
```

7.4 B

The B combinator is what most people think of when they hear "composition." It is called `compose` in Racket, `comp` in Clojure, in Julia, `(.)` in Haskell and exists in many other languages. It is the monadic 2-train in both APL and BQN. It is the composition of two unary functions, applying one function after the other. It was first introduced in Schönfinkel 1924[29]. Note that Schönfinkel originally called this combinator Z but Curry would later rename it to B[12]. It was introduced in Curry and Feys[19] as the **Elementary Compositor**. Smullyan nicknamed it the Bluebird[31]. It has so many forms in each of the array languages because many of the dyadic operators define the monadic version to be the B combinator.

Table 10. B combinator.

Language	Name	Symbol
APL	(Monadic) Beside	∘
APL	(Monadic) Atop	⍤
APL	(Monadic) Over	⍥
APL	(Monadic) Atop	2-train
BQN	(Monadic) Atop	∘
BQN	(Monadic) Over	○
BQN	(Monadic) Atop	2-train
J	(Monadic) At	`@:`
J	(Monadic) Appose	`&:`

Potential implementations in Python:

```
# Includes application
def b(f, g, x):
    return f(g(x))

# Excludes application
def b(f, g):
    return lambda x: f(g(x))
```

7.5 B_1

The B_1 combinator does not exist in most languages. It can be found as `(.:)` in Haskell in the `Data.Composition` library. It is the dyadic 2-train in both APL and BQN. It is the composition of one binary function and one unary function, where the binary function is first applied to two arguments and then the unary function is applied to its result. It was first introduced in Curry 1948[18]. Smullyan nicknamed it the Blackbird[31].

Table 11. B_1 combinator.

Language	Name	Symbol
APL	(Dyadic) Atop	⍤
APL	(Dyadic) Atop	2-train
BQN	(Dyadic) Atop	∘
BQN	(Dyadic) Atop	2-train
J	(Dyadic) At	`@:`

Potential implementations in Python:

```
# Includes application
def b(f, g, x, y):
    return f(g(x, y))

# Excludes application
def b(f, g):
    return lambda x, y: f(g(x, y))
```

7.6 C

The C combinator is known as `flip` in Haskell and is similar to `SWAP` in FORTH. It is a unary function that takes a binary function as its argument and "flips" the order the two arguments are passed. Note that for commutative functions, the C combinator effectively has no effect (such as with addition or multiplication). It was first introduced in Schönfinkel 1924[29]. Note that Schönfinkel originally called this combinator T but Curry would later rename it to C[12]. It was introduced in Curry and Feys[19] as the **Elementary Permutator**. Smullyan nicknamed it the Cardinal[31].

Table 12. C combinator.

Language	Name	Symbol
APL	Commute	⍨
BQN	Swap	~
J	Passive	~

Potential implementations in Python:

```
# Includes application
def c(f, x, y):
    return f(y, x)

# Excludes application
def c(f):
    return lambda x, y: f(y, x)
```

7.7 W

The W combinator is known as `join` in Haskell and is similar to `DUP` in FORTH. It takes a binary function and turns it into a unary function by duplicating/using the single argument as both arguments for the original binary function. It was first introduced in Curry 1929[12]. It was introduced in Curry and Feys[19] as the **Elementary Duplicator**. Smullyan nicknamed it the Warbler[31].

Table 13. W combinator.

Language	Name	Symbol
APL	Commute	⍨
BQN	Self	~
J	Reflex	~

Potential implementations in Python:

```
# Includes application
def w(f, x):
    return f(x, x)

# Excludes application
def w(f):
    return lambda x: f(x, x)
```

7.8 Ψ

The Ψ combinator can be found as on in Haskell in the `Data.Function` library. It is the composition of a binary function and a unary function where the two arguments each have the unary function applied to them, and each result is then passed to the binary function as the first and second argument. It was first introduced in Curry 1958[19].

Table 14. Ψ combinator.

Language	Name	Symbol
APL	Over	⍥
BQN	Over	○
J	Appose	&:

Potential implementations in Python:

