PFA: Portable Format for Analytics

Jim Pivarski

Sometime in 2014

Abstract

This specification defines the syntax and semantics of the Portable Format for Analytics (PFA).

PFA is a mini-language for mathematical calculations that is usually generated programmatically, rather than by hand. A PFA document is a string of JSON-formatted text that describes an executable called a scoring engine. Each engine has a well-defined input, a well-defined output, and functions for combining inputs to construct the output in an expression-centric syntax tree. In addition, it has centralized facilities for maintaining state, with well-defined semantics for sharing state among scoring engines in a thread-safe way. The specification defines a suite of mathematical and statistical functions for transforming data, but it does not define any means of communication with an operating system, file system, or network. A PFA engine must be embedded in a larger system that has these capabilities, and thus an analytic workflow is decoupled into a part that manages data pipelines (such as Hadoop or Storm), and a part that describes the algorithm to be performed on data (PFA).

PFA is similar to the Predictive Model Markup Language (PMML), an XML-based specification for statistical models, but whereas PMML's focus is on statistical models in the abstract, PFA's focus is on the scoring procedure itself. The same input given to two PFA-enabled systems must yield the same output, regardless of platform (e.g. a JVM in Hadoop, a client's web browser, a GPU kernel function, or even an IP core directly embedded in an integrated circuit). Unlike PMML, the PFA specification defines the exact bit-for-bit behavior of any well-formed document, the semantics of data types and data structures, including behavior in concurrent systems, and all cases in which an exception should be thrown. Like PMML, PFA is a specification, not an implementation, it defines a suite of statistical algorithms for analyzing data, and it is usually generated programmatically, as the output of a machine learning algorithm, for instance.

Status of this document

This section describes the status of this document at the time of the current draft. Other documents may supersede this document.

Contents

1 Introduction

1.1 Motivation for PFA

The Portable Format for Analytics (PFA) is a mini-language for mathematical calculations. It differs from most programming languages in that it is optimized for automatic code generation, rather than writing programs by hand. The primary use-case is to represent the output of machine learning algorithms, such that they can be freely moved between systems. Traditionally, this field has been dominated by special-purpose file formats, each representing only one type of statistical model. The Predictive Model Markup Language (PMML) provides a means of unifing the most common model types into one file format. However, PMML can only express a fixed set of pre-defined model types; new model types must be agreed upon by the Data Mining Group (DMG) and integrated into a new version of PMML, then that new version must be adopted by the community before it is widely usable.

PFA represents models and analytic procedures more generally by providing generic programming constructs, such as conditionals, loops, persistent state, and callback functions, in addition to a basic suite of statistical tools. Conventional models like regression, decision trees, and clustering are expressed by referencing the appropriate library function, just as in PMML, but new models can be expressed by composing library functions or passing user-defined callbacks. Most new statistical techniques are variants of old techniques, so a small number of functions with the appropriate hooks for inserting user code can represent a wide variety of methods, many of which have not been discovered yet.

Given this need for flexibility, one might consider using a general purpose programming language, such as C, Java, Python, or especially R, which is specifically designed for statistics. While this is often the easiest method for small problems that are explored, formulated, and solved on the analyst's computer, it is difficult to scale up to network-sized solutions or to deploy on production systems that need to be more carefully controlled than a personal laptop. The special-purpose code may depend on libraries that cannot be deployed, or may even be hard to identify exhaustively. In some cases, the custom code might be regarded as a stability or security threat that must be thoroughly reviewed before deployment. If the analytic algorithm needs to be deployed multiple times before it is satisfactory and each deployment is reviewed for reasons unrelated to its analytic content, development would be delayed unnecessarily. This problem is solved by decoupling the analytic workflow into a part that deals exclusively with mathematics (the PFA scoring engine) and the rest of the infrastructure (the PFA host). A mathematical algorithm implemented in PFA can be updated frequently with minimal review, since PFA is incapable of raising most stability or security issues, due to its limited access.

