A COMBINATOR, N-DIMENSIONAL ARRAY LIBRARY IN SMALLTALK

by

Conor Hoekstra

Bachelor of Science, Simon Fraser University, 2014

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public for the purpose of scholarly research only.

Abstract

A Combinator, N-Dimensional Array Library in Smalltalk

Conor Hoekstra

Master of Science, Computer Science

Ryerson University, 2022

My thesis is that programming in array languages with combinators (from Combinatory Logic) is a powerful and important aspect of programming and that it can naturally fit into an object-oriented language like Smalltalk. Evidence for this is provided by an experimental implementation of a combinator, n-dimensional array library in Smalltalk. A component of this thesis is a survey of combinatory logic and combinators as they currently exist in modern array programming languages and a study of what makes them so powerful and expressive.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. David Mason, Chair of Computer Science at Ryerson University, for not only being a veritable source of knowledge and an excellent supervisor but also a valuable mentor.

I would also like to thank the following individuals for their review and feedback on parts of this thesis:

- Adám Brudzewsky, Computer Programmer at Dyalog Limited and creator of Extended Dyalog APL[9] and APLCart[8]
- Marshall Lochbaum, creator of the I[39] and BQN[38] programming languages
- Dr. Troels Henriksen, assistant professor at DIKU and creator of the Futhark[55] programming language

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Array programming with combinators enables a power[1] and expressivity[2] not found in many programming languages. An exploration of whether this power and expressivity can be achieved in an object-oriented, message passing language like Smalltalk will be the primary goal of the experimental implementation. This involved writing an experimental implementation of a combinator, n-dimensional array library in Smalltalk. The implementation source code is open source with an MIT License and is freely available at the `Pharo-NDArray`[27] GitHub repository. A component of this thesis is a survey of combinatory logic and combinators as they currently exist in modern array programming languages and a study of what makes them so powerful and expressive.

1.1 Problem Background

1.1.1 Intellectual Gold

While being interviewed on the CoRecursive Podcast[28] in June 2021, I made a remark about APL[3] and the APL family of languages that I would like to quote here[29].

> I feel like I have stumbled on an island of treasure. People who are on boats, they see it, they see the gold lying on the beaches, but there is some kind of scare crow and they say "it looks scary" and so they just keep sailing by. But when I stumbled on this island, I immediately saw a treasure chest of gold and was just looking around wondering "why is everybody just sailing

[1] Power: When used in this thesis, power means "the ability to do a lot with a little."

[2] Expressivity: When used in this thesis, expressivity means "essence without ceremony."[24]

[3] APL is a programming language created by Ken Iverson. It stands for "A Programming Language" taken from the Iverson's 1962 book[33] of the same name.

> by, sure the Unicode symbols look a bit odd, but this paradigm (island) is mind-blowing." So I'm just walking around this island by myself bumping into a ton of folks in the J community and the APL community that are super happy to have curious people show up at their doorstep and there is just so much gold ... intellectual gold.

The array language paradigm is an extremely powerful programming paradigm. Although array languages have waned in popularity over the past two or three decades, n-dimensional array libraries have not.

1.1.2 N-Dimensional Array Libraries

N-dimensional array libraries are libraries that provide an `ndarray` or similarly named type that is a multidimensional container or collection that has both *rank* and *shape*. *Rank* refers to the number of dimensions which are specified in the *shape*. For example, a rank-2 `ndarray` is more commonly referred to as a *matrix* and can have dimensions $n \times m$. Along with the `ndarray` type that n-dimensional array libraries provide, a rich suite of methods that operate on the `ndarray` are typically provided.

N-dimensional array libraries are some of the most popular libraries in the world. The most popular is the Python library NumPy[45]. Python is the most popular programming language in the world according to the TIOBE[56] and PYPL[46] indexes as of April 2022 and NumPy is the most popular library in Python[42].

NumPy and most n-dimensional array libraries originate from APL. In "*Numerical Python*"[4], APL's influence on NumPy is noted:

> The languages which were used to guide the development of NumPy include the infamous APL family of languages, Basis [sic], MATLAB, FORTRAN, S and S+, and others.

So, the most popular programming language's most popular library was influenced by the APL family of languages. While we don't have programming language rankings for the 1960s and 1970s, APL would very likely be in the top ten. All of this is to say that clearly APL did something right and it is worth learning what that is.

NumPy is just one of many n-dimensional array library implementations. Table 1.1 contains a non-exhaustive list of some examples of other implementations. The $*$ indicates the n-dimensional array type is a part of the language.

Based on my research, `Pharo-NDArray` is the first implementation an n-dimensional array library in a Smalltalk. Because Pharo Smalltalk already has `Array` as one of the

Programming Language	Library	Name
C++	ISO Proposal	`std::mdarray`[10]
Python	NumPy	`ndarray`[45]
Julia	*	`Array`[43]
R	*	`array`[47]
Octave	*	`NDArray`[23]
Rust	ndarray	`array`[44]
Wolfram	*	`Array`[3]
Maple	*	`Array`[2]

Table 1.1: N-Dimensional libraries.

built-in collections, I decided to call the n-dimensional array `NDArray` as that is the next most common name.

1.1.3 Combinatory Logic and Combinators

According to the Stanford Encyclopedia of Philosophy[7]:

> Combinatory logic is an elegant and powerful logical theory that is connected to many areas of logic, and has found applications in other disciplines, especially, in computer science and mathematics.

Combinatory logic defines combinators which are functions or expressions with no free variables. This means combinators only refer to their function arguments and can be thought of as "function argument manipulators" or "function application modifiers". Combinators enable "point free" or "tacit" programming (a programming paradigm where function definitions omit their "points" or "arguments") because of this "argument manipulation". For an excellent introduction to point free programming, see the 2016 StrangeLoop conference talk by Amar Shah called "*Point Free Programming or Die*."[53]

Although combinatory logic and combinators were first described in theoretical-mathematics terms, they have become an increasingly useful idiom in programming languages. Functional languages like Haskell have support for point free programming and combinators and concatenative languages like FORTH and Factor also have combinators. However, the languages with the richest support for combinators and point free or tacit programming are array languages, especially what I call "modern combinator array languages" like Dyalog APL, J and BQN. The question is why are they so

powerful and useful in array languages and can this power be replicated in a different programming paradigm such as the object-oriented one where Smalltalk excels.

1.2 Objectives and Proposed Methodology

The objectives are:

- To survey the history of combinatory logic and combinators in array languages, specifically Dyalog APL, J and BQN;
- To design and implement the `NDArray` type and the corresponding rank-polymorphic methods;
- To design and implement both `NDArray` combinators and `Symbol` combinators;
- To explore how to replicate the power of array language combinators in Smalltalk.

The proposed methodology is to implement the `Pharo-NDArray`[27] library as an MIT License open source project on GitHub. I started out using Pharo-9 but transitioned to Pharo-10 when that became available. Meticulous research and surveying was done on the history of combinatory logic and combinators all the way back to the origins of the field in the 1920s.

1.3 Contributions

The main contributions in this thesis can be broken up into two categories:

- Contributions to/from combinatory logic literature:
  - Surveying the history of combinatory logic and combinators in array languages, specifically Dyalog APL, J and BQN;
  - Identifying the relationships between existing combinators in the form of hierarchical specializations;
  - Identifying and naming missing combinators from combinatory logic literature;
  - Identifying the relationship between the B_1 and the Ψ combinators.
- Contributions to/from Smalltalk:

  - Designing and implementing the `NDArray` type, the rank-polymorphic `NDArray` methods, the `NDArray` combinators and the `Symbol` combinators;
  - Providing motivation for a new syntax in Pharo Smalltalk that improves the ergonomics of `FullBlockClosure` invocation.

1.4 Thesis Outline

This thesis is organized as follows:

- Chapter 2 provides:
  - an introduction to array languages
  - a brief history of array languages
  - a brief history of combinatory logic
  - an overview of combinators as they exist in modern array languages
  - a discussion of what makes combinators so powerful in modern array languages
- Chapter 3 discusses the implementation of the `NDArray` type and the corresponding rank-polymorphic methods and combinators
- Chapter 4 discusses the results of the `Pharo-NDArray` design and the contributions of this thesis.
- Chapter 5 presents the conclusions from the above and suggests future work.

Chapter 2

Background

2.1 An Introduction to Array Languages by Example

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 the read the symbols, not only is it extremely readable but it can also be extremely expressive and powerful. The next few examples are aimed at demonstrating this.

2.1.1 First 10 Odd Numbers

The APL code in Figure 2.1 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
```

Figure 2.1: Odd numbers in APL.

2.1.2 Multiplication Table

Other examples that demonstrate the expressivity and power of APL are examples using outer product. The APL code in Figure 2.2 can be used to create a multiplication table.

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

Figure 2.2: Multiplication table in APL.

2.1.3 Maximum Consecutive Ones

The "maximum consecutive ones" problem consists of finding the longest substring of consecutive ones in an array of ones and zeros. Figure 2.3 shows a solution in APL.

```
      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
```

Figure 2.3: Maximum consecutive ones in APL.

2.1.4 Minimum Adjacent Difference

The APL code in Figure 2.4 finds the minimum difference between adjacent numbers.

```
      vec ← 55 42 1429 343
      ⍝ catenate adjacent elements together
      2,/vec
┌─────┬────────┬─────────┐
│55 42│42 1429│1429 343│
└─────┴────────┴─────────┘
      ⍝ replace catenate with minus
      2-/vec
13 ¯1387 1086
      ⍝ take absolute value
      |2-/vec
13 1387 1086
      ⍝ min reduction
      ⌊/|2-/vec
13
```

Figure 2.4: Minimum adjacent difference in APL.

2.1.5 Index of Maximum Value

The APL code in Figure 2.5 finds the index of the maximum value.