```
# Includes application
def psi(f, g, x, y):
    return f(g(x), g(y))

# Excludes application
def psi(f, g):
    return lambda x, y: f(g(x), g(y))
```

7.9 Φ (S′)

The Φ combinator is known as "infix notation" in FP[9] and FL[7] when called monadically and `liftA2` in Haskell. It is the monadic 3-train in both APL, BQN and J. It is the composition of two unary functions and one binary function. It is similar to the Ψ combinator but instead of applying the same unary function to two different arguments, you apply two different unary functions to the same argument. It was first introduced in Curry 1931[14] as the Φ combinator. It was later reintroduced with a different name, the S′ combinator in Turner 1979[33]. Smullyan nicknamed it the Phoenix[31].

Table 15. Φ (S′) combinator.

Language	Name	Symbol
APL	(Monadic) Fork	3-train
BQN	(Monadic) Fork	3-train
J	(Monadic) Fork	3-train

Potential implementations in Python:

```
# Includes application
def phi(f, g, h, x):
    return g(f(x), h(x))

# Excludes application
def phi(f, g, h):
    return lambda x: g(f(x), h(x))
```

7.10 Φ_1

The Φ_1 combinator is known as "infix notation" in FP[9] and FL[7] when called dydically. It is the dyadic 3-train in APL, BQN and J. It is the composition of three binary functions. It can be thought of as the Φ combinator but with the two unary functions replaced with binary functions. It was first introduced in Curry 1931[14]. In fact, both the Φ and Φ_1 combinators are specializations of the Φ_n combinator. Smullyan did not name the Φ_1 combinator, so this paper is introducing the bird name "Pheasant" due to it also starting with "ph" (similar to B and B_1 having the names Bluebird and Blackbird).

Table 16. Φ_1 combinator.

Language	Name	Symbol
APL	(Dyadic) Fork	3-train
BQN	(Dyadic) Fork	3-train
J	(Dyadic) Fork	3-train

Potential implementations in Python:

```
# Includes application
def e_(f, g, h, x, y):
    return g(f(x, y), h(x, y))

# Excludes application
def e_(f, g, h):
    return lambda x, y:
        g(f(x, y), h(x, y))
```

7.11 D

The D combinator is unique to array languages. It is the composition of one binary function and one unary function where the unary function is applied to the right argument and then passed along with the other argument to the binary function. It was first introduced in Smullyan 1985[31]. Smullyan nicknamed it the Dove[31].

Table 17. D combinator.

Language	Name	Symbol
APL	(Dyadic) Beside	∘
BQN	(Dyadic) After	⟜

Potential implementations in Python:

```
# Includes application
def d(f, g, x, y):
    return f(x, g(y))

# Excludes application
def d(f, g):
    return lambda x, y: f(x, g(y))
```

7.12 D_2

The D_2 combinator doesn't explicitly exist in any of the array languages in the form of a train or glyph. However, it can be spelled in BQN due to the fact that both the *hook* from J (called *after* in BQN) and the *backHook* from I[23] (called *before* in BQN) both exist. Furthermore, the D_2 combinator can also be spelled in Extended Dyalog APL[10] which implements *reverse compose* (the equivalent of I's *backHook* and BQN's *before*).

Table 18. D_2 combinator.

Language	D_2
BQN	`a⟜b⟜c`
Extended Dyalog APL	`a⍥b∘c`

In the above spellings, *a* and *c* are unary functions and *b* is a binary function. Smullyan nicknamed the D_2 combinator the Dovekie[31].

Potential implementations in Python:

```
# Includes application
def d2(f, g, h, x, y):
    return g(f(x), h(y))