PFA is restricted to the following operations: mathematical functions on numbers, strings, raw bytes, homogeneous lists, homogeneous maps (also known as hash-tables, associative arrays, or dictionaries), heterogeneous records, and unions of the above, where mathematical functions include basic operations, special functions, data sturcture manipulations, missing data handling, descriptive statistics, and common model types such as regression, decision trees, and clustering, parameterized for flexibility. PFA does not include any means of accessing the operating system, the file system, or the network, so a rouge PFA engine cannot expose or manipulate data other than that which is intentionally funneled into it by the host system. The full PFA specification allows recursion and unterminated loops, but execution time is limited by a timeout. PFA documents may need to be reviewed for mathematical correctness, but they do not need to be reviewed for safety.

Another reason to use PFA as an intermediate model representation is for simplicity of code generation. A machine learning algorithm generates an executable procedure, usually a simple, parameterized decider algorithm that either categorizes or makes predictions based on new data. Although the parameters might be encoded in a static file, some component must be executable. A PFA document bundles the executable with its parameters, simplifying version control.

The syntax of PFA is better suited to automatic code generation than most programming languages.

Many languages have complex syntax to accomodate the way people think while programming, including infix operators, a distinction between statements and expressions, and in some cases even meaningful whitespace. Though useful when writing programs by hand, these features only complicate automatic code generation. A PFA document is an expression tree rendered in JSON, and trees are easy to programmatically compose into larger trees without introducing syntax errors in the generated code. This is well-known in the Lisp community, since the ease of writing code-modifying macros in Lisp is often credited to its exclusive use of expression trees, rendered as parenthesized lists (known as S-expressions). PFA uses JSON, rather than S-expressions, because libraries for manipulating JSON objects are more widely available and JSON provides a convenient syntax for maps, but the transliteration between JSON and S-expressions is straight-forward.

Another benefit of PFA's simplicity relative to general programming languages is that it is more amenable to static analysis. A PFA host can more thoroughly examine an incoming PFA document for undesirable features. Although PFA makes use of callback functions to tweak behavior, functions are not first-class objects in the language, meaning that they cannot be dynamically assigned to variables. The identity of every function call can be determined without running the engine, which makes it possible to statically generate a graph of function calls and identify recursive loops. In very limited runtime environments, such as some GPUs, the compiler implicitly inlines all function calls, so recursion is not possible. In cases like these, static analysis of the PFA document is a necessary step in generating the executable.

A PFA document can also be statically type-checked. This allows for faster execution times, since types do not need to be checked at run-time, but it also provides additional safety to the PFA host.

PFA uses Apache Avro schemae as type annotations. Avro is an open-source serialization protocol, widely used in Hadoop and related projects, whose type schemae are expressed as JSON objects and whose data structures can be expressed as JSON objects. Therefore, PFA control structures, Avro type annotations, and data structure literals are all expressed as a single nested JSON object. Avro additionally has well-defined rules to resolve different but possibly compatible schemae, which PFA reinterprets as type promotion (allowing integers to be passed to a function that expects floating-point numbers, for instance). When interpreted this way, Avro also has a type-safe null, which PFA uses to ensure that missing data are always explicitly handled. Finally, the input and output of every PFA engine can always be readily (de)serialized into Avro's binary format or JSON representation, since Avro libraries are available for a wide variety of platforms.

1.2 Terminology used in this Specification

Within this specification, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 (see [RFC2119]). However, for readability, these words do not appear in all uppercase letters in this specification.

At times, this specification provides hints and suggestions for implementation. These suggestions are not normative and conformance with this specification does not depend on their realization. These hints contain the expression "We suggest..." or similar wording.

This specification uses the terms "JSON object", "JSON object member name", "JSON object member value", "JSON array", "JSON array value", "number", "integer", and "string" as defined in the JSON specification (RFC-4627), sections 2.2 through 2.5. It also references and quotes sections of the Avro 1.7.6 specification (`http://avro.apache.org/docs/1.7.6/spec.html`).