```
      vec ← 4 2 3 7 9 1 5 4
      ⍝ max reduction
      ⌈/vec
9
      ⍝ values equal to maximum
      (⌈/=⊢)vec
0 0 0 0 1 0 0 0
      ⍝ indices (index of values equal to 1)
      ⍸(⌈/=⊢)vec
5
      ⍝ first (in case there are multiple indices)
      ⊃⍸(⌈/=⊢)vec
5
```

Figure 2.5: Index of maximum value in APL.

2.1.6 The Power of Array Languages

As demonstrated by these five 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 built-in 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*[1] not found in any other paradigm
- Rank polymorphism (primitives work on different rank arrays)

2.2 A 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[33]. This book 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[36][57].

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[22] 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*"[30]. For a full list of K dialects, visit the APL Wiki[34].

[1] *Flexibility* here means how quickly one can implement different algorithms to solve the same problem. APL's primitive enable multiple solutions to a single problem in only a few keystrokes. See "*Algorithms as Tool of Thought*"[25] for an example.

Two of the most recent array languages that have been developed are I and BQN created by Marshall Lochbaum. 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 APL, J or K. See Table 2.1 for a list of significant array languages and when they were introduced.

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

Table 2.1: Array languages timeline.

Array languages influencing other array languages is very common. See Figure 2.6 for a dependency graph of how array languages influenced each other.

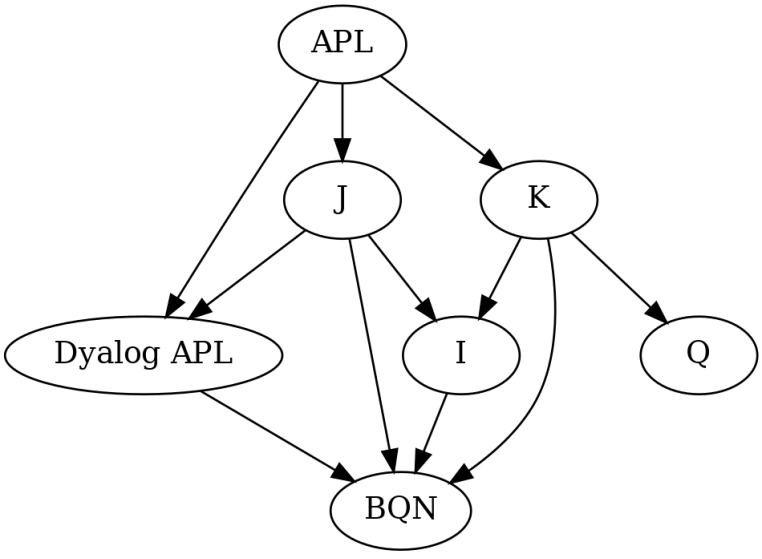

Figure 2.6: Array language influences.

For a more detailed array language influence graph, visit the APL Wiki[21].

2.3 A Brief History of Combinatory Logic

2.3.1 Moses Schönfinkel, 1924

The first paper to introduce combinators was by Moses Schönfinkel in his 1924 "*Über die Bausteine der mathematischen Logik*"[52]. The German title translates to "*On the building blocks of mathematical logic*"[51]. In this paper, Schönfinkel introduces the I, C, S, T and Z combinators. The C, Z and T combinators would later be respectively renamed to K, B and C by Haskell Curry[13]. For an in depth background on the origins of Schönfinkel, see "*Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel*"[62].

2.3.2 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*"[13]. 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.

2.3.3 Haskell Curry, 1930

In Curry's dissertation "*Grundlagen der Kombinatorischen Logik*"[16] (*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.

2.3.4 Haskell Curry, 1931

After publishing his dissertation[16] in 1930, Curry would quickly publish "*The Universal Quantifier in Combinatory Logic*"[18] 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*.

2.3.5 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*"[17] in 1932, "*A Revision of the Fundamental*

Rules of Combinatory Logic"[15] in 1941, and "*The Combinatory Foundations of Mathematical Logic*"[14] in 1942. John Rosser would remark in his review[50] of Curry's "*The Combinatory Foundations of Mathematical Logic*"[14] 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*"[12] in 1948 which was a summary and simplification of his work on combinatory logic up until that point.

2.3.6 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 2.2).

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

Table 2.2: The elementary combinators.

2.3.7 David Turner, 1979

David Turner worked on three different programming languages (SASL[58], KRC[61] and Miranda[60]) 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[59].

2.3.8 Raymond Smullyan, 1985

Raymond Smullyan would extend the list of existing combinators in his logic puzzle book "*To Mock a Mockingbird*"[54]. Among the combinators introduced were the D (Dove), D_2 (Dovekie), E (Eagle) and $\hat{E}$ (Bald Eagle) combinators.

2.4 Combinator Specializations

The list of combinators that will be covered in Section 2.6 and the corresponding lambda expressions are listed in Table 2.3. For a more comprehensive list of combinators, see the list compiled by Chris Rathman[48].

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)
$\hat{E}$	λabcdefg.a(bde)(cfg)
Y	λa.a(λa)

Table 2.3: Combinators and lambda expressions.

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 2.7.

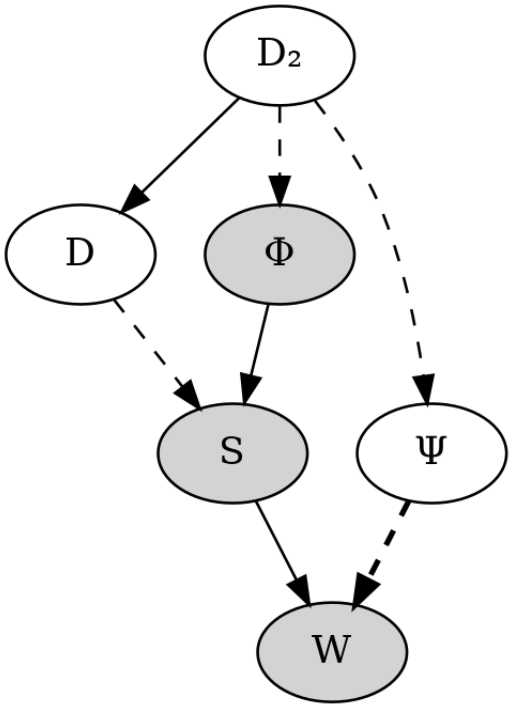

Figure 2.7: D_2 combinator hierarchy.

The two different types of lines and the light gray shading in Figure 2.7 represent the following:

- **Light Gray Node:** The function returned from the combinator is monadic
- **Solid Line Specialization:** Two lambda terms are set equal to each other
- **Dashed Line Specialization:** One lambda term is set to the I combinator
- **Bold Dashed Line Specialization:** Two lambda terms are set equal to each other and a lambda term is set to the I combinator

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

- E is a specialization of $\hat{\text{E}}$ with $b = K$
- Φ_1 is a specialization of $\hat{\text{E}}$ with $d = f$ and $e = g$

A visualization of the specializations/dependencies of the E-family of combinators is shown in Figure 2.8.

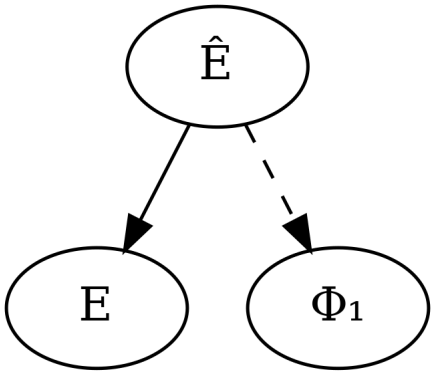

Figure 2.8: Ê combinator hierarchy.

2.5 Evolution of Combinatory Logic in Array Languages

2.5.1 "*Phrasal Forms*", 1989

In 1989, Kenneth Iverson and Eugene McDonnell wrote a paper entitled "*Phrasal Forms*"[41]. 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[2] and 3-trains[3]. They are also sometimes referred to as "invisible modifiers"[49] 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[5] 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[4] 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

[2]An example of a 2-train from APL is: `numAtoms ← ×/⍴` which is functionally equivalent to `numAtoms ← {×/⍴⍵}` and can be read as `numAtoms(x) = product(shape(x))`

[3]An example of a 3-train from APL is: `avg ← +/÷≢` which is functionally equivalent to `avg ← {(+/⍵)÷(≢⍵)}` and can be read as `avg(x) = sum(x) / length(x)`

[4]FP is a programming language created by John Backus. It stands for "Function Programming" and was introduced in his Turing Award paper.

trains. The fork from APL and J as well as *infix notation*[6] from FP are both the Φ and Φ_1 combinators.

2.5.2 "*Hook Conjunction?*", 2006

Table 2.4 highlights the difference in choice of 2-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

Table 2.4: 2 and 3-trains in APL, BQN and J.

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?*"[31]. 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.

2.5.3 Dyalog APL and BQN

Dyalog APL added combinators in the following years[20] and versions seen in Table 2.5.

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	⍥⍤⍨

Table 2.5: History of combinators in Dyalog APL.

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*"[37].

2.6 Combinators in APL, BQN & J

In this section, APL specifically refers to Dyalog APL version 18.0. Furthermore, see Table 2.6 for definitions of the terms used for arity (number of function arguments) in this section.

Arity	Greek Term	Latin Term
1	Monadic	Unary
2	Dyadic	Binary

Table 2.6: Arity terms.

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.

2.6.1 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[52]. It was introduced in Curry and Feys[19] as the **Elementary Identificator**.

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

Table 2.7: I combinator.

A potential implementation in Python:

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

2.6.2 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[52]. Note that Schönfinkel originally called this combinator C but it was later renamed to K by Curry[13]. 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[54].

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

Table 2.8: K combinator.

A potential implementation in Python:

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

2.6.3 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[39] 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[52]. Smullyan nicknamed it Starling[54].

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

Table 2.9: S combinator.

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))
```

2.6.4 B

The B combinator is what most people think of when they hear "composition." It is called `compose` in Racket, `comp` in Clojure, `(.)` in 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[52]. Note that Schönfinkel originally called this combinator Z but Curry would later rename it to B[13]. It was introduced in Curry and Feys[19] as the **Elementary Compositor**. Smullyan nicknamed it the Bluebird[54]. 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.

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	`&:`

Table 2.10: B combinator.

Potential implementations in Python:

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

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

2.6.5 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[12]. Smullyan nicknamed it the Blackbird[54].

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

Table 2.11: B_1 combinator.

Potential implementations in Python:

```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))
```

2.6.6 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[52]. Note that Schönfinkel originally called this combinator T but

Curry would later rename it to C[13]. It was introduced in Curry and Feys[19] as the **Elementary Permutator**. Smullyan nicknamed it the Cardinal[54].

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

Table 2.12: C combinator.

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)
```

2.6.7 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[13]. It was introduced in Curry and Feys[19] as the **Elementary Duplicator**. Smullyan nicknamed it the Warbler[54].

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

Table 2.13: W combinator.

Potential implementations in Python:

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

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

2.6.8 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].

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

Table 2.14: Ψ combinator.

Potential implementations in Python:

```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))
```

2.6.9 Φ (S′)

The Φ combinator is known as "infix notation" in FP[6] and FL[1] 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[18] as the Φ combinator. It was later reintroduced with a different name, the S′ combinator in Turner 1979[59]. Smullyan nicknamed it the Phoenix[54].

Potential implementations in Python:

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

Table 2.15: Φ (S′) combinator.

```
# 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))
```

2.6.10 Φ_1

The Φ_1 combinator is known as "infix notation" in FP[6] and FL[1] 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[18]. In fact, both the Φ and Φ_1 combinators are specializations of the Φ_n combinator. It was nicknamed the Pheasant[26].

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

Table 2.16: Φ_1 combinator.

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))
```

2.6.11 D

The D combinator is also 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[54]. Smullyan nicknamed it the Dove[54].

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

Table 2.17: D combinator.

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))
```

2.6.12 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[39] (called *before* in BQN) both exist. Furthermore, the D_2 combinator can also be spelled in Extended Dyalog APL[9] which implements *reverse compose* (the equivalent of I's *backHook* and BQN's *before*).

Language	D_2
BQN	a⊸b⊸c
Extended Dyalog APL	a⍥b∘c

Table 2.18: D_2 combinator.

In the above spellings, a and c are unary functions and b is a binary function. Smullyan nicknamed the D_2 combinator the Dovekie[54].

Potential implementations in Python:

```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))
```

2.6.13 Y

The Y combinator is also known as the fixed-point combinator. It can be used to define recursive functions in functional programming languages that don't support recursion. All of Dyalog APL, J and BQN support recursion so there is no Y combinator in these languages. However, it was worth including a short section on it due to it being the most well known of all Curry's combinators.

2.7 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 is a rich set of them

(6) They are built-in

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

2.7.1 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 2.19.

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.

	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

Table 2.19: Combinators in Haskell.

(6) Built-in Combinators

It matters less in Haskell that the combinators are not built-in, 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.

(5) 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.

(4) Uniformly Designed Combinators

In Haskell, the combinators are defined nonuniformly. Certain combinators are infix operators, certain combinators are functions and one combinator is both. This 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.

(3) Invisible Combinators

Also known as "invisible modifiers"[49], 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.

(2) Combinator Spelling

The spelling of combinators in Haskell makes the "tax" of using them higher than in array languages. For instance, Table 2.20 shows the additional code required in order to *commute*[5] a minus operation.

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

Table 2.20: Commuting minus.

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.

(1) 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 2.21 shows the precedence of the combinators that take the form of infix operators.

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

Table 2.21: Combinator precedence in Haskell.

[5]Commute means to swap the order of the arguments for a binary function.

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 when using the S combinator to determine if a string is a palindrome:

```
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.

```
≡∘⊸ "tacocat" # success
```

2.7.2 Example: All Absolute Differences

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

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

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.

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 with implicit B
    (⥊|∘-⌜˜) vec
⟨ 0 6 2 9 5 6 0 8 3 11 ...

    # Deduplicate with explicit 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.

Chapter 3

Design of `Pharo-NDArray`

This chapter will discuss in depth the design and implementation of the `NDArray` type and the associated methods and combinators. Note that the following terminology will be used to refer to arrays of different rank:

- **Scalar:** rank-0 array
- **Vector:** rank-1 array
- **Matrix:** rank-2 array

Higher rank arrays are referred to as rank-n arrays. Furthermore, the following terminology will be used when categorizing different types of `Pharo-NDArray` methods:

- **Verb:** A method that acts on arrays[32]
- **Adverb:** A method that acts on verbs and arrays[32]

In the original "*Dictionary APL*"[32], arrays are called "nouns" and adverbs act on verbs *or* nouns, but our `NDArray` methods act on both, while the `Symbol` adverbs and combinators act only on verbs (aka `Symbol`s) and return verbs (aka `FullBlockClosure`s).

3.1 An Introduction to `Pharo-NDArray` by Example

This section will introduce the `Pharo-NDArray` library by implementing solutions to the same examples shown in Section 2.1 when the array languages were introduced.

3.1.1 First 10 Odd Numbers

Figure 3.1 shows how to generate the first 10 odds numbers in Smalltalk using `Pharo-NDArray`.

```
10 iota asNDArray * 2 - 1

"This results in:
1 3 5 7 9 11 13 15 17 19"
```

Figure 3.1: Odd numbers in Smalltalk using `Pharo-NDArray`.

This is actually more verbose than it needs to be because `Integer >> iota` returns an `Interval` class which also supports the mathematical binary operator such as `*` and `-`. Figure 3.2 shows this in action.

```
10 iota * 2 - 1

"This results in:
1 3 5 7 9 11 13 15 17 19"
```

Figure 3.2: Odd numbers in Smalltalk using `Interval`.

3.1.2 Multiplication Table

The multiplication example shown in Figure 3.3 is much more interesting.

```
(9 iota) outerProduct: (9 iota) with: #*

"This results in:
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81"
```

Figure 3.3: Multiplication table in Smalltalk using `Pharo-NDArray` (1).

However, this is slightly irritating because we need to type `10 iota` twice whereas in APL we only needed to type it once thanks to the W combinator in the form of commute. Furthermore, the method `NDArray >> outerProduct:with:` isn't truly in the spirit of an array language because it doesn't return a function. Figure 3.4 shows a solution that is much more in the spirit of APL.

```
9 iota dupWith: #* outerProduct

"This results in:
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81"
```

Figure 3.4: Multiplication table in Smalltalk using `Pharo-NDArray` (2).

Here, we are making use of `Symbol >> outerProduct` instead of `NDArray >> outerProduct`. Discussion of `NDArray >> outerProduct` and other *adverbs* that are defined on `Symbol` can be found in Section 3.4 Design of `Symbol` Adverbs.

3.1.3 Maximum Consecutive Ones

Combining `Pharo-NDArray` with `Pharo-Functional` leads to a very beautiful solution to the maximum consecutive ones problem.

```
| vector |
vector := #(1 1 0 1 1 1 0 0 0 1).
vector dupWith: #partition:
   :> collect: #size
   :> max

"This results in: 3"
```

Figure 3.5: Maximum consecutive ones in Smalltalk using `Pharo-NDArray`.

3.1.4 Minimum Adjacent Difference

Once again, combining `Pharo-NDArray` with `Pharo-Functional` leads to a very beautiful solution to the minimum adjacent difference problem. Note that the *N-wise reduction* from APL is called `windowed:reduce:` in `Pharo-NDArray`.

```
| vector |
vector := #(55 42 1429 343).
vector windowed: 2 reduce: #-
   :> abs
   :> min

"This results in: 13"
```

Figure 3.6: Minimum adjacent difference in Smalltalk using `Pharo-NDArray`.

3.1.5 Index of Maximum Value

Once again, combining `Pharo-NDArray` with `Pharo-Functional` leads to a very beautiful solution to the index of maximum value problem. Note you can use the `dupWith:fork:and:` Φ combinator in `Pharo-NDArray`, but due to its verbosity it isn't quite as elegant as the straight forward solution.

```
| vector |
vector := #(4 2 3 7 9 1 5 4) asNDArray.

"Solution 1"
vector max = vector
   :> indices
   :> first

"Solution 2"
vector dupWith: #max fork: #= and: #yourself
   :> indices
   :> first

"These both result in: 5
(Note both APL and Pharo-NDArray use 1-indexing"
```

Figure 3.7: Index of maximum value in Smalltalk using `Pharo-NDArray`.

3.1.6 All Absolute Differences

This is the "simple example" covered in Section 2.7.2. Once again, Combining `Pharo-NDArray` with `Pharo-Functional` leads to a very beautiful solution.

```
| vector |
vector := #(1 -5 3 -8 6).
vector triangleProduct2: #-
   :> abs
   :> enlist
   :> unique

"This results in:
0 6 2 9 5 8 3 11 14"
```

Figure 3.8: All absolute differences in Smalltalk using `Pharo-NDArray`.

This is actually more expressive than the array solution shown in Section 2.7.2 because array languages currently have no support for *upper* or *triangle* products. The difference between outer, upper and triangle products is shown in Figure 3.9.

```
5 iota dupWith: #+ outerProduct
"1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
 4  8 12 16 20
 5 10 15 20 25"

5 iota upperProduct: #*
"0 2 3  4  5
 0 0 6  8 10
 0 0 0 12 15
 0 0 0  0 20
 0 0 0  0  0"

5 iota triangleProduct: #*
"2 3  4  5
   6  8 10
     12 15
        20"
```

Figure 3.9: `outerProduct`, `upperProduct` and `triangleProduct` in `Pharo-NDArray`.