# Excludes application
def d2(f, g, h):
    return lambda x, y: g(f(x), h(y))
```

8 The Power of Combinators in Array Languages

Combinators in array languages are powerful for several reasons:

1. They have higher precedence than function application
2. They are spelled with one glyph or through juxtaposition
3. Some of them (trains) are "invisible"
4. They are uniformly designed
5. There are a rich set of them
6. They are builtin

To understand this better, a contrast between another (non-array) "combinator" language will be drawn.

8.1 Combinator Support: Haskell vs Array Languages

Arguably, one of the best combinator languages outside of array languages is Haskell. A list of combinators in Haskell is shown in Table 19.

Table 19. Combinators in Haskell.

	Name	Library
I	`id`	Prelude
K	`const`	Prelude
W	`join`	Control.Monad
C	`flip`	Prelude
B	`(.)`	Prelude
S	`(<*>)` / `ap`	Control.Applicative
B_1	`(.:)`	Data.Composition
Ψ	`on`	Data.Function
Φ	`liftA2`	Control.Applicative
A	`($)`	Prelude

However, combinators in Haskell fail on reasons (1), (2), (3), (4) and (6) when compared with the six reasons why modern array language combinators are so powerful. Below is an examination of each reason, in reverse order.

8.1.1 Builtin Combinators. It matters less in Haskell that the combinators are not builtin, but the fact that they are scattered in various forms across four different libraries (five if Prelude is included) makes them a hassle to import. That being said, a single library providing all combinators could easily be provided.

8.1.2 Rich Set of Combinators. Arguably, Haskell has a "rich" set of combinators. Compared to array languages however, Haskell has a relatively "less rich" set of combinators. Missing from the list of combinators in Haskell are D, D_2 and Φ_1. That being said, once again it would be easy to provide the missing combinators in a different or one of the existing libraries.

8.1.3 Uniformly Designed Combinators. The nonuniform defining of certain combinators as infix operators, certain combinators as functions and one as both leads to a suboptimal design. As with the previous two reasons, this could be remedied with a standalone library that redefined the combinators in a uniform manner.

8.1.4 Invisible Combinators. Also known as "invisible modifiers"[27], invisible combinators exist in array languages in the form of trains (see Table 4). This is a language level feature making it impossible to replicate in Haskell

or any other language through a library. As the B and Φ combinators are ubiquitous in programming, it leads to an extremely convenient way of composing functions because there is zero syntax that needs to be added for the composition to occur.

8.1.5 Combinator Spelling. The spelling of combinators in Haskell makes the "tax" of using them higher than in array languages. For instance, Table 20 shows the additional code required in order to "commute" a minus operation.

Table 20. Commuting minus.

Language	Minus	Commuted Minus
Haskell	`-`	`flip (-)`
APL	`-`	`-⍨`

Haskell's C combinator isn't terrible, but the minus operation went from one character to eight whereas in APL it only went from 1 to 2. This is a non-trivial difference that is even worse when it comes to other combinators such as Φ in Haskell.

8.1.6 Combinator Precedence. Of all the five reasons why combinators are so powerful in array languages, one stands out as by far the most important: the fact that combinators have higher precedence than function application. In Haskell, infix operators have a specified precedence. However, the maximum precedence of an infix operator is 9, meaning it can never be higher than the precedence of function application which is 10. Table 21 shows the precedence of the combinators that take the form of infix operators.

Table 21. Combinator precedence in Haskell.

	Name	Precedence
	Function Application	10
B	`(.)`	9
B_1	`(.:)`	8
S	`(<*>)`	4
A	`($)`	0

It makes complete sense that combinators should have higher precedence than function application because combinators can be viewed as "function application modifiers." In fact, BQN's combinators sit in a category of primitives called *modifiers*. The documentation even states that combinators control the application of functions.

This leads to more expressive code in that composition of functions using combinators doesn't need to be parenthesized. In Haskell, you encounter the following:

```haskell
import Control.Applicative (<*>)

> (==) <*> reverse "tacocat" -- error
-- parentheses required
> ((==) <*> reverse) "tacocat" -- success
```

Compare this to array languages where the parentheses are not needed. Below is the equivalent expression in BQN.

```
≡∘⊸Φ "tacocat" # success
```

8.2 A Simple Example

A simple problem will be solved in both Haskell and BQN to illustrate how richer, builtin combinators with higher precedence than function application can lead to more expressive code.

The problem is to generate a list of absolute differences between all pairs of numbers generated from a list of numbers (with replacement).

8.2.1 Haskell.