1.3 PFA MIME Type and File Name Extension

The MIME type for PFA is "application/pfa+json". The registration of this MIME type is in progress.

It is recommended that PFA files have the extension ".pfa" (all lowercase) on all platforms. It is recom-

mended that gzip-compressed PFA files have the extension ".pfaz" (all lowercase) on all platforms.

1.4 Levels of PFA Conformance and PFA Subsets

2 PFA Document Structure

2.1 Input and Output Type Specification

2.2 Scoring Method: Map, Emit, and Fold

2.3 Execution Phases: Begin, Action, and End

some applications would only have a begin and action, but no end

2.4 Specification of Persistent and Shared State

type specification

initialization

2.5 Engine Options

randseed

overridable options

2.6 Engine Name, Documentation, and Metadata

3 Type System

3.1 Avro Types

Type-safe null

3.2 Type Inference

3.3 Type Resolution, Promotion, and Covariance

3.4 Function Parameter Patterns

4 Symbols, Scope, and Data Structures

4.1 Immutable Data, Reassignable Symbols

4.2 Expression-Level Scope and Mutation Restrictions

4.3 Data Structure Limitations

No circular references

String-only map keys

5 User-defined Functions

5.1 Syntax and Scope

5.2 Anonymous Callbacks and Function References

6 Persistent and Shared State

6.1 Cells and Pools

cells and pools (specification only: link to extraction and manipulation)

6.2 Concurrent Access and Manipulation of Shared State

7 Expressions

Special forms and ordinary function calls

7.1 Function Calls

7.2 Symbol References

7.3 Literal Values

7.4 Creating Arrays, Maps, and Records

7.5 Symbol Assignment and Reassignment

7.6 Extracting from and Updating Arrays, Maps, and Records

7.7 Extracting from and Updating Cells and Pools

7.8 Do blocks

7.9 Conditionals: if and cond

7.10 While loops: Pretest and Posttest

7.11 For loops: by Index, Array Element, and Key-Value

7.12 Type-Safe Casting

7.13 Inline Documentation

7.14 User-Defined Exceptions

7.15 Log Messages

8 Core Library and Basic Data Manipulation

```
{"s.translate": [s, oldchars, newchars]}
    s          String
    oldchars   String
    newchars   String
               → String
```

```
{"a.countPredicate": [a, predicate]}
    a           Array(Wildcard(A,Set()))
    predicate   Fcn(List(Wildcard(A,Set())),Boolean)
                → Int
```

```
{"a.argminN": [a, n]}
    a   Array(Wildcard(A,Set()))
    n   Int
        → Array(Int)
```

```
{"a.geomean": [a]}
    a   Array(Double)
        → Double
```

```
{"s.lower": [s]}
    s   String
        → String
```

```
{"model.tree.simpleWalk": [datum, treeNode]}
    datum      WildRecord(D,Map())
    treeNode   WildRecord(T,Map(fail -> Union(List(WildRecord(T,Map()), Wildcard(S,Set()))),
               field -> String, operator -> String, pass -> Union(List(WildRecord(T,Map()), Wild-
               card(S,Set()))), value -> Wildcard(V,Set())))
               → Wildcard(S,Set())
```

```
{"a.fold": [a, zero, fcn]}
    a      Array(Wildcard(A,Set()))
    zero   Wildcard(B,Set())
    fcn    Fcn(List(Wildcard(B,Set()), Wildcard(A,Set())),Wildcard(B,Set()))
           → Wildcard(B,Set())
```

```
{"a.insert": [a, index, item]}
    a       Array(Wildcard(A,Set()))
    index   Int
    item    Wildcard(A,Set())
            → Array(Wildcard(A,Set()))
```

```
{"a.flatmap": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set())),Array(Wildcard(B,Set())))
          → Array(Wildcard(B,Set()))
```

```
{"a.filtermap": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set())),Union(List(Wildcard(B,Set()), Null)))
          → Array(Wildcard(B,Set()))
```

```
{"m.cos": [x]}
    x   Double
        → Double
```