3.2 Design of the NDArray Type

The design of the NDArray type is quite simple. There are only two instance side variables[1]:

- data, a Smalltalk Array
- shape, a Smalltalk Array

The elements of an NDArray are stored in the data array that is a flattened, contiguous array. The shape array is used to represent the dimensions of the NDArray. For example, if an NDArray has three rows and four columns, the shape with be equal to #(3 4) and the data will be equal to #(1 2 3 4 5 6 7 8 9 10 11 12). This can be seen visually below in Figure 3.10.

```
| matrix |
matrix := 12 iota reshape: #(3 4)

"matrix is:
1  2  3  4
5  6  7  8
9 10 11 12

The instance side variables of matrix are:
data  = #(1 2 3 4 5 6 7 8 9 10 11 12)
shape = #(3 4)"
```

Figure 3.10: NDArray type in Pharo-NDArray.

3.2.1 Directly Constructing NDArrays

There are two methods used to directly construct an NDArray:

- NDArray >> withAll:
- NDArray >> withShape:with:

The implementation of both methods are shown and discussed below. NDArray >> withAll:, constructs rank-1 NDArrays. Below is the implementation in Figure 3.11 and Figure 3.12 shows an example of its usage.

```
NDArray >> withAll: aCollection
   data := aCollection flattened.
   shape := aCollection shape
```

Figure 3.11: `NDArray >> withAll:` implementation.

```
| matrixFromArray |
matrixFromArray := NDArray withAll: #(1 2 3)
"a NDArray: #(1 2 3) with shape: #(3)"
```

Figure 3.12: `NDArray >> withAll:` example usage.

In order to directly construct an `NDArray` of any rank, the method `NDArray >> withShape:with:` must be used. The implementation is shown in Figure 3.13 and Figure 3.14 shows an example of its usage.

```
NDArray >> withShape: aCollection with: anObject
   shape := aCollection.
   data := anObject asArray flattened cycle: shape product
```

Figure 3.13: `NDArray >> withShape:with:` implementation.

```
| matrixA matrixB |
matrixA := NDArray withShape: #(2 4) with: 8 iota.
matrixB := NDArray withShape: #(2 4) with: 5 iota.

"matrixA is:
1 2 3 4
5 6 7 8

matrix B is:
1 2 3 4
5 1 2 3"
```

Figure 3.14: `NDArray >> withShape:with:` example usage (1).

Note that when the number of elements provided is less than the required number of elements for the resulting array, the elements are repeated as necessary. This is achieved

[1] Instance side variables are variables private to the instance.

by using the SequenceableCollection >> cycle: method. This behavior is borrowed from APL which is shown in Figure 3.15.

```
      ⍴ 2 4 reshape iota 5
      2 4⍴⍳5
      1 2 3 4
      5 1 2 3
```

Figure 3.15: Reshape's *cycling* behavior in APL.

In Figure 3.16, an example of constructing an NDArray with rank three and shape equal to #(2 3 4) is shown.

```
| rectangularPrism |
rectangularPrism := NDArray withShape: #(2 3 4) with: 5 iota.

"rectangularPrism is:
1 2 3 4
5 1 2 3
4 5 1 2

3 4 5 1
2 3 4 5
1 2 3 4"
```

Figure 3.16: NDArray >> withShape:with: example usage (2).

3.2.2 Converting to NDArrays

Although you are able to directly construct an NDArray with the two methods NDArray >> withAll: and NDArray >> withShape:with:, it is typically much easier to convert to an NDArray using the asNDArray method. Several objects support the asNDArray method. An implementation of one of these is shown in Figure 3.17.

```
SequenceableCollection >> asNDArray
   ^ NDArray withAll: self asArray
```

Figure 3.17: SequenceableCollection >> asNDArray implementation.

```
vector := #(1 2 3 4) asNDArray.

"vector is:
1 2 3 4"
```

Figure 3.18: `SequenceableCollection >> asNDArray` example usage.

Furthermore, if you would like to convert a Smalltalk `Array` or another collection to an `NDArray` whose rank is greater than one, a convenient option is to use the `SequenceableCollection >> reshape:` method.

```
matrix := #(1 2 3 4 5 6) reshape: #(2 3).

"matrix is:
1 2 3
4 5 6"
```

Figure 3.19: `SequenceableCollection >> reshape:` example usage.

3.3 Design of the `NDArray` Methods

In this section, several examples will be shown from each of the following categories:

- Scalar Monadic Verbs
- Scalar Dyadic verbs
- Monadic Verbs
- Dyadic verbs

However, this is only covers a small portion of the verbs and adverbs that were implemented in `Pharo-NDArray`. For a full list of verbs and adverbs, see Appendix A. Furthermore, in order to keep the examples as short, only examples using array with rank less than three are used. However, many `Pharo-NDArray` methods support `NDArrays` of any rank (I personally have tested up to rank-20).

3.3.1 Scalar Monadic Verbs

The scalar monadic verbs are by far the easiest methods to implement. This is because they are always applied to each 0-rank element of the `NDArray` regardless of the shape. Due to the fact that the elements are stored contiguously in `data`, it is simply a matter of mapping over each element in `data` and performing the monadic operation. In Smalltalk, the `collect:` method can be used to perform such an operation.

A few examples of scalar monadic verbs implemented in `Pharo-NDArray` are:

- `NDArray >> abs` (absolute value)
- `NDArray >> ceiling` (rounds *up* to closest integer)
- `NDArray >> floor` (rounds *down* to closest integer)
- `NDArray >> not` (converts 0s to 1s and 1s to 0s)
- `NDArray >> sign` (converts +ive, 0 and -ive numbers to 1, 0 and -1)

An implementation of several of the scalar monadic verbs will be shown below.

```
NDArray >> abs
   ^ self class new withShape: shape with: (data collect: #abs)
```

Figure 3.20: `NDArray >> abs` implementation.

```
| matrixA matrixB |
matrixA := 6 iota reshape: #(3 3) :> - 5.
matrixB := matrixA abs

"matrixA is:
-4 -3 -2
-1  0  1

matrixB is:
4 3 2
1 0 1"
```

Figure 3.21: `NDArray >> abs` example usage.

```
NDArray >> ceiling
   ^ self class new withShape: shape with: (data collect: #ceiling)
```

Figure 3.22: `NDArray >> ceiling` implementation.

```
| matrixA matrixB |
matrixA := 12 iota reshape: #(3 4) :> + 4 :> / 10.0.
matrixB := matrixA ceiling

"matrixA is:
0.5 0.6 0.7 0.8
0.9 1.0 1.1 1.2
1.3 1.4 1.5 1.6

matrixB is:
1 1 1 1
1 1 2 2
2 2 2 2"
```

Figure 3.23: `NDArray >> ceiling` example usage.

```
NDArray >> sign
   ^ self class new withShape: shape with: (data collect: #sign)
```

Figure 3.24: `NDArray >> sign` implementation.

```
| matrixA matrixB |
matrixA := 6 iota reshape: #(3 3) :> - 5 .
matrixB := matrixA sign

"matrixA is:
-4 -3 -2
-1  0  1

matrixB is:
-1 -1 -1
-1  0  1"
```

Figure 3.25: `NDArray >> sign` example usage.

3.3.2 Scalar Dyadic Verbs

The scalar dyadic verbs are the next easiest to implement after scalar monadic verbs. These verbs are rank-polymorphic so the rank of each `NDArray` (argument) to the dyadic verb will affect the behavior. When either of the `NDArrays` is rank-0, the dyadic verb is applied to each rank-0 elements of the other `NDArray`. When the shapes match, each corresponding pair of rank-0 elements have the dyadic verb applied to them. Otherwise, the operation will result in a `ShapeError`.

A few examples of scalar dyadic verbs implemented in `Pharo-NDArray` are:

- `NDArray >> +` (addition)
- `NDArray >> -` (subtraction)
- `NDArray >> *` (multiplication)
- `NDArray >> /` (division)
- `NDArray >> min:` (minimum)
- `NDArray >> max:` (maximum)

An implementation of several of the scalar dyadic verbs will be shown below.

`NDArray >> +`

The implementation of `NDArray >> +` is shown in Figure 3.26.

```
NDArray >> + anObject
   ^ self genericScalarDyadic: anObject with: #+
```

Figure 3.26: `NDArray >> +` implementation.

Below in Figure 3.27 are examples of how `NDArray >> +` can be used rank-polymorphically.

`NDArray >> - * / min: max:`

If we take a look at five other dyadic scalar verbs in Figure 3.28, we will notice a pattern. They all invoke a helper method called `NDArray >> genericScalarDyadic:with:`.

The implementation is shown in Figure 3.29 and it deals with three different cases:

```
| scalar vector matrix |
scalar := 10.
vector := 3 iota asNDArray.
matrix := 6 iota reshape: #(2 3).

"scalar is: 10
 vector is: 1 2 3
 matrix is: 1 2 3
            4 5 6"

scalar + vector. "11 12 13"
vector + vector. "2 4 6"
matrix + scalar.
"11 12 13
 14 15 16"

matrix + matrix.
"2  4  6
 8 10 12"

matrix + vector. "ShapeError"
```

Figure 3.27: `NDArray >> +` example usage.

```
NDArray >> - anObject
   ^ self genericScalarDyadic: anObject with: #-

NDArray >> * anObject
   ^ self genericScalarDyadic: anObject with: #*

NDArray >> / anObject
   ^ self genericScalarDyadic: anObject with: #/

NDArray >> min: aNumber
   ^ self genericScalarDyadic: aNumber with: #min:

NDArray >> max: aNumber
   ^ self genericScalarDyadic: aNumber with: #max:
```

Figure 3.28: `NDArray >> - * / min: max:` implementations.

1. `ShapeError` This occurs when neither `NDArray` 's rank is zero and the shapes are different.

2. `collect:with:` This occurs when shapes are equal. An element-wise application of the dyadic verb occurs.

3. `collect:` This occurs when the rank of one `NDArray` is 0. An application of a partially applied dyadic verb (using `@@` from `Pharo-Functional` and the scalar) to each rank-0 element in the other `NDArray` occurs.

```
NDArray >> genericScalarDyadic: anObject with: aBlock
   | rank0 shapesEqual |
   rank0 := self rank = 0 or: anObject rank = 0.
   shapesEqual := shape = anObject shape.
   (rank0 or: shapesEqual) ifFalse: [ ^ ShapeError signal ].
   shapesEqual ifTrue: [
      ^ self class new withShape: shape
          with: (self data with: anObject data collect: aBlock)
      ].
   ^ self class new withShape: shape
        with: (data collect: aBlock @@ anObject)
```

Figure 3.29: `NDArray >> genericScalarDyadic:with:` implementation.

3.3.3 Monadic Verbs

Implementation of non-scalar verbs is in many cases a lot tricker than scalar verbs. This is because supporting the verbs rank-polymorphically can differ from verb to verb. Let's take a look at a few. Below is a list of a few of the monadic (non-scalar) verbs supported by `Pharo-NDArray`.

- `NDArray >> reverse` (reverses array)
- `NDArray >> ravel` (flattens array to be rank-1)
- `NDArray >> indices` (returns indices of elements with value 1)
- `NDArray >> unique` (removes duplicates from array)
- `NDArray >> uniqueMask` (returns boolean mask with 1s for first occurrence)

An implementation of several of the monadic verbs will be shown below.

`NDArray >> reverse`

`NDArray >> reverse` always operates on the *last axis* meaning "visually" on every row. This is rank-polymorphism in action. It requires the implementation to first "group" the data by the length of the last axis and only then reverse each of the "groups" individually. Taking a closer look at the implementation of `NDArray >> reverse`, `shape last` gives the length of the last axis and `data groupsOf: shape last` does the "grouping." From there, each "group" can be reversed using the familiar `collect:` method.

```
NDArray >> reverse
   | newData |
   self data isEmpty ifTrue: [ ^ self ].
   newData := data groupsOf: shape last
           :> collect: #reversed
           :> asArray.
   ^ self class new withShape: shape with: newData
```

Figure 3.30: `NDArray >> reverse` implementation.

```
| vector matrix |
vector := 3 iota asNDArray.
matrix := 6 iota reshape: #(2 3).

"vector is: 1 2 3
 matrix is: 1 2 3
            4 5 6"

vector reverse. "3 2 1"
matrix reverse.
"3 2 1
 6 5 4"
```

Figure 3.31: `NDArray >> reverse` example usage.

`NDArray >> ravel`

Probably one of my favorite algorithm implementations because of its simplicity is `NDArray >> ravel`. It is more well known as *flatten* or *join* depending on the programming language. In array languages, it essentially reshapes the array to be rank-1. For vectors (rank-1 arrays), this has no impact as the array already has rank-1. For

higher rank arrays, it effectively changes the shape to be rank-1 with a length of the product of the previous shape. Without knowing how the `NDArray` type is designed, this might sound like a non-trivial implementation, but because we already store the `data` contiguously, we can just "remove" the shape and use `data` to create a new `NDArray`.

```
NDArray >> ravel
   ^ self class new withAll: data
```

Figure 3.32: `NDArray >> ravel` implementation.

```
| vector matrix |
vector := 3 iota asNDArray.
matrix := 6 iota reshape: #(2 3).

"vector is: 1 2 3
 matrix is: 1 2 3
            4 5 6"

vector ravel. "1 2 3"
matrix ravel. "1 2 3 4 5 6"
```

Figure 3.33: `NDArray >> ravel` example usage.

`NDArray >> indices`

`NDArray >> indices` is rare to find in other programming languages. Before we take a look at the implementation lets review the *collect* methods we have seen up until now including a new one:

- `SequenceableCollection >> collect:`
- `SequenceableCollection >> with:collect:`
- `SequenceableCollection >> collectWithIndex:`

These are all extremely important methods and not only is `SequenceableCollection >> collectWithIndex:` a great addition to our collection, it is exactly what we need for `NDArray >> indices`. We will also make use of Smalltalk's *filter* algorithm, `SequenceableCollection >> select:`. We will also need to understand the `Point` class in Smalltalk. Figure 3.32 shows a short usage example.

```
| p |
p := Point x: 1 y: 2. "construct a point 1@2"
p := 1@2.             "shortcut construction"
p x.                  "Accessing first element 1"
p y.                  "Accessing second element 2"
```

Figure 3.34: `Point` example usage.