```haskell
import Data.List.HT (outerProduct)
import Control.Monad (join)
import Data.Composition ((.:))
import Data.List.Unique (sortUniq)

allDiffs
   = sortUniq
   . concat
   . join (outerProduct (abs .: (-)))

allDiffs [1,-5,3,-8,6]
-- Results in:
-- [0,2,3,5,6,8,9,11,14]
```

Multiple parentheses are required to get the correct order of function application, not to mention the difference between the prefix `join` and the infix `.:` / `.` and that they each come from separate libraries.

8.2.2 BQN.

```
    vec ← 1‿¯5‿3‿¯8‿6

    # Difference of all pairs
    # with C
    (-⌜˜) vec
┌─
╵  0  6  ¯2  9  ¯5
  ¯6  0  ¯8  3 ¯11
   2  8   0 11  ¯3
  ¯9 ¯3 ¯11  0 ¯14
   5 11   3 14   0
                  ┘
    # Absolute value with B1
    (|∘-⌜˜) vec
┌─
╵ 0  6  2  9  5
  6  0  8  3 11
  2  8  0 11  3
```

```
  9  3 11  0 14
  5 11  3 14  0
              ⌟
    # Deshape
    (↩|∘-⌜˜) vec
⟨ 0 6 2 9 5 6 0 8 3 11 ...

    # Deduplicate with B
    (⍷∘↩|∘-⌜˜) vec
⟨ 0 6 2 9 5 8 3 11 14 ⟩

    AllDiffs ← ⍷∘↩|∘-⌜˜
```

In the BQN code, no parentheses are required. The ˜ and ∘ have higher precedence than function application and no libraries are required.

9 Examples

Each of the examples are run with one or two test cases in only APL, but the output is the same for all of the APL, BQN and J functions.

9.1 average

```
// Translation (Tacit)
avg = phi(sum, divide, length)
// Translation (Explicit)
avg(x) = divide(sum(x), length(x))
```

```
⍝ APL
avg ← +/÷≢  ⍝ Φ
```

```
# BQN
Avg ← +´÷≠  # Φ
```

```
NB. J
avg =. +/%# NB. Φ
```

```
⍝ Test
      avg 1 2 3 4
2.5
```

9.2 plusOrMinus

```
// Translation (Tacit)
plusOrMinus = phi1(plus, concat, minus)
// Translation (Explicit)
plusOrMinus(x, y) =
    concat(plus(x,y), minus(x,y))
```

```
⍝ APL
plusOrMinus ← +,-  ⍝ Φ1
```

```
# BQN
PlusOrMinus ← +⋈-  # Φ1
```

```
NB. J
plusOrMinus =. +,- NB. Φ1
```

```
⍝ Test
      10 plusOrMinus 5
15 5
```

9.3 absoluteDifference

```
// Translation (Tacit)
absDiff = b1(abs, minus)
// Translation (Explicit)
absDiff(x, y) = abs(minus(x,y))
```

```
⍝ APL
absDiff ← |-    ⍝ B1
absDiff ← |⍤-   ⍝ B1
```

```
# BQN
AbsDiff ← |-    # B1
AbsDiff ← |∘-   # B1
```

```
NB. J
absDiff =. |@:- NB. B1
```

```
⍝ Tests
      10 absDiff 7
3
      7 absDiff 10
3
```

9.4 isPalindrome

```
// Translation (Tacit)
isPalindrome = phi(reverse, equals, i)
isPalindrome = s(equals, reverse)
// Translation (Explicit)
isPalindrome(x) = equals(reverse(x), i(x))
isPalindrome(x) = equals(x, reverse(x))
```

```
⍝ APL
isPalindrome ← ⍞≡⊢      ⍝ Φ I
isPalindrome ← ≡∘⍞⍨     ⍝ D W
```

```
# BQN
IsPalindrome ← ⍞≡⊢      # Φ I
IsPalindrome ← ≡⊸⍞      # S
IsPalindrome ← ≡⊸⍞˜     # D W
```