`{"cmp": [x, y]}`
`x` Wildcard(A,Set())
`y` Wildcard(A,Set())
→ Int

`{"a.append": [a, item]}`
`a` Array(Wildcard(A,Set()))
`item` Wildcard(A,Set())
→ Array(Wildcard(A,Set()))

`{"m.ln": [x]}`
`x` Double
→ Double

`{"a.argminLT": [a, lessThan]}`
`a` Array(Wildcard(A,Set()))
`lessThan` Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
→ Int

`{"s.upper": [s]}`
`s` String
→ String

`{"m.expm1": [x]}`
`x` Double
→ Double

`{"*": [x, y]}`
`x` Wildcard(A,Set("int", "long", "float", "double"))
`y` Wildcard(A,Set())
→ Wildcard(A,Set())

`{"<=": [x, y]}`
`x` Wildcard(A,Set())
`y` Wildcard(A,Set())
→ Boolean

`{"m.log": [x, base]}`
`x` Double
`base` Int
→ Double

`{"a.shuffle": [a]}`
`a` Array(Wildcard(A,Set()))
→ Array(Wildcard(A,Set()))

`{"a.endswith": [haystack, needle]}`
`haystack` Array(Wildcard(A,Set()))
`needle` Array(Wildcard(A,Set()))
→ Boolean
or
`haystack` Array(Wildcard(A,Set()))
`needle` Wildcard(A,Set())
→ Boolean

`{"%": [k, n]}`
`k` Wildcard(A,Set("int", "long", "float", "double"))
`n` Wildcard(A,Set())
→ Wildcard(A,Set())

`{"a.rindex": [haystack, needle]}`

```
    haystack   Array(Wildcard(A,Set()))
    needle     Array(Wildcard(A,Set()))
               → Int
```
or
```
    haystack   Array(Wildcard(A,Set()))
    needle     Wildcard(A,Set())
               → Int
```

```
{"m.log10": [x]}
    x   Double
        → Double
```

```
{"s.replaceall": [s, original, replacement]}
    s             String
    original      String
    replacement   String
                  → String
```

```
{"a.intersect": [a, b]}
    a   Array(Wildcard(A,Set()))
    b   Array(Wildcard(A,Set()))
        → Array(Wildcard(A,Set()))
```

```
{"a.max": [a]}
    a   Array(Wildcard(A,Set()))
        → Wildcard(A,Set())
```

```
{"impute.errorOnNull": [x]}
    x   Union(List(Wildcard(A,Set()), Null))
        → Wildcard(A,Set())
```

```
{"m.abs": [x]}
    x   Wildcard(A,Set("int", "long", "float", "double"))
        → Wildcard(A,Set())
```

```
{"a.minNLT": [a, n, lessThan]}
    a          Array(Wildcard(A,Set()))
    n          Int
    lessThan   Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
               → Array(Wildcard(A,Set()))
```

```
{"s.repeat": [s, n]}
    s   String
    n   Int
        → String
```

```
{"m.atan2": [y, x]}
    y   Double
    x   Double
        → Double
```

```
{"stat.sample.updateMean": [runningSum, w, x]}
    runningSum   WildRecord(A,Map(sum_w -> Double, sum_wx -> Double))
    w            Double
    x            Double
                 → Wildcard(A,Set())
```

```
{"s.substr": [s, start, end]}
```

```
    s       String
    start   Int
    end     Int
            → String
```

```
{"a.concat": [a, b]}
    a   Array(Wildcard(A,Set()))
    b   Array(Wildcard(A,Set()))
        → Array(Wildcard(A,Set()))
```

```
{"a.reduce": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Wildcard(A,Set()))
          → Wildcard(A,Set())
```

```
{"s.len": [s]}
    s   String
        → Int
```

```
{"a.subset": [little, big]}
    little   Array(Wildcard(A,Set()))
    big      Array(Wildcard(A,Set()))
             → Boolean
```

```
{"a.diff": [a, b]}
    a   Array(Wildcard(A,Set()))
    b   Array(Wildcard(A,Set()))
        → Array(Wildcard(A,Set()))
```

```
{"m.exp": [x]}
    x   Double
        → Double
```

```
{"a.argminNLT": [a, n, lessThan]}
    a          Array(Wildcard(A,Set()))
    n          Int
    lessThan   Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
               → Array(Int)
```

```
{"a.index": [haystack, needle]}
    haystack   Array(Wildcard(A,Set()))
    needle     Array(Wildcard(A,Set()))
               → Int
```