When you combine `Point` with the *collect* methods you can write code like this:

```
| a b ps xs ys |
a  := #(1 2 3 4 5).
b  := #(10 20 30 40 50).
ps := a with: b collect: #@.
xs := ps collect: #x.
ys := ps collect: #y.

"ps is {(1@10). (2@20). (3@30). (4@40). (5@50)}
 xs is #(1 2 3 4 5)
 ys is #(10 20 30 40 50)"
```

Figure 3.35: `Point` and `SequenceableCollection >> collect:` example usage.

We now have everything we need. `NDArray >> indices` bundles each 0 or 1 element with its corresponding index using `SequenceableCollection >> colletWithIndex:` and the `Point >> @` construction shortcut, filters out the `Points` where `x` is equal to 0 using `SequenceableCollection >> select:`, and then removes the 1s by using `SequenceableCollection >> collect:` with the `Point >> y` method.

```
NDArray >> indices
   | newData |
   self max > 1 ifTrue: [ ^ DomainError signal ].
   self min < 0 ifTrue: [ ^ DomainError signal ].
   self rank > 1 ifTrue: [ ^ NotYetImplemented signal ].
   newData := self data collectWithIndex: #@
           :> select: #= @@ 1 <| #x
           :> collect: #y.
   ^ newData asNDArray
```

Figure 3.36: `NDArray >> indices` implementation.

```
| vectorA vectorB |
vectorA := #(1 0 1 0 1).
vectorB := #(0 1 0 1 0).

vectorA indices. "1 3 5"
vectorB indices. "2 4"
```

Figure 3.37: `NDArray >> indices` example usage.

We will discuss the usage of `<|` more in Section 3.5.2 on `Symbol` Combinators.

`NDArray >> unique`

Let's take a look a `NDArray >> unique`. This implementation is interesting especially when contrasted with `NDArray >> reverse`. It is opposite of `NDArray >> reverse` in some ways, because instead of "grouping" elements by on the last axis, the deduplication happens on the "major cells" or visually the largest entity that is not the whole array. The term "major" can be defined as follows for the different types of arrays:

- Major cells of vectors are the elements (aka scalars)
- Major cells of matrices are the rows (aka vectors)
- Major cells of rank-n arrays are the rank-$n-1$ arrays

In order to find the uniqueness, we then need to do the "opposite" of what was done for `NDArray >> reverse` by taking not the "last" but "all but the first".

- `NDArray >> reverse` uses `shape last`
- `NDArray >> unique` uses `shape allButFirst product`

Defining `atoms` to be `shape allButFirst product` enables us to operate on the major cells. Once we have done this, it is just simply a matter of making a call to `SequenceableCollection >> unique`. The full implementation can be seen below in Figure 3.38.

`NDArray >> uniqueMask`

`NDArray >> uniqueMask` is similar to `NDArray >> unique` but instead of removing duplicate values, it returns an `NDArray` boolean mask where the 1s correspond *not*

```
NDArray >> unique
   | atoms newData newShape |
   self rank = 0 ifTrue: [ ^ self ].
   self rank = 1 ifTrue: [ ^ data unique asNDArray ].
   atoms := shape allButFirst product.
   newData := (data groupsOf: atoms) asArray unique.
   newShape := (Array with: newData size) , shape allButFirst.
   ^ self class new withShape: newShape with: newData
```

Figure 3.38: `NDArray >> unique` implementation.

```
| vector matrix |
vector := #(3 4 2 1 3 5 2 1 4 4 3) asNDArray.
matrix := 6 iota reshape: #(4 3).

"vector is: 3 4 2 1 3 5 2 1 4 4 3
 matrix is: 1 2 3
            4 5 6
            1 2 3
            4 5 6"

vector unique.      "3 4 2 1 5"
vector unique sort. "1 2 3 4 5"
matrix unique.
"1 2 3
 4 5 6"
```

Figure 3.39: `NDArray >> unique` example usage.

to unique elements in the array but to the first occurrence of each unique value in the `NDArray`. The implementation is almost identical to the implementation of `NDArray >> unique` but instead of invoking `SequenceableCollection >> unique` on `data`, `SequenceableCollection >> uniqueMask` is invoked. Figure 3.40 shows the implementation.

`SequenceableCollection >> uniqueMask` is a `Pharo-NDArray` method and its implementation is shown below in Figure 3.42.

Behind the scenes, a `Set` is being used to keep track of the values that have already been seen. Whenever a new value that hasn't been seen before is encountered, a 1 is "collected", otherwise a 0 is.

```
NDArray >> uniqueMask
    | atoms newData newShape |
    self rank = 0 ifTrue: [ ^ 1 ].
    self rank = 1 ifTrue: [ ^ data uniqueMask asNDArray ].
    atoms := shape allButFirst product.
    newData := (data groupsOf: atoms) asArray uniqueMask.
    newShape := Array with: shape first.
    ^ self class new withShape: newShape with: newData
```

Figure 3.40: `NDArray >> uniqueMask` implementation.

```
| vectorA vectorB matrix |
vectorA := #(1 2 3 4 5) asNDArray.
vectorB := #(1 2 2 3 3 3) asNDArray.
matrix := 3 iota reshape: #(2 3).

"vectorA is 1 2 3 4 5
 vectorB is 1 2 2 3 3 3
 matrix is 1 2 3
           1 2 3"

vectorA uniqueMask. "1 1 1 1 1"
vectorB uniqueMask. "1 1 0 1 0 0"
matrix uniqueMask.  "1 0"
```

Figure 3.41: `NDArray >> uniqueMask` example usage.

```
SequenceableCollection >> uniqueMask
   | values |
   values := Set new.
   ^ self collect: [ :e |
        (values includes: e)
           ifTrue:  [ 0 ]
           ifFalse: [ values add: e. 1 ]
   ]
```

Figure 3.42: `SequenceableCollection >> uniqueMask` example usage.

3.3.4 Dyadic Verbs

A few of the dyadic (non-scalar) verbs implemented in `Pharo-NDArray` are:

- `NDArray >> matches:` (returns 1 is both arrays are equal)
- `NDArray >> rotate:` (shifts the first n elements to the back)
- `NDArray >> drop:` (*drops* the n elements)
- `NDArray >> take:` (*takes* the n elements)
- `NDArray >> partition:` (splits array by 0s)
- `NDArray >> membersOf:` (returns 1 if elements exist in array)

Let's take a look a some of their implementations.

`NDArray >> matches:`

One of the simplest verbs to implement is `NDArray >> matches:`. `NDArray >> matches:` checks if both the shapes and the values are the same. If both of these are true, `NDArray >> matches:` returns true, otherwise `NDArray >> matches:` returns false. The implementation is shown below in Figure 3.43.

```
NDArray >> matches: anObject
   self shape = anObject asNDArray shape ifFalse: [ ^ false ].
   ^ self data = anObject asNDArray data
```

Figure 3.43: `NDArray >> matches:` implementation.

```
| matrixA matrixB |
matrixA := 10 iota asNDArray + 10 reshape: #(5 2).
matrixB := (20 iota asNDArray drop: 10) reshape: #(5 2).

"matrixA and matrixB are:
11 12 13 14 15
16 17 18 19 20"

matrixA matches: matrixB.           "true"
matrixA matches: matrixB transpose. "false"
```

Figure 3.44: `NDArray >> matches:` example usage.

`NDArray >> rotate:`

In array languages, `NDArray >> rotate:` is the dyadic version of `NDArray >> reverse:`. Figure 3.45 shows an example in APL.

```
    ⍴ iota 5
    ⍳5
1 2 3 4 5

    ⍴ reverse iota 5
    ⌽⍳5
5 4 3 2 1

    ⍴ 2 rotate iota 5
    2⌽⍳5
3 4 5 1 2
```

Figure 3.45: Reverse and rotate in APL.

As you can see, *rotate* moves the first two elements to the back of the array in the above example. `NDArray >> rotate:` falls into the same "axis-last" algorithm category as `reverse`, `reduce:`, `scan:` and others. It uses `shape last` to group the "axis-last" elements and then `collect:`s over them using a partially applied `rotate:` method using `@@` from `Pharo-Functional`. The full implementation of `NDArray >> rotate:` is below in Figure 3.46.

```
NDArray >> rotate: anInteger
    | newData |
    newData := data groupsOf: shape last
            :> collect: #rotate: @@ anInteger
            :> asArray.
    ^ self class new withShape: shape with: newData
```

Figure 3.46: `NDArray >> rotate:` implementation.

`@@` binds an argument to a symbol and is a very useful binary message. Figure 3.48 shows different ways to "partially apply" or "bind" to methods. All of the message and block invocations result in the value `3`.

```
| vector matrix |
vector := #(1 2 3 4 5).
matrix := 8 iota reshape: #(4 4).

"vector is: 1 2 3 4 5
 matrix is: 1 2 3 4
            5 6 7 8"

vector rotate: 2. "3 4 5 1 2"
matrix rotate: 2.
"3 4 1 2
 7 8 5 6"
```

Figure 3.47: `NDArray >> rotate:` example usage.

```
| plus plusOne |

"Binary message"
1 + 2

"Binary block"
plus := [ :a :b | a + b ].
plus value: 1 value: 2

"Binary block using #"
plus := #+.
plus value: 1 value: 2

"Unary block"
plusOne := [ :a | a + 1 ].
plusOne value: 2

"Unary block using @@"
plusOne := 1 @@ #+.
plusOne value: 2
```

Figure 3.48: Partial application in Smalltalk.

`NDArray >> take:`

`NDArray >> take:` is an algorithm that exists in many functional languages under the same name. It takes an integer argument and returns that many elements from the

array (starting from the beginning). If the number of elements "taken" is more than the number of elements in the array, it will "pad" the default value (0 for numbers) to make up the difference. This behavior is once again borrowed from APL.

```
    ⍳ iota 5
    ⍳5
1 2 3 4 5

    ⍳ 10 take iota 5
    10↑⍳5
1 2 3 4 5 0 0 0 0 0
```

Figure 3.49: Take *padding* behavior in APL.

Figure 3.49 shows the implementation of `NDArray >> take:`.

```
NDArray >> take: anInteger
   | n |
   anInteger isInteger ifFalse: [ ^ InvalidArgumentError signal ].
   self rank > 1 ifTrue: [ ^ NotYetImplemented signal ].
   self rank = 0 ifTrue: [ ^ data reshape: anInteger asArray ].
   n := data size min: anInteger.
   ^ ((data first: n) , (0 repeat: anInteger - n)) asNDArray
```

Figure 3.50: `NDArray >> take:` implementation.

```
| scalar vector |
scalar := 10.
vector := #(1 2 3 4 5).

scalar take: 5.  "10 10 10 10 10"
vector take: 3.  "1 2 3"
vector take: 5.  "1 2 3 4 5"
vector take: 10. "1 2 3 4 5 0 0 0 0 0"
```

Figure 3.51: `NDArray >> take:` example usage.

`NDArray >> partition:`

`NDArray >> partition:` was used in Section 2.1.3 Maximum Consecutive Ones. It takes

a boolean NDArray (aka an NDArray consisting of 0s and 1s) as the right argument and then "splits" the left argument by dropping the 0s and creating sublists of the elements that correspond to consecutive ones. The implementation makes use of:

- SequenceableCollection >> groupByRuns:
- SequenceableCollection >> select:
- SequenceableCollection >> collect:

The SequenceableCollection >> groupByRuns: is the second algorithm in the "group" family. SequenceableCollection >> groupsOf: "groups" or "chunks" n elements at a time while SequenceableCollection >> groupByRuns: "groups" consecutive equal elements together. If we construct Points from the values and the boolean mask and then group based on the boolean mask, we will then have *partitioned* points and we can simply filter out the booleans by using SequenceableCollection >> select: and SequenceableCollection >> collect:. Figure 3.52 shows the implementation.

```
NDArray >> partition: anObject
   ^ data with: anObject asNDArray data collect: #@
      :> groupByRuns: #y
      :> select: #first |> #y |> #asBoolean
      :> collect: #collect: @@ #x |> #asNDArray
      :> asNDArray
```

Figure 3.52: NDArray >> partition: implementation.

```
| vector nestedList |
vector := #(1 2 3 4 5) asNDArray.
nestedList := vector partition: #(1 1 0 1 1).

"nestedList is: [[1 2] [4 5]]"
```

Figure 3.53: NDArray >> partition: example usage.

We will discuss the usage of |> more in Section 3.5.2 on Symbol Combinators.

3.3.5 Adverbs

As mentioned at the beginning of Chapter 3, in Pharo-NDArray adverbs can be thought of as functions that take verbs as arguments. Most programming languages refer to these

kinds of functions as higher order functions. APL (and array languages in general) is one of the few languages that makes a clear delineation between "functions" (aka verbs) and "higher order functions" (aka adverbs).

This delineation between functions and higher order functions is part of what enables APL and array languages to give higher precedence to adverbs. It seems unfortunate that other languages to not adopt this pattern is it naturally makes sense for functions that operate on functions to evaluate before functions that operate on data.

Here are a few of the adverbs that are implemented in `Pharo-NDArray`:

- `NDArray >> reduce:`
- `NDArray >> scan:`
- `NDArray >> outerProduct:with:`
- `NDArray >> triangleProduct:`
- `NDArray >> windowed:reduce:`

An implementation of several of the adverbs will be shown below.

`NDArray >> reduce`

If you remember the `NDArray >> reverse` method, `NDArray >> reduce` is very similar in that it "groups" on the last axis. The resulting shape of the array is different however because we are effectively collapsing one of the dimensions and therefore reducing the rank by one. This is where the name "reduce" comes from. Many people erroneously think the name "reduce" comes from the fact that we are usually "reducing" down to a single value, but in fact it is because we are reducing the dimensions by one. Note that there are also many convenience methods for common reductions such as `NDArray >> sum`, `NDArray >> product` and others. Figure 3.54 shows the implementation and figure 3.54 shows some examples.

`NDArray >> scan`

`NDArray >> scan:` is the sibling algorithm of `NDArray >> reduce:`. It has identical behavior but it returns all the incremental results as well. This means that a property of a `NDArray >> scan:` is that the last value (on each axis) will be equal to the result of a `NDArray >> reduce:`. Note that this implementation is simpler because the shape is not changing. Figure 3.56 shows the implementation and figure 3.57 shows some examples.

```
NDArray >> reduce: aBlock
   | newData |
   self rank = 1 ifTrue: [ ^ data reduce: aBlock ].
   newData := data groupsOf: shape last
           :> collect: aBlock reduce
           :> asArray.
   ^ self class new withShape: shape allButLast with: newData
```

Figure 3.54: `NDArray >> reduce:` implementation.

```
| vector matrix |
vector := #(1 2 3 4 5) asNDArray.
matrix := 6 iota reshape: #(3 3).

"vector is: 1 2 3 4 5
 matrix is: 1 2 3
            4 5 6"

vector reduce: #+. "15"
vector reduce: #*. "120"
matrix reduce: #+. "6 15"
matrix reduce: #*. "6 120"
```

Figure 3.55: `NDArray >> reduce:` example usage.

```
NDArray >> scan: aBlock
   | newData |
   newData := data groupsOf: shape last
           :> collect: aBlock scan
           :> asArray.
   ^ self class new withShape: shape with: newData
```

Figure 3.56: `NDArray >> scan:` implementation.