```
NB. J
isPalindrome =. |.-:]   NB. Φ I
isPalindrome =. -:|.    NB. S
isPalindrome =. (-:|.)~ NB. D W
```

```
⍝ Tests
      isPalindrome 'tacocat'
1
      isPalindrome 'tacodog'
0
```

9.5 isAnagram

```
// Translation (Tacit)
isAnagram = psi(equals, sort)
// Translation (Explicit)
isAnagram(x, y) = equals(sort(x), sort(y))
```

```
⍝ APL
sort      ← ⊂⍤⍋⌷⊢     ⍝ B1 Φ I
isAnagram ← ≡⍤sort    ⍝ Ψ
```

```
# BQN
IsAnagram ← ≡O˜        # Ψ
# J
sort      =. /:~       NB. W
isAnagram =. -:&:sort  NB. Ψ
A Tests
      'owls' isAnagram 'slow'
1
      'cats' isAnagram 'dogs'
0
```

9.6 isDisjoint

```
// Translation (Tacit)
isDisjoint = e(nil, equals, intersect)
// Translation (Explicit)
isDisjoint(x, y) = equals(nil, intersect(x, y))
A APL
isDisjoint ← ⍬≡∩     A E
# BQN
IsDisjoint ← ⟨⟩≡∊/¬ # E Φ1
IsDisjoint ← ¬·∨´∊  # B B1
NB. J
isDisjoint =. (i.0)-:e.#[ NB. E Φ1
A Tests
      1 2 isDisjoint 3 4 5
1
      2 3 isDisjoint 3 4 5
0
```

9.7 isPrefixOf