or

```
    haystack   Array(Wildcard(A,Set()))
    needle     Wildcard(A,Set())
               → Int
```

```
{"a.maxN": [a, n]}
    a   Array(Wildcard(A,Set()))
    n   Int
        → Array(Wildcard(A,Set()))
```

```
{"a.distinct": [a]}
    a   Array(Wildcard(A,Set()))
        → Array(Wildcard(A,Set()))
```

```
{"a.sum": [a]}
    a   Array(Wildcard(A,Set("int", "long", "float", "double")))
        → Wildcard(A,Set())
```

```
{"a.reduceright": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Wildcard(A,Set()))
          → Wildcard(A,Set())

{"a.any": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set())),Boolean)
          → Boolean

{"s.substrto": [s, start, end, replacement]}
    s            String
    start        Int
    end          Int
    replacement  String
                 → String

{"a.argmin": [a]}
    a  Array(Wildcard(A,Set()))
       → Int

{"impute.defaultOnNull": [x, default]}
    x        Union(List(Wildcard(A,Set()), Null))
    default  Wildcard(A,Set())
             → Wildcard(A,Set())

{"a.permutations": [a]}
    a  Array(Wildcard(A,Set()))
       → Array(Array(Wildcard(A,Set())))

{"s.join": [array, sep]}
    array  Array(String)
    sep    String
           → String

{"a.slidingWindow": [a, size, step, allowIncomplete]}
    a                Array(Wildcard(A,Set()))
    size             Int
    step             Int
    allowIncomplete  Boolean
                     → Array(Array(Wildcard(A,Set())))

{"<": [x, y]}
    x  Wildcard(A,Set())
    y  Wildcard(A,Set())
       → Boolean

{"a.seteq": [a, b]}
    a  Array(Wildcard(A,Set()))
    b  Array(Wildcard(A,Set()))
       → Boolean

{"s.split": [s, sep]}
    s    String
    sep  String
         → Array(String)

{"a.all": [a, fcn]}
```

```
    a      Array(Wildcard(A,Set()))
    fcn    Fcn(List(Wildcard(A,Set())),Boolean)
           → Boolean
```

```
{"&": [x, y]}
    x  Int
    y  Int
       → Int
```

or

```
    x  Long
    y  Long
       → Long
```

```
{"a.mode": [a]}
    a  Array(Wildcard(A,Set()))
       → Wildcard(A,Set())
```

```
{"s.lstrip": [s, chars]}
    s      String
    chars  String
           → String
```

```
{"a.min": [a]}
    a  Array(Wildcard(A,Set()))
       → Wildcard(A,Set())
```

```
{">=": [x, y]}
    x  Wildcard(A,Set())
    y  Wildcard(A,Set())
       → Boolean
```

```
{"a.combinations": [a, size]}
    a     Array(Wildcard(A,Set()))
    size  Int
          → Array(Array(Wildcard(A,Set())))
```

```
{"a.reverse": [a]}
    a  Array(Wildcard(A,Set()))
       → Array(Wildcard(A,Set()))
```

```
{"m.atan": [x]}
    x  Double
       → Double
```

```
{"min": [x, y]}
    x  Wildcard(A,Set())
    y  Wildcard(A,Set())
       → Wildcard(A,Set())
```

```
{"s.replacefirst": [s, original, replacement]}
    s            String
    original     String
    replacement  String
                 → String
```

```
{"m.copysign": [mag, sign]}
    mag   Wildcard(A,Set("int", "long", "float", "double"))
    sign  Wildcard(A,Set())
          → Wildcard(A,Set())
```

```
{"m.sin": [x]}
```

    x Double
       → Double

`{"|": [x, y]}`
    x Int
    y Int
       → Int
or
    x Long
    y Long
       → Long