`NDArray >> outerProduct:with:`

`NDArray >> outerProduct:with:` is a very powerful algorithm. It takes two vectors and creates a "table" or matrix by pairing all elements from both arrays and applying a specific binary operation to them. The implementation currently only support rank-1 arrays. It operates by constructing columns by collecting over the right array with the left array repeatedly. This means the `newShape` is actually reversed so that we end up

```
| vector matrix |
vector := #(1 2 3 4 5) asNDArray.
matrix := 6 iota reshape: #(3 3).

"vector is: 1 2 3 4 5
 matrix is: 1 2 3
            4 5 6"

vector scan: #+. "1 3 6 10 15"
vector scan: #*. "1 2 6 24 120"
matrix scan: #+.
"1  3  6
 4  9 15"

matrix scan: #*.
"1  2   6
 4 20 120"
```

Figure 3.57: `NDArray >> scan:` example usage.

with the correct shape when we `NDArray >> transpose` the `NDArray` before returning it. Figure 3.58 shows the implementation and figure 3.59 shows some examples.

```
NDArray >> outerProduct: anObject with: aBlock
   | newData newShape |
   self rank > 1 ifTrue: [ ^ RankError signal ].
   anObject rank > 1 ifTrue: [ ^ RankError signal ].
   newData := anObject asNDArray collect: self @@ aBlock.
   newShape := { anObject size. self data size }.
   ^ self class new withShape: newShape with: newData enlist asArray
      :> transpose
```

Figure 3.58: `NDArray >> outerProduct:with:` implementation.

`NDArray >> windowed:reduce:`

Currently, `NDArray >> windowed:reduce:` is only implemented for rank-1 arrays. There is a check as well to make sure that the *window size* is less than or equal to the size of the array. Once this is done, we use `NDArray >> with:collect:` over the "windows" of size `anInteger` and a reduction is performed with the specified `aBlock`.

```
| vectorA vectorB |
vectorA := #(1 2 3 4).
vectorB := #(1 2 3).

vectorA outerProduct: vectorB with: #+.
"2 3 4
 3 4 5
 4 5 6
 5 6 7"

vectorB outerProduct: vectorA with: #*.
"1 2 3  4
 2 4 6  8
 3 6 9 12"
```

Figure 3.59: `NDArray >> outerProduct:with:` example usage.

`SequenceableCollection >> copyFrom:to:` is used to create the "windows" on which the reduction is performed. Figure 3.60 shows the implementation and figure 3.61 shows some examples.

```
NDArray >> windowed: anInteger reduce: aBlock
   | n newData fold |
   self rank = 0 ifTrue: [ ^ RankError signal ].
   self rank > 1 ifTrue: [ ^ NotYetImplemented ].
   anInteger isInteger ifFalse: [ ^ InvalidArgumentError signal ].
   anInteger > self size ifTrue: [ ^ ShapeError signal ].
   n := self size - anInteger + 1.
   fold := [ :i :j | (data copyFrom: i to: j) reduce: aBlock ].
   newData := n iota with: n iota + anInteger - 1 collect: fold.
   ^ self class new withAll: newData asArray
```

Figure 3.60: `NDArray >> windowed:reduce:` implementation.

3.4 Design of the `Symbol` Adverbs

Many of the `NDArray` adverbs are also implemented on `Symbol`. They all return `FullBlockClosure`s making them much more in the spirit of array language adverbs because in array languages, the equivalent of the adverbs implemented in `Pharo-NDArray`

```
| vector |
vector := #(1 2 3 4 5).

vector windowed: 2 reduce: #+. "3 5 7 9"
vector windowed: 2 reduce: #+. "2 6 12 20"
vector windowed: 3 reduce: #+. "6 9 12"
vector windowed: 3 reduce: #+. "6 24 60"
```

Figure 3.61: `NDArray >> windowed:reduce:` example usage.

take only verbs as arguments and return verbs. Here is a list of a few of the adverbs that are implemented on `Symbol`.

- `Symbol >> reduce`
- `Symbol >> scan`
- `Symbol >> outerProduct`
- `Symbol >> upperProduct`

Note that all of these are *unary messages*. If we take a look at the implementations of each of these, we can see that they all return either a unary `FullBlockClosure` or a binary `FullBlockClosure`.

```
Symbol >> reduce
   ^ [ :e | e reduce: self ]
```

Figure 3.62: `Symbol >> reduce` example usage.

```
Symbol >> scan
   ^ [ :e | e scan: self ]
```

Figure 3.63: `Symbol >> scan` example usage.

A simple example of the difference in usage is shown in Figure 3.66

Note that because the `Symbol >> reduce` adverb returns a `FullBlockClosure`, if you immediately need to call the result of the adverb you must use the `BlockClosure >> value:` method. However, you can combine `Symbol` adverbs with

```
Symbol >> outerProduct
   ^ [ :x :y | x outerProduct: y with: self ]
```

Figure 3.64: `Symbol >> outerProduct` example usage.

```
Symbol >> upperProduct
   ^ [ :x | x upperProduct: self ]
```

Figure 3.65: `Symbol >> upperProduct` example usage.

```
| vector |
vector := #(1 2 3 4 5) asNDArray.

"Pharo-NDArray reduce: adverb"
vector reduce: #+.

"Symbol reduce adverb"
(#+ reduce) value: vector.
```

Figure 3.66: `NDArray >> reduce:` vs `Symbol >> reduce`.

`Pharo-NDArray` combinators to get more expressive code that does not need to use the `BlockClosure >> value:` method because you are not using the `FullBlockClosure` immediately. An example of this is when using the `NDArray >> dupWith:` combinator with is shown in Figure 3.67.

```
4 iota dupWith: #= outerProduct.
"1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1"
```

Figure 3.67: `Symbol >> outerProduct` with `dupWith:` example usage.

3.5 Design of the Combinators in `Pharo-NDArray`

In `Pharo-NDArray` there are two different types of combinators implemented:

- `NDArray` combinators

- `Symbol` combinators

There are two primary differences between the two. The first is that the `NDArray` combinators are *keyword messages* while the `Symbol` combinators are *binary messages*. For further detail see the top of Section 3.5.1 Smalltalk Messages & Precedence. The second difference is that the `Pharo-NDArray` combinators take both verbs, `NDArray`s and include the application of the verbs while the `Symbol` combinators return `FullBlockClosure`s and only take verbs as arguments and do not include the application.

An example of this can be seen below in Figure 3.68.

```
"Example using NDArray >> dupWith:hook:"
'tacocat' dupWith: #matches: hook: #reverse. "true"

"Example using Symbol >> <*>"
#matches: <*> #reverse value: 'tacocat'. "true"
```

Figure 3.68: `Symbol` vs `NDArray` combinators.

Now let's dive into the implementations and some more examples.

3.5.1 `NDArray` Combinators

Smalltalk Messages & Precedence

Before we take a look at the `Pharo-NDArray` combinators, the types and precedence of Smalltalk messages should be explained. There are three types of messages in Smalltalk[11]:

1. Unary messages
2. Binary messages
3. Keyword messages

Unary messages are postfix and are a single word that do *not* have a colon at the end. Binary messages are infix and use one or more ASCII symbols from the set shown below. Keyword messages are one or more words each ending in colons. Unary messages have higher precedence than binary messages and binary messages have higher precedence than keyword messages. Some examples are shown in Figure 3.69.

Note that when messages with the same precedence are juxtaposed (as they are in the binary message example above), the expression is evaluated from left to right.

```
"Unary messages"
#(1 2 3 4 5) first. "1"
#(1 2 3 4 5) last.  "5"
#(1 2 3 4) size.    "4"

"Binary messages (!%&*+,-/<=>?@\|~)"
1 + 2.     "3"
1 + 2 * 3. "9"

"Keyword messages"
#(1 2 3 4 5) first: 2. "1 2"
#(1 2 3 4 5) last: 2.  "4 5"

"Unary, binary and keyword messages"
#(1 2 3 4 5) last * 2 repeat: 3.           "10 10 10"
#(1 2 3 4 5) first: 2 + #(3 2 1) last.     "1 2 3"
#(1 2 3 4 5) first: (2 + (#(3 2 1) last)). "1 2 3"
```

Figure 3.69: Smalltalk unary, binary and keyword message precedence.

Furthermore, the last two expressions are equivalent due to the precedence order of messages.

`NDArray` Combinator Table

Below in Table 3.1 is a list of the combinators and their equivalent bird names from To Mocking a Mockingbird by Raymond Smullyan[54] and their corresponding `NDArray` method names.

We are going to look in detail at the following six `NDArray` combinators:

- `NDArray >> dupWith:hook:`
- `NDArray >> dupWith:backHook:`
- `NDArray >> and:with:over:`
- `NDArray >> and:with:atop:`
- `NDArray >> and:with:hook:`
- `NDArray >> and:with:backHook:`

Combinator	Bird Name	NDArray Method Name
C	Cardinal	`flip:with:`
B	Bluebird	`with:atop:`
B_1	Blackbird	`and:with:atop:`
		`and:with:backHook:`
Φ_1	Pheasant	`and:with:fork:and:`
D	Dove	`and:with:hook:`
Ψ	Psi	`and:with:over:`
W	Warbler	`dupWith:`
		`dupWith:atop:`
		`dupWith:backHook:`
Φ	Phoenix	`dupWith:fork:and:`
S	Starling	`dupWith:hook:`
		`dupWith:over:`

Table 3.1: `NDArray` combinator method names.

`NDArray >> dupWith:hook:`

The S combinator is one of the most well known combinators. The composition (as discussed in Section 2.6.3) is to evaluate a unary function applied to the `NDArray`, and then along with the original `NDArray` passes these as the two arguments to the binary function. Its implementation is shown below in Figure 3.70.

```
NDArray >> dupWith: f hook: g
     "This is known as the S combinator from Combinatory
     Logic and the 'starling' from Combinator Birds"
     f numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
     g numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
     ^ f value: self value: (g value: self)
```

Figure 3.70: `NDArray >> dupWith:hook:` implementation.

Figure 3.71 shows one of the classic examples of using the S combinator to check whether an array or string is palindromic. It can also frequently be used with `NDArray` verbs that take a boolean mask as one of the arguments, such as in the last example of Figure 3.71 where we us `NDArray >> filter:` to filter in the odd elements.

`NDArray >> dupWith:backHook:`

This "combinator" doesn't actually correspond to a named combinator in combinatory logic literature. However, it does exist in the I programming language under the name

```
'tacocat' dupWith: #matches: hook: #reverse. "true"
'tacodog' dupWith: #matches: hook: #reverse. "false"
#(1 2 3 4 5) dupWith: #filter: hook: #% @@ 2.  "1 3 5"
```

Figure 3.71: `NDArray >> dupWith:hook:` example usage.

backHook[39], in Extended Dyalog APL under the name *reverse compose*[9] and in BQN under the name *before*[38]. It is the sibling combinator to the S combinator. The implementation is shown below in Figure 3.72.

```
NDArray >> dupWith: f backHook: g
   "The name backHook is taken from I, a predecessor language to BQN."
   f numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
   g numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
   ^ g value: (f value: self) value: self
```

Figure 3.72: `NDArray >> dupWith:backHook:` implementation.

With a commutative binary function like `NDArray >> matches:`, `NDArray >> dupWith:hook:` and `NDArray >> dupWith:backHook:` have the same behavior. The way the verbs have been designed in `Pharo-NDArray` makes `NDArray >> dupWith:backHook:` much less useful than `NDArray >> dupWith:hook:`. The final example in Figure 3.73 is trivial but demonstrates the behavior.

```
'tacocat' dupWith: #reverse backHook: #matches:.        "true"
'tacodog' dupWith: #reverse backHook: #matches:.        "false"
#(1 2 3 4 5) dupWith: #* @@ 2 backHook: #intersection:. "2 4"
```

Figure 3.73: `NDArray >> dupWith:backHook:` example usage.

`NDArray >> and:with:over:`

The Ψ combinator comes up very frequently. `NDArray >> and:with:over:` takes a unary function and applies it to two different arguments and then passes the results of each of those to the binary function. Its implementation is shown below in Figure 3.74.

The two examples in Figure 3.75 are calculating the difference in sizes of two strings (`NDArrays` of characters) and checking to see if two strings are anagrams of each other. This composition pattern shows up all over the place once you start to look for it.

```
NDArray >> and: anNDArray with: f over: g
   "This is also known as the Psi combinator,
   'on' in Haskell and 'over' in APL and BQN"
   f numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
   g numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
   ^ f value: (g value: self) value: (g value: anNDArray)
```

Figure 3.74: `NDArray >> and:with:over:` implementation.

```
'horse' and: 'cat' with: #- over: #size.          "2"
'horse' and: 'cat' with: #matches: over: #sort.   "false"
'horse' and: 'shore' with: #matches: over: #sort. "true"
```

Figure 3.75: `NDArray >> and:with:over:` example usage.

`NDArray >> and:with:over:`

The B_1 is the sibling combinator of the Ψ combinator because it "reverses" how you apply the binary and unary functions. This composition pattern here is to evaluate the binary function first with the two arguments provided and then to pass the result of that operation to the unary function. The implementation is shown below in Figure 3.76.

```
NDArray >> and: anNDArray with: f atop: g
   "This is known as the B1 combinator from Combinatory
   Logic and the 'blackbird' from Combinator Birds"
   f numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
   g numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
   ^ f value: (g value: self value: anNDArray)
```

Figure 3.76: `NDArray >> and:with:atop:` implementation.

Figure 3.77 is testing whether two vectors are disjoint.

```
#(1 2 3) and: #(2 3 4) with: #isEmpty atop: #intersection:. "false"
#(1 2 3) and: #(4 5 6) with: #isEmpty atop: #intersection:. "true"
```

Figure 3.77: `NDArray >> and:with:atop:` example usage.