```
// Translation (Tacit)
isPrefixOf = b1(first, find)
// Translation (Explicit)
isPrefixOf(x, y) = first(find(x, y))
A APL
isPrefixOf ← ⊃⍷  A B1
isPrefixOf ← ⊃⍤⍷ A B1
# BQN
IsPrefixOf ← ⊑⍷  # B1
IsPrefixOf ← ⊑⍤⍷ # B1
NB. J
isPrefixOf =. {.@:E. NB. B1
A Tests
      'cat' isPrefixOf 'catch'
1
      'dog' isPrefixOf 'catch'
0
```

10 Summary

In this paper, a brief history of both array languages and combinatory logic was provided. The relationships between combinators were examined and captured visually by the combinator hierarchy in Figure 2. Combinators as they exist in the modern array languages Dyalog APL, J and BQN were enumerated and explained and differences were highlighted where they existed. Finally, the power and expressivity of combinators in modern array languages was examined.

Acknowledgments

Thank you to the following individuals who have provided reviews and meaningful feedback: Adám Brudzewsky, Computer Programmer at Dyalog Limited and creator of Extended Dyalog APL[10] and APLCart[11]; Marshall Lochbaum, creator of the I[23] and BQN[24] programming languages; Dr. Troels Henriksen, assistant professor at DIKU and creator of the Futhark[1] programming language; and Dr. David Mason, Chair of Computer Science at Toronto Metropolitan University.

References

[1] 2013. The Futhark Programming Language. https://futhark-lang.org/
[2] 2021. Family tree of array languages - APL Wiki. https://aplwiki.com/wiki/Family_tree_of_array_languages
[3] 2021. K - APL Wiki. https://aplwiki.com/wiki/K
[4] 2021. Timeline of APL dialects - APL Wiki. https://aplwiki.com/wiki/Timeline_of_APL_dialects
[5] 2022. Dyalog APL - APL Wiki. https://aplwiki.com/wiki/Dyalog_APL
[6] 2022. List of language developers - APL Wiki. https://aplwiki.com/wiki/List_of_language_developers
[7] Alexander Aiken, John H Williams, and Edward L Wimmers. 1994. *The FL project: The design of a functional language.* Citeseer.
[8] John Backus. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* 21, 8 (1978), 613–641. Publisher: ACM New York, NY, USA.
[9] John Backus. 1985. From function level semantics to program transformation and optimization. In *Colloquium on Trees in Algebra and Programming*. Springer, 60–91.
[10] Adám Brudzewsky. 2018. dyalog-apl-extended: Dyalog APL Extended. https://github.com/abrudz/dyalog-apl-extended
[11] Adám Brudzewsky. 2019. APLcart - Find your way in APL. https://aplcart.info/
[12] Haskell B Curry. 1929. An analysis of logical substitution. *American journal of mathematics* 51, 3 (1929), 363–384. Publisher: JSTOR.
[13] Haskell Brooks Curry. 1930. Grundlagen der kombinatorischen Logik. *American journal of mathematics* 52, 4 (1930), 789–834. Publisher: JSTOR.
[14] Haskell Brooks Curry. 1931. The universal quantifier in combinatory logic. *Annals of Mathematics* (1931), 154–180. Publisher: JSTOR.
[15] Haskell Brooks Curry. 1932. Some additions to the theory of combinators. *American Journal of Mathematics* 54, 3 (1932), 551–558. Publisher: JSTOR.
[16] Haskell B. Curry. 1941. A Revision of the Fundamental Rules of Combinatory Logic. *The Journal of Symbolic Logic* 6, 2 (1941), 41–53. http://www.jstor.org/stable/2266655 Publisher: Association for Symbolic Logic.
[17] Haskell B Curry. 1942. The combinatory foundations of mathematical logic. *The Journal of Symbolic Logic* 7, 2 (1942), 49–64. Publisher: Cambridge University Press.
[18] Haskell B Curry. 1948. A simplification of the theory of combinators. *Synthese* (1948), 391–399. Publisher: JSTOR.

[19] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam.

[20] Roger Kwok Wah Hui. 1992. *An Implementation of J*. Iverson Software.

[21] Roger Kwok Wah Hui. 2006. Essays/Hook Conjunction? - J Wiki. https://code.jsoftware.com/wiki/Essays/Hook_Conjunction%3F

[22] Kenneth E. Iverson. 1962. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA. http://portal.acm.org/citation.cfm?id=SERIES11430.1098666

[23] Marshall Lochbaum. 2012. ILanguage: An interpreter for a J-inspired language. https://github.com/mlochbaum/ILanguage

[24] Marshall Lochbaum. 2020. BQN: finally, an APL for your flying saucer. https://mlochbaum.github.io/BQN/

[25] Marshall Lochbaum. 2022. BQN for birdwatchers. https://mlochbaum.github.io/BQN/doc/birds.html

[26] Eugene E McDonnell and Kenneth E Iverson. 1989. Phrasal forms. In *Conference proceedings on APL as a tool of thought*. 197–199.

[27] Henry Rich. 2002. J for C Programmers. (2002).

[28] Barkley Rosser. 1943. Review of: The Combinatory Foundations of Mathematical Logic by Haskell B. Curry. *The Journal of Symbolic Logic* 8, 1 (1943), 31–31. http://www.jstor.org/stable/2267984 Publisher: Association for Symbolic Logic.

[29] Moses Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Mathematische annalen* 92, 3 (1924), 305–316. Publisher: Springer.

[30] Moses Schönfinkel. 1967. On the building blocks of mathematical logic. *From Frege to Gödel* (1967), 355–366. Publisher: Harvard University Press Cambridge MA.

[31] Raymond M Smullyan. 2000. *To Mock a Mockingbird: and other logic puzzles including an amazing adventure in combinatory logic*. Oxford University Press, USA.

[32] DA Turner. 1976. SASL Language Manual. *Andrews University, Fife, Scotland* (1976).

[33] David A Turner. 1979. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic* 44, 2 (1979), 267–270. Publisher: Cambridge University Press.

[34] David A Turner. 1982. Recursion equations as a programming language. In *A List of Successes That Can Change the World*. Springer, 459–478.

[35] David A Turner. 1985. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 1–16.

[36] Stephen Wolfram. 2021. Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel. *arXiv preprint arXiv:2108.08707* (2021).