`{"m.cosh": [x]}`
    x Double
       → Double

`{"m.pi": []}`
      → Double

`{"or": [x, y]}`
    x Boolean
    y Boolean
       → Boolean

`{"a.len": [a]}`
    a Array(Wildcard(A,Set()))
       → Int

`{"s.contains": [haystack, needle]}`
    haystack String
    needle   String
              → Boolean

`{"a.lnsum": [a]}`
    a Array(Double)
       → Double

`{"s.startswith": [haystack, needle]}`
    haystack String
    needle   String
              → Boolean

`{"a.maxLT": [a, lessThan]}`
    a   Array(Wildcard(A,Set()))
    lessThan Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
              → Wildcard(A,Set())

`{"a.argmaxNLT": [a, n, lessThan]}`
    a   Array(Wildcard(A,Set()))
    n   Int
    lessThan Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
              → Array(Int)

`{"-": [x, y]}`
    x Wildcard(A,Set("int", "long", "float", "double"))
    y Wildcard(A,Set())
       → Wildcard(A,Set())

`{"a.takeWhile": [a, fcn]}`

`a` Array(Wildcard(A,Set()))
`fcn` Fcn(List(Wildcard(A,Set())),Boolean)
→ Array(Wildcard(A,Set()))

`{"m.tanh": [x]}`
`x` Double
→ Double

`{"m.signum": [x]}`
`x` Double
→ Int

`{"m.hypot": [x, y]}`
`x` Double
`y` Double
→ Double

`{"s.concat": [x, y]}`
`x` String
`y` String
→ String

`{"max": [x, y]}`
`x` Wildcard(A,Set())
`y` Wildcard(A,Set())
→ Wildcard(A,Set())

`{"a.replace": [a, index, item]}`
`a` Array(Wildcard(A,Set()))
`index` Int
`item` Wildcard(A,Set())
→ Array(Wildcard(A,Set()))

`{"m.tan": [x]}`
`x` Double
→ Double

`{"a.remove": [a, start, end]}` or `{"a.remove": [a, index]}`
`a` Array(Wildcard(A,Set()))
`start` Int
`end` Int
→ Array(Wildcard(A,Set()))
or
`a` Array(Wildcard(A,Set()))
`index` Int
→ Array(Wildcard(A,Set()))

`{"a.groupby": [a, fcn]}`
`a` Array(Wildcard(A,Set()))
`fcn` Fcn(List(Wildcard(A,Set())),String)
→ Map(Array(Wildcard(A,Set())))

`{"a.contains": [haystack, needle]}`
`haystack` Array(Wildcard(A,Set()))
`needle` Array(Wildcard(A,Set()))
→ Boolean
or
`haystack` Array(Wildcard(A,Set()))
`needle` Wildcard(A,Set())
→ Boolean

```
{"**": [x, y]}
```
    x Wildcard(A,Set("int", "long", "float", "double"))
    y Wildcard(A,Set())
       → Wildcard(A,Set())

```
{"==": [x, y]}
```
    x Wildcard(A,Set())
    y Wildcard(A,Set())
       → Boolean

```
{"s.index": [haystack, needle]}
```
    haystack String
    needle   String
              → Int

```
{"xor": [x, y]}
```
    x Boolean
    y Boolean
       → Boolean

```
{"a.subseq": [a, start, end]}
```
    a   Array(Wildcard(A,Set()))
    start Int
    end   Int
           → Array(Wildcard(A,Set()))

```
{"u-": [x]}
```
    x Wildcard(A,Set("int", "long", "float", "double"))
       → Wildcard(A,Set())

```
{"a.corresponds": [a