`NDArray >> and:with:hook:`

The D combinator is a generalization of the S combinator, as shown in Section 2.4 Combinator Specializations. Its implementation is shown in Figure 3.78.

```
NDArray >> and: anNDArray with: f hook: g
   "This is known as the D combinator from Combinatory
   Logic and the 'dove' from Combinator Birds"
   f numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
   g numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
   ^ f value: self value: (g value: anNDArray)
```

Figure 3.78: `NDArray >> and:with:hook:` implementation.

`NDArray >> and:with:hook:` is useful anytime you want to use a result of an operation from one `NDArray` on another. In Figure 3.79, the shape of `vectorB` is used to `NDArray >> reshape:` another `NDArray`. The D combinator is the same as the S combinator but instead of duplicating an `NDArray`, another `NDArray` is specified.

```
| vectorA vectorB |
vectorA := 8 iota asNDArray.
vectorB := 3 iota asNDArray.

"vectorA is 1 2 3 4 5 6 7 8
 vectorB is 1 2 3"

vectorA and: vectorB with: #reshape: hook: #shape. "1 2 3"
```

Figure 3.79: `NDArray >> and:with:hook:` example usage.

`NDArray >> and:with:backHook:`

This "combinator" doesn't actually correspond to a named combinator in combinatory logic literature. However, it does exist in BQN under the name *(dyadic) before*[38]. It is the sibling combinator to the D combinator. `NDArray >> and:with:backHook:` has the same composition pattern as the `NDArray >> and:with:hook:` but instead of applying the unary function to the right argument of the binary function, it is applied to the left argument. The implementation is shown in Figure 3.80.

The example in Figure 3.81 first ravels `matrix` and then removes the odd elements from the now rank-1 `NDArray` by using `NDArray >> without:`.

```
NDArray >> and: anNDArray with: f backHook: g
   "This is known as (dyadic) before in BQN."
   f numArgsAsBlock = 1 ifFalse: [ ^ ArityError signal ].
   g numArgsAsBlock = 2 ifFalse: [ ^ ArityError signal ].
   ^ g value: (f value: self) value: anNDArray
```

Figure 3.80: `NDArray >> and:with:backHook:` implementation.

```
| vector matrix |
vector := 4 iota asNDArray * 2 - 1.
matrix := 8 iota reshape: #(3 4).

"vector is 1 3 5 7
 matrix is 1 2 3 4
           5 6 7 8"

matrix and: vector with: #ravel backHook: #without:. "2 4 6 8"
```

Figure 3.81: `NDArray >> and:with:backHook:` example usage.

3.5.2 `Symbol` Combinators

Now that we have taken a look at some `NDArray` combinators, let's take a look at their equivalents in the `Symbol` combinators. Below in Table 3.2 is a comprehensive list of the `Symbol` combinators implemented in `Pharo-NDArray`.

Message	Arity	Combinator	APL	BQN	J
`<|`	Monadic	B	atop	atop	atop
`|>`	Monadic	Q	_	_	_
`<*>`	Monadic	S & _	_ & _	before & after	hook & _
`<->`	Dyadic	D & _	_ & beside	before & after	hook & _
`<|>`	Dyadic	B_1 & Ψ	atop & over	atop & over	atop & over

Table 3.2: `Symbol` combinator method names.

We will come back to the B and Q combinators and first look at the others.

`Symbol >> <*>`

This combinator is both of the following `Pharo-NDArray` combinators in one:

- `NDArray >> dupWith:hook:`

- NDArray >> dupWith:backHook:

In the Symbol combinators, we are able to "overload" based on the arity of the functions passed in. For Symbol >> <*>, NDArray >> dupWith:hook: corresponds to when the binary function is on the left of <*> and NDArray >> dupWith:backHook: corresponds to when the binary function is on the right. The implementation is shown below in Figure 3.82. Note that <*> comes from Haskell.

```
Symbol >> <*> aBlock
   | a b |
   a := self numArgsAsBlock.
   b := aBlock numArgsAsBlock.
   (a + b = 3 and: (a min: b) > 0) ifFalse: [ ^ ArityError signal ].
   "This is the backHook from I"
   b = 2 ifTrue: [
      ^ [ :x | aBlock value: (self value: x) value: x ] ].
   "This is the S combinator also known as the Starling"
   ^ [ :x | self value: x value: (aBlock value: x) ]
```

Figure 3.82: Symbol >> <*> implementation.

The examples from the previous section have been reimplemented using Symbol >> <*> in Figure 3.83.

```
#matches: <*> #reverse value: 'tacocat'.        "true"
#matches: <*> #reverse value: 'tacodog'.        "false"
#filter: <*> (#% @@ 2) value: #(1 2 3 4 5).     "1 3 5"
#* @@ 2 <*> #intersection: value: #(1 2 3 4 5). "2 4"
```

Figure 3.83: Symbol >> <*> example usage.

"Overloading" on arity is something that is not possible in array languages because functions/verbs are both unary and binary at the same time. In many tacit definitions of functions in array languages, you are actually defining two functions: one monadic and one dyadic. An example in APL is shown below.

Symbol >> <|>

This combinator is both of the following Pharo-NDArray combinators in one:

- NDArray >> and:with:over:

```
      ⍝ -4 + iota 7
      vec ← ¯4+⍳7
      vec
¯3 ¯2 ¯1 0 1 2 3
      ⍝ define a tacit function
      f ← +/⍤|
      ⍝ in the monadic case, absolute value and then sum
      ⍝ | vec = 3 2 1 0 1 2 3
      ⍝ +/ | vec = 12
      f vec
12
      ⍝ in the dyadic case, modulus 2 and then sum
      ⍝ 2 | vec = 1 0 1 0 1 0 1
      ⍝ +/ 2 | vec = 4
      2 f vec
4
```

Figure 3.84: Tacit function with a monadic and dyadic meaning in APL.

- `NDArray >> and:with:atop:`

For `Symbol >> <|>`, `NDArray >> and:with:over:` corresponds to when the binary operation is on the left and `NDArray >> and:with:atop:` corresponds to when the binary operation is on the right. The implementation is shown in Figure 3.85. The `<|>` symbol is chosen because it slightly resembles a mountain (making it fit with the names "over" and "atop") and is similar to `<*>`.

```
Symbol >> <|> aBlock
   | a b |
   a := self numArgsAsBlock.
   b := aBlock numArgsAsBlock.
   (a + b = 3 and: (a min: b) > 0) ifFalse: [ ^ ArityError signal ].
   "Psi or 'over'"
   a = 2 ifTrue: [
      ^ [ :x :y |
        self value: (aBlock value: x) value: (aBlock value: y) ] ].
   "B1 or 'atop'"
   ^ [ :x :y | self value: (aBlock value: x value: y) ]
```

Figure 3.85: `Symbol >> <|>` implementation.

The examples from the previous section are reimplemented using `Symbol >> <|>` in Figure 3.86.

```
#- <|> #size value: 'horse' value: 'cat'.          "2"
#matches: <|> #sorted value: 'horse' value: 'cat'.   "false"
#matches: <|> #sorted value: 'horse' value: 'shore'. "true"

#isEmpty <|> #intersection: value: #(1 2 3) value: #(2 3 4). "false"
#isEmpty <|> #intersection: value: #(1 2 3) value: #(4 5 6). "true"
```

Figure 3.86: `Symbol >> <|>` example usage.

`Symbol >> <->`

This combinator is both of the following `Pharo-NDArray` combinators in one:

- `NDArray >> and:with:hook:`
- `NDArray >> and:with:backHook:`

For `Symbol >> <->`, `NDArray >> and:with:hook:` corresponds to when the binary function is on the left and `NDArray >> and:with:backHook:` corresponds to when the binary function is on the right. The implementation is shown in Figure 3.87. The `<->` symbol is chosen because it slightly resembles a combination of the two symbols for *after* (⥽) and *before* (⥼) from BQN.

```
Symbol >> <-> aBlock
   | a b |
   a := self numArgsAsBlock.
   b := aBlock numArgsAsBlock.
   (a + b = 3 and: (a min: b) > 0) ifFalse: [ ^ ArityError signal ].
   "This is the D combinator also known as the Dove bird"
   a = 2 ifTrue: [
      ^ [ :x :y | self value: x value: (aBlock value: y) ] ].
   ^ [ :x :y | aBlock value: (self value: x) value: y ]
```

Figure 3.87: `Symbol >> <->` implementation.

The examples from the previous section are reimplemented using `Symbol >> <|>` in Figure 3.88.

```
| vectorA vectorB |
vectorA := 8 iota asNDArray.
vectorB := 3 iota asNDArray.

"vectorA is 1 2 3 4 5 6 7 8
 vectorB is 1 2 3"

#reshape: <-> #shape value: vectorA value: vectorB. "1 2 3"

| vector matrix |
vector := 4 iota asNDArray * 2 - 1.
matrix := 8 iota reshape: #(2 4).
"vector is 1 3 5 7
 matrix is 1 2 3 4
           5 6 7 8"

#ravel <-> #without: value: matrix value: vector. "2 4 6 8"
```

Figure 3.88: `Symbol >> <->` example usage.

`Symbol >> |>`

This is the B combinator and it corresponds to `NDArray >> with:atop:`. `Symbol >> |>` composes unary functions together from left to right. Its implementation is shown in Figure 3.89.

```
Symbol >> |> aBlock
   "This is the B combinator also known as the Bluebird"
   (self numArgsAsBlock = 1 and: aBlock numArgsAsBlock = 1)
      ifFalse: [ ^ ArityError signal ].
   ^ [ :x | aBlock value: (self value: x) ]
```

Figure 3.89: `Symbol >> |>` implementation.

A simplified example seen previously in the `NDArray >> partition:` implementation is shown in Figure 3.90. Combined with `@@` and `:>` from `Pharo-Functional`, we can really start to see the power of combinators and composition patterns.

```
| points |
points := 6 iota with: #(1 1 0 1 1 0) collect: #@.

"points is  {(1@1). (2@1). (3@0). (4@1). (5@1). (6@0)}"

points groupByRuns: #y
   :> select: #first |> #y |> #asBoolean
   :> collect: #collect: @@ #x
"[[1 2] [4 5]]"
```

Figure 3.90: `Symbol >> |>` example usage.

`Symbol >> <|`

This is the Q combinator and it actually doesn't exist in array languages. However, it is useful in Smalltalk because it eliminates certain cases where you might need to parenthesize otherwise. `Symbol >> <|` composes unary functions together from *right to left*. Its implementation is shown in Figure 3.91.

```
Symbol >> <| aBlock
   "This is the Q combinator known as the Queer Bird"
   (self numArgsAsBlock = 1 and: aBlock numArgsAsBlock = 1)
      ifFalse: [ ^ ArityError signal ].
   ^ [ :x | self value: (aBlock value: x) ]
```

Figure 3.91: `Symbol >> <|` implementation.

A simplified example seen previously in the `NDArray >> indices` implementation is shown in Figure 3.92. If we were to try and use `Symbol >> |>` here, we would end up with `#x |> (#= @@ 1)` which is less expressive.

```
| vector |
vector := #(0 1 0 0 1).
vector collectWithIndex: #@
   :> select: #= @@ 1 <| #x
   :> collect: #y
"2 5"
```

Figure 3.92: `Symbol >> <|` example usage.

Chapter 4

Results

This chapter will discuss the results of the `Pharo-NDArray` design laid out in Chapter 3 and as well as contributions and observations made during the research done for the background in Chapter 2.

4.1 Contributions to Combinatory Logic

The brief history of combinatory logic and how combinators exist in array languages is in itself a contribution to the existing combinatory logic literature because:

1. A comprehensive history detailing when individual combinators were first discovered does not exist;

2. There is very little mention of the influence that combinatory logic has had on the modern array language programming paradigm (the one explicit mention is 1989 "*Phrasal Forms*" paper[41]).

However, there are other contributions and observations that were made.

4.1.1 Missing Combinators

In designing a full set of combinators in `Pharo-NDArray`, specifically every combination of one unary and one binary function, it became clear that there are some missing combinators from combinatory logic literature. In Table 4.1, the combinators with no name in the left column are the ones that are missing.

Combinator	`NDArray`	`Symbol`	**BQN**
C	`flip:with:`		`˜`
B	`with:atop:`	`|>`	`∘`
B_1	`and:with:atop:`	`<|>`	`∘`
	`and:with:backHook:`	`<->`	`⊸`
Φ_1	`and:with:fork:and:`		3-train
D	`and:with:hook:`	`<->`	`⊸`
Ψ	`and:with:over:`	`<|>`	`○`
W	`dupWith:`		`˜`
	`dupWith:atop:`		`f∘g˜`
	`dupWith:backHook:`	`<*>`	`⊸`
Φ	`dupWith:fork:and:`		3-train
S	`dupWith:hook:`	`<*>`	`⊸`
	`dupWith:over:`		`f○g˜`

Table 4.1: Missing combinators.

It is curious that these combinators have been left out of combinatory logic literature. Note that all the combinators listed in Table 4.1 can be spelled in BQN and only two of the spellings require a composition of combinators. Below in 4.2 are names for the missing combinators. Δ and Σ are the *sibling* combinators of D and S respectively. This is because D and S are specializations of D_2 and Φ where the **right** function argument is set to the I combinator. Δ and Σ are specializations of D_2 and Φ where the **left** function argument is set to the I combinator. Δ and Σ are chosen as they are the capital Greek D and S. H_1 and H_2 are chosen for the other two missing combinators as placeholder names.

Combinator	`NDArray`	`Symbol`	**BQN**
C	`flip:with:`		`˜`
B	`with:atop:`	`|>`	`∘`
B_1	`and:with:atop:`	`<|>`	`∘`
Δ	`and:with:backHook:`	`<->`	`⊸`
Φ_1	`and:with:fork:and:`		3-train
D	`and:with:hook:`	`<->`	`⊸`
Ψ	`and:with:over:`	`<|>`	`○`
W	`dupWith:`		`˜`
H_1	`dupWith:atop:`		`f∘g˜`
Σ	`dupWith:backHook:`	`<*>`	`⊸`
Φ	`dupWith:fork:and:`		3-train
S	`dupWith:hook:`	`<*>`	`⊸`
H_2	`dupWith:over:`		`f○g˜`

Table 4.2: Δ, Σ, H_1 and H_2 combinators.

As well as the updated combinator table shown in Table 4.2, Table 4.3 shows an updated list of combinators and their corresponding 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)
Σ	λabc.a(bc)c
D	λabcd.ab(cd)
Δ	λabcd.a(bc)d
B_1	λabcd.a(bcd)
H_1	λabcd.a(bcc)
Ψ	λabcd.a(bc)(bd)
H_2	λabcd.a(bc)(bc)
Φ	λabcd.a(bd)(cd)
D_2	λabcde.a(bd)(ce)
E	λabcde.ab(cde)
Φ_1	λabcde.a(bde)(cde)
$\hat{E}$	λabcdefg.a(bde)(cfg)
Y	λa.a(λa)

Table 4.3: Combinators and lambda expressions with Δ, Σ, H_1 and H_2.

4.1.2 Combinator Specializations

The combinator hierarchies that were presented in Figure 2.2 and Figure 2.3 have been combined into Figure 4.1 below. These hierarchies and the relationships implied by them are important because it enables one to determine a spelling of a "missing" combinator in a specific language. For instance, Dyalog APL does not have support for the S combinator that exists in J as the *hook*. However, because S is a specialization of the Φ combinator which does exist in Dyalog as the *fork*, you are able to spell it by fixing the left monadic verb in the fork to be *same*, like this: `⊢fg`.

In my first two years of studying APL, I referred to the Φ combinator as the S′ combinator, which was David Turner's alternate name for it. This was because I wasn't aware that Φ was the original name. The advantage of the combinator name S′ is that it highlights the fact that S is a specialization of S′. However, most existing literature refers to this combinator as the Φ combinator and the benefit of staying consistent with most literature outweighs highlighting the relationship.

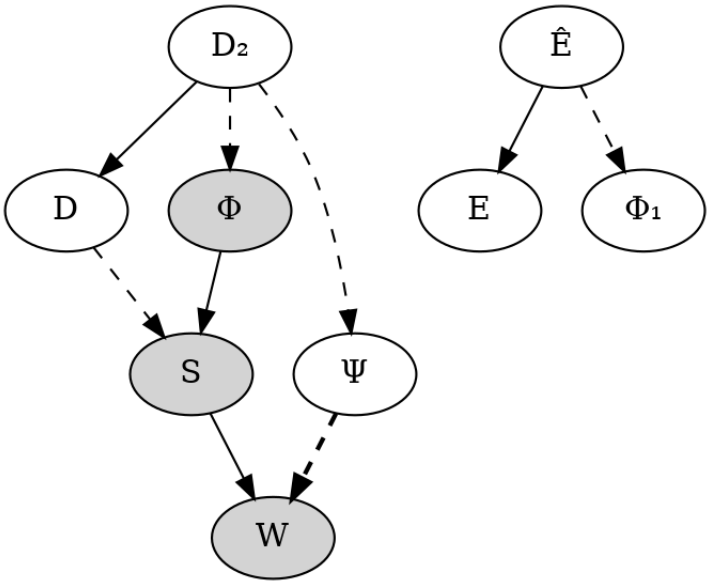

Figure 4.1: D_2 and $\hat{E}$ combinator hierarchies.

By combining the missing combinators from Table 4.2 with the D_2 combinator hierarchy in Figure 4.1, we get the resulting hierarchy shown in Figure 4.2.

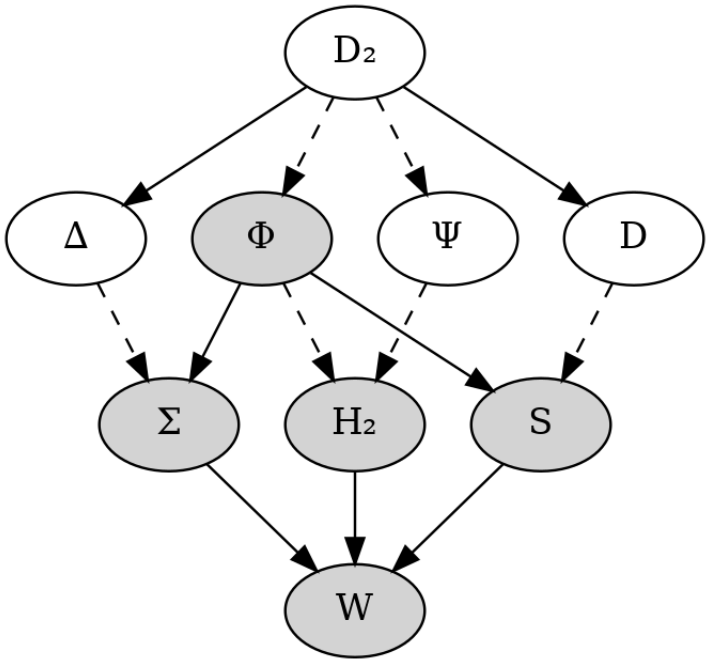

Figure 4.2: D_2 combinator hierarchy with Δ, Σ and H_2.

When juxtaposed with a hierarchy graph with the spellings of each combinator in BQN, we can see that the H_2 combinator is missing. This can be seen in Figure 4.3.

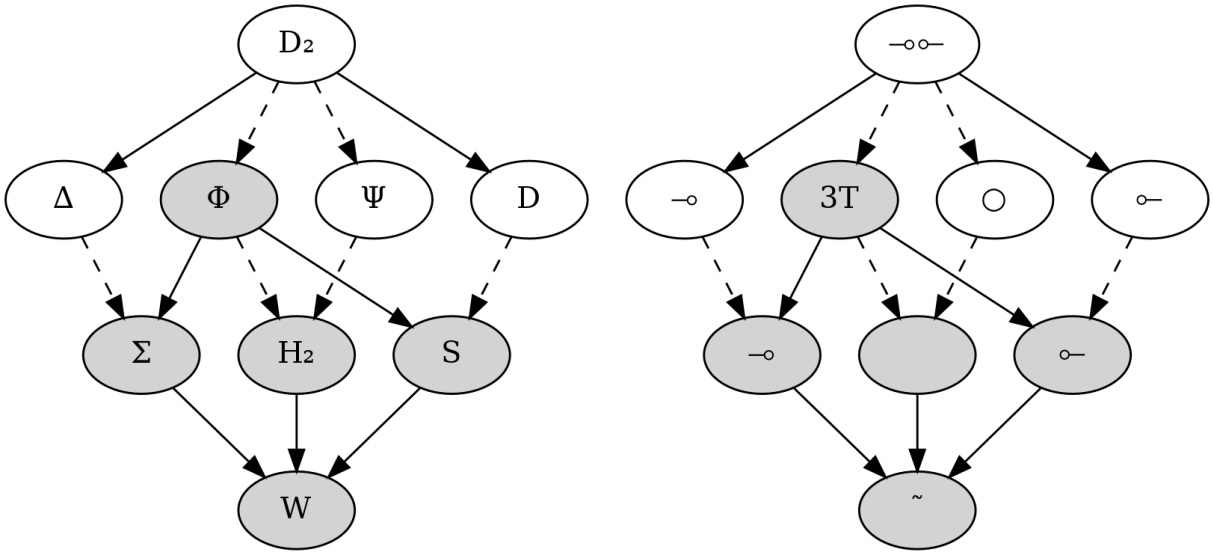

Figure 4.3: D_2 combinator hierarchy vs BQN spellings.

This arguably means that a mistake was made in BQN (and other array languages) for choosing the B combinator to be the monadic meaning of the Ψ combinator (known as *over* in BQN). Note that the B_1 and Ψ combinator (*atop* and *over* respectively in BQN) chose their monadic definitions to be the B combinator, meaning there are two ways to spell one combinator. This definitely makes it seem like an oversight when you could have used the monadic spelling of *over* to be the H_2 combinator which not only would provide a way to spell a missing combinator, it would also make *over* more consistent with *before* and *after* (the Δ and D combinators) in that both of monadic meanings are "dotted line" specializations.

4.1.3 Relationship between Ψ and B_1

In designing the `Symbol` combinators as a part of `Pharo-NDArray`, I discovered the relationship between Ψ and B_1 which made them perfect to combine in the overloaded `Symbol >> <|>`. If we take a look at the two Python implementations of Ψ and B_1 again, we can see clearly that they are "*opposites*" or "*arity symmetric*".

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

```
def b1(f, g):
    return lambda x, y: f(g(x, y))
```

We can tell they are opposites in that for Ψ, `f` is the binary function and `g` is the unary function and for B_1 it is the reverse. Ψ applies the unary function first to both arguments, followed by the binary function whereas B_1 applies the binary function first to both arguments, followed by the unary function. Until designing `Symbol >> <|>`, I never noticed this relationship.

4.2 `NDArray` Methods

Smalltalk as a programming language is a very good fit for implementing an `NDArray` type and rank-polymorphic methods. The rich set of functions/methods that come with Smalltalk, some commonly used ones of which are listed below, make implementing the `NDArray` methods quite easy.

- `SequenceableCollection >> collect:`
- `SequenceableCollection >> collectWithIndex:`
- `SequenceableCollection >> groupByRuns:`
- `SequenceableCollection >> groupsOf:`
- `SequenceableCollection >> select:`
- `SequenceableCollection >> with:collect:`

Implementing a selection of the common verbs from array languages uncovered several categories that many of the verbs fit into.

- Scalar verbs
- Axis-last verbs
- Major-cell verbs

Examples of scalar verbs were covered in Section 3.3.1 on Scalar Monadic Verbs and Section 3.3.2 on Scalar Dyadic Verbs. Axis-last verbs were covered in several subsections of Section 3.3 on `NDArray` Methods. Finally, both of the major-cell verbs were covered in

Section 3.3.3 on Monadic Verbs. The categories map exactly to a subset of the list that can be found on the APL Wiki documentation for Leading Axis Theory[35]. Leading Axis Theory was not intentionally taken into account when designing `Pharo-NDArray` but further development would most likely be aided by an in depth exploration of the topic.

4.3 `NDArray` and `Symbol` Combinators

The following is a summary of the observations made about both the `NDArray` and `Symbol` combinators.

- Verbosity of `NDArray` combinators
- Symmetry in naming of `NDArray` combinators
- Ergonomics of `Symbol` combinators
- Arity overloading in `Symbol` combinators
- Precedence of `Symbol` combinators over `NDArray` combinators

Each of these observations will be discussed below.

4.3.1 Verbosity of `NDArray` combinators

The first remark that must be made about the `NDArray` combinators is on the verbosity of their names. Compared to the combinators in APL and other array languages (and other programming languages for that matter), `NDArray` combinators are more verbose by a wide margin. `NDArray` combinator names (fully listed in Table 3.1) have an average combinator name length of 14 characters and the maximum is 18 characters. This is, in my opinion, overly cumbersome especially when compared to the alternative of the `Symbol` combinators.

In Section 2.7, the second most important reason why combinators are so powerful is that "they are spelled with one glyph or through juxtaposition." Having to use an average of 14 characters for each of the `NDArray` combinators means that this reason definitely does not apply to `NDArray` combinators.

4.3.2 Symmetry in naming of `NDArray` combinators

Although the verbosity of the `NDArray` combinator names is a disadvantage, one of the advantages of the `NDArray` combinator names is the symmetry in the names. It is explicitly clear from the names of the `NDArray` combinators (listed in Table 3.1) which ones are monadic and dyadic. The dyadic combinators all start with `and:` and the monadic combinators do not. Furthermore, any combinator that "duplicates" one of the arguments thereby using it in multiple places has a `dupWith:` in the name of the combinator. This symmetry is an advantage because it reveals relationships between combinators that are less explicitly clear from how they manifest in array languages or other programming languages like Haskell. It was never obvious to me that the S combinator, `NDArray >> dupWith:hook:`, had an implicit W combinator in it until the symmetry in the naming of the `NDArray` combinators revealed it.

This naming symmetry of `NDArray` combinators also made it clear what the full list of combinators should be. For example, every monadic `NDArray >> dupWith:xyz:` combinator should have a corresponding dyadic `NDArray >> and:with:xyz:` combinator. Implementing a comprehensive list of combinators for `NDArray` identified some missing combinators from combinatory logic literature.

4.3.3 Ergonomics of `Symbol` combinators

As for the `Symbol` combinators, the spelling is the opposite of cumbersome as each of the `Symbol` combinators are either two or three characters. However, because each of the combinators returns a `FullBlockClosure`, invoking the results of these combinators requires a `value:` in the monadic case and a `value:value:` in the dyadic case. The only times that this can be avoided is when passing the result of a `Symbol` combinator to another function that takes a `FullBlockClosure` as an argument. This is not the most frequent way that the `Symbol` combinators will be used, so it is a win-lose situation. When it comes to the ergonomics of `Symbol` combinators, the advantage is that they are spelled with only two or three characters. However, this is offset by the tax of the `value:(value:)` required to invoke them. There is currently a *Proposed Syntax Extension*[40] to `Pharo-Functional` that would remove this disadvantage. This is discussed in more detail in Section 5.2.1.

4.3.4 Arity overloading in `Symbol` combinators

Another advantage of `Symbol` combinators is that you can "overload" them based on the arity of the functions that are passed on the left and right of the binary messages.

This technically can be done with `NDArray` combinators as well but it is less natural due to the fact that keyword messages have zero symmetry in their individual spelling, whereas each of the three `Symbol` combinators that take a unary and binary function as arguments:

- `NDArray >> <*>`
- `NDArray >> <|>`
- `NDArray >> <->`

are not only vertically symmetric but the composition patterns are reflected by which side of the `Symbol` combinators the unary and binary messages go. For instance, for the `NDArray >> <*>` combinator, when the unary function goes on the left of the combinator, it is applied to the *left* argument and vice versa.

It should be once again noted as it was in Section 3.5.2 on Symbol Combinators that "overloading" on arity is something that is not possible in array languages because functions/verbs are both unary and binary at the same time. This makes this an advantage of implementing combinators in a programming language like Smalltalk.

4.3.5 Precedence of `Symbol` combinators over `NDArray` combinators

A significant advantage of implementing combinators in a language like Smalltalk is that binary messages have higher precedence than keyword messages and therefore we are closer to the combinator precedence that array languages have when compared to other languages like Haskell. In Section 2.7, the most important reason for combinators being so powerful in array languages is that "they have higher precedence than function application." Section 2.7.1 on Combinator Precedence goes on to say that:

> 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.

The fact that function application in Haskell has a precedence level of 10 and the maximum of an infix operator is 9 means it is impossible to write code in Haskell that is possible in Smalltalk or array languages.

It is also fascinating that there is no strong delineation between "function types" in most languages. Array languages like APL and J have categories of "verbs" and "adverbs" where most languages would just say "functions" and "higher order functions." However, most programming languages do not actually distinguish between functions and higher order functions, it is just a naming difference used to explain a type of function that can take functions as arguments or returns them. Both of these functions are typically treated as equivalent from the point of view of the compiler or the interpreter.

4.4 `Pharo-NDArray` vs Smalltalk

Finally, it is worth looking at an example in Smalltalk without making use of `Pharo-NDArray`. Below are two different solutions in Figures 4.4 and 4.5 to the "all adjacent differences" problem that has been shown multiple times in this thesis (in Section 2.1.4 and Section 3.1.6). Figure 4.4 shows a Smalltalk solution that makes use of no built-in algorithms and just makes use of the `Set` collection.

```
| vec res |
vec := #(1 -5 3 -8 6).
res := Set new.
1 to: vec size do: [ :i |
    i to: vec size do: [ :j |
        res add: ((vec at: i) - (vec at: j)) abs
    ].
].

"res is a Set(0 9 11 5 3 14 8 6 2)"
```

Figure 4.4: All absolute differences in Smalltalk (1).

Figure 4.5 makes use of:

- `SequenceableCollection >> combinationsOf:`
- `SequenceableCollection >> collect:`
- `Collection >> copyWithoutDuplicates`

However, `SequenceableCollection >> combinationsOf:` doesn't "combine with replacement" so a zero must be appended to account for subtracting an element from itself.

```
| vec res |
vec := #(1 -5 3 -8 6).
res := vec combinationsOf: 2
    :> collect: [ :e | (e first - e second) abs ]
    :> copyWithoutDuplicates
    :> , #(0)

"res is #(8 3 11 6 14 9 2 5 0)"
```

Figure 4.5: All absolute differences in Smalltalk (2).

Compared to the `Pharo-NDArray` solution shown in Section 3.1.6 which has been duplicated below in Figure 4.6, more detail and complexity is required when not using `Pharo-NDArray`. Specifically, note how no blocks or element accessing using `at:`, `first` or `second` is needed. Maximum detail is suppressed so the important aspects of the solution are highlighted.

```
| vec res |
vec := #(1 -5 3 -8 6).
res := vec triangleProduct2: #-
    :> abs
    :> enlist
    :> unique

"res is #(0 6 2 9 5 8 3 11 14)"
```

Figure 4.6: All absolute differences in Smalltalk using `Pharo-NDArray`.

Chapter 5

Conclusions

My thesis is that programming in array languages with combinators (from Combinatory Logic) is a powerful and important aspect of programming and that it can naturally fit into an object-oriented language like Smalltalk. The contributions are enumerated in Section 5.1 below.

5.1 Contributions

The main contributions in this thesis can be broken up into two categories:

- Contributions to/from combinatory logic literature:
  - Surveying the history of combinatory logic and combinators in array languages, specifically Dyalog APL, J and BQN;
  - Identifying the relationships between existing combinators in the form of hierarchical specializations;
  - Identifying and naming missing combinators from combinatory logic literature;
  - Identifying the relationship between the B_1 and the Ψ combinators.
- Contributions to/from Smalltalk:
  - Designing and implementing the `NDArray` type, the rank-polymorphic `NDArray` methods, the `NDArray` combinators and the `Symbol` combinators;
  - Providing motivation for a new syntax in Pharo Smalltalk that improves the ergonomics of `FullBlockClosure` invocation.

5.2 Future Work

The potential future work to follow the experimental library implementation completed as a part of this thesis are listed below.

- The *Proposed Syntax Extension*[40] (discussed below in Section 5.2.1) could be implemented to remove the one disadvantage of the `Symbol` combinators implemented in `Pharo-NDArray`.
- A continuation of the "completion" and "naming" of the missing combinators identified in Section 4.1.1 as well other missing combinators yet to be identified.
- An investigation into the advantages and disadvantages of using the H_2 combinator instead of the B combinator as the monadic definition of *over* in BQN, APL or a new array language (as discussed in Section 4.1.2).
- A continuation of the combinator hierarchies that incorporate the Elementary Combinators discussed in Table 2.2 in Section 2.3.6
- An exploration of Leading Axis Theory[35] should be conducted and the results folded into `Pharo-NDArray`.
- Currently, the `data` member of the `NDArray` type is stored as an `Array`. An investigation into the possibility of replacing this with a `FloatArray` should be conducted. This could potentially enable offloading much of the computation to accelerated devices such as GPUs which are tailor-made for array intensive computations.
- A further investigation into the implications of function application and the precedence of function application should be conducted to see the full impact that this has on use of combinators in programming languages.
- An investigation into why most programming languages do not delineate between functions and high order functions the way that array languages do should be conducted.

5.2.1 Future Pharo Smalltalk Syntax

There is currently a *Proposed Syntax Extension*[40] to `Pharo-Functional` that would obviate the necessity of calling `value:` in order to invoke the `FullBlockClosure`s. The examples from Section 3.5.2 on Symbol Combinators would drastically improve as shown in Figure 5.1.

```
#matches: <*> #reverse value: 'tacocat'. "Before"
'tacocat' (#matches: <*> #reverse).      "After"
"true"

#matches: <*> #reverse value: 'tacodog'. "Before"
'tacodog' (#matches: <*> #reverse).      "After"
"false"

#- <|> #size value: horse value: cat. "Before"
horse (#- <|> #size): cat.            "After"
"2"

#matches: <|> #sort value: 'horse' value: 'cat'. "Before"
'horse' (#matches: <|> #sort): 'cat'.            "After"
"false"

#matches: <|> #sort value: 'horse' value: 'shore'. "Before"
'horse' (#matches: <|> #sort): 'shore'.            "After"
"true"
```

Figure 5.1: *Proposed Syntax Extension* example usage.

As the *Proposed Syntax Extension*[40] explains, an alternative to using `([...] value: x)` to invoke a `FullBlockClosure` that takes a single argument is to use `x (...)` as a shortcut. Similarly, an alternative to using `([...] value: x value: y)` to invoke a `FullBlockClosure` that takes two arguments is to use `x (...): y` as a shortcut. If the *Proposed Syntax Extension* is accepted, it will eliminate the one disadvantage of the `Symbol` combinators and make them extremely ergonomic to use.

Appendix A: `Pharo-NDArray` Verbs and Adverbs

This is a full list of the verbs and adverbs implemented in `Pharo-NDArray` as of May 2022.

Monadic Scalar	**Dyadic Scalar**	**Monadic**	**Dyadic**	**Adverbs**
abs	%	all	drop:	collect
ceiling	*	any	filter:	outerProduct:with:
exp	/	asString	filterPred:	reduce:
factorial	+	first	gather:	reduceFirst:
floor	-	enlist	intersection:	scan:
invFactorial	**	indices	matches:	triangleProduct:
not	<	ints	memberOf:	triangleProduct2:
reciprocal	<=	invIota	notMatches:	upperProduct:
roll	>	invWhere	partition:	upperProduct2:
sign	>=	isEmpty	reshape:	windowed:reduce:
	=	last	rotate:	
	min:	max	take:	
	max:	maxs	without:	
		min		
		mins		
		mix		
		negated		
		product		
		ravel		
		reverse		
		size		
		sort		
		split		
		sum		
		sums		
		transpose		
		unique		

Table 5.1: `Pharo-NDArray` Verbs and Adverbs.

Bibliography

[1] Alexander Aiken, John H Williams, and Edward L Wimmers. *The FL project: The design of a functional language*. Citeseer, 1994.

[2] *Array - Maple Help*. URL: https://www.maplesoft.com/support/help/maple/view.aspx?path=Array.

[3] *Array—Wolfram Language Documentation*. URL: https://reference.wolfram.com/language/ref/Array.html.

[4] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. *Numerical python*. 2001.

[5] John Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Communications of the ACM* 21.8 (1978). Publisher: ACM New York, NY, USA, pp. 613–641.

[6] John Backus. "From function level semantics to program transformation and optimization". In: *Colloquium on Trees in Algebra and Programming*. Springer, 1985, pp. 60–91.

[7] Katalin Bimbó. *Combinatory Logic (Stanford Encyclopedia of Philosophy)*. 2008. URL: https://plato.stanford.edu/entries/logic-combinatory/.

[8] Adám Brudzewsky. *APLcart - Find your way in APL*. URL: https://aplcart.info/.

[9] Adám Brudzewsky. *dyalog-apl-extended: Dyalog APL Extended*. URL: https://github.com/abrudz/dyalog-apl-extended.

[10] *C++ mdarray: An Owning Multidimensional Array Analog of mdspan*. URL: https://isocpp.org/files/papers/D1684R0.html.

[11] *Chapter 2: Objects and Messages*. URL: http://esug.org/data/Old/ibm/tutorial/CHAP2.HTML#2.10.

[12] Haskell B Curry. “A simplification of the theory of combinators”. In: *Synthese* (1948). Publisher: JSTOR, pp. 391–399.

[13] Haskell B Curry. “An analysis of logical substitution”. In: *American journal of mathematics* 51.3 (1929). Publisher: JSTOR, pp. 363–384.

[14] Haskell B Curry. “The combinatory foundations of mathematical logic”. In: *The Journal of Symbolic Logic* 7.2 (1942). Publisher: Cambridge University Press, pp. 49–64.

[15] Haskell B. Curry. “A Revision of the Fundamental Rules of Combinatory Logic”. In: *The Journal of Symbolic Logic* 6.2 (1941). Publisher: Association for Symbolic Logic, pp. 41–53. ISSN: 00224812. URL: http://www.jstor.org/stable/2266655 (visited on 2022-04-04).

[16] Haskell Brooks Curry. “Grundlagen der kombinatorischen Logik”. In: *American journal of mathematics* 52.4 (1930). Publisher: JSTOR, pp. 789–834.

[17] Haskell Brooks Curry. “Some additions to the theory of combinators”. In: *American Journal of Mathematics* 54.3 (1932). Publisher: JSTOR, pp. 551–558.

[18] Haskell Brooks Curry. “The universal quantifier in combinatory logic”. In: *Annals of Mathematics* (1931). Publisher: JSTOR, pp. 154–180.

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

[20] *Dyalog APL - APL Wiki*. URL: https://aplwiki.com/wiki/Dyalog_APL.

[21] *Family tree of array languages - APL Wiki*. URL: https://aplwiki.com/wiki/Family_tree_of_array_languages.

[22] *First Derivatives completes $53.8m acquisition of Kx Systems*. URL: https://www.irishtimes.com/business/technology/first-derivatives-completes-53-8m-acquisition-of-kx-systems-1.3929204.

[23] *GNU Octave: NDArray Class Reference*. URL: http://octave.org/doxygen/4.0/d3/dd5/classNDArray.html.

[24] Stuart Halloway. *Ending Legacy Code In Our Lifetime*. 2008-04. URL: https://cognitect.com/blog/2008/4/1/ending-legacy-code-in-our-lifetime.

[25] Conor Hoekstra. *Algorithms as a Tool of Thought // APL Seeds '21*. URL: https://www.youtube.com/watch?v=GZuZgCDql6g.

[26] Conor Hoekstra. “Combinatory Logic and Combinators in Array Languages [submitted for publication]”. In: 2022.

[27] Conor Hoekstra. *Pharo-NDArray: Library for n-dimensional arrays and common array programming language algorithms and combinators.* URL: https://github.com/codereport/Pharo-NDArray.

[28] Conor Hoekstra and Adam Gordon Bell. *From Competitive Programming to APL With Conor Hoekstra - CoRecursive Podcast.* 2022-06. URL: https://corecursive.com/065-competitive-coding-with-conor-hoekstra/.

[29] Conor Hoekstra and Adam Gordon Bell. *Trying APL With Conor Hoekstra - CoRecursive Interview.* 2022-06. URL: https://www.youtube.com/watch?v=lG-CcPb7ggU.

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

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

[32] Kenneth E Iverson. "A dictionary of APL". In: *ACM SIGAPL APL Quote Quad* 18.1 (1987). Publisher: ACM New York, NY, USA, pp. 5–40.

[33] Kenneth E. Iverson. *A programming language.* New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 0-471-43014-5. URL: http://portal.acm.org/citation.cfm?id=SERIES11430.1098666.

[34] *K - APL Wiki.* URL: https://aplwiki.com/wiki/K.

[35] *Leading axis theory - APL Wiki.* URL: https://aplwiki.com/wiki/Leading_axis_theory.

[36] *List of language developers - APL Wiki.* URL: https://aplwiki.com/wiki/List_of_language_developers.

[37] Marshall Lochbaum. *BQN for birdwatchers.* URL: https://mlochbaum.github.io/BQN/doc/birds.html.

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

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

[40] David Mason. *Pharo-Functional: Functional support for Pharo - Proposed Syntax Extensions.* URL: https://github.com/dvmason/Pharo-Functional#proposed-syntax-extensions.

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

[42] *Most Popular Python Packages in 2021*. URL: https://learnpython.com/blog/most-popular-python-packages/.

[43] *Multi-dimensional Arrays · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/arrays/.

[44] *ndarray - Rust*. URL: https://docs.rs/ndarray/0.14.0/ndarray/.

[45] *numpy.ndarray — NumPy v1.22 Manual*. URL: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html.

[46] *PYPL PopularitY of Programming Language index (April 2022)*. URL: https://web.archive.org/web/20220409185824/https://pypl.github.io/PYPL.html.

[47] *R array: Multi-way Arrays*. URL: https://rdrr.io/r/base/array.html.

[48] Chris Rathman. *Combinator Birds*. URL: https://www.angelfire.com/tx4/cus/combinator/birds.html.

[49] Henry Rich. *J for C Programmers*. 2002.

[50] Barkley Rosser. "Review of: The Combinatory Foundations of Mathematical Logic by Haskell B. Curry". In: *The Journal of Symbolic Logic* 8.1 (1943). Publisher: Association for Symbolic Logic, pp. 31–31. ISSN: 00224812. URL: http://www.jstor.org/stable/2267984 (visited on 2022-04-04).

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

[52] Moses Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Mathematische annalen* 92.3 (1924). Publisher: Springer, pp. 305–316.

[53] Amar Shah. *"Point-Free or Die: Tacit Programming in Haskell and Beyond"*. 2016-09. URL: https://www.youtube.com/watch?v=seVSlKazsNk.

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

[55] *The Futhark Programming Language*. URL: https://futhark-lang.org/.

[56] *The TIOBE Index for April 2022*. URL: https://web.archive.org/web/20220413234339/https://www.tiobe.com/tiobe-index/.

[57] *Timeline of APL dialects - APL Wiki*. URL: https://aplwiki.com/wiki/Timeline_of_APL_dialects.

[58] DA Turner. "SASL Language Manual". In: *Andrews University, Fife, Scotland* (1976).

[59] David A Turner. “Another algorithm for bracket abstraction”. In: *The Journal of Symbolic Logic* 44.2 (1979). Publisher: Cambridge University Press, pp. 267–270.

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

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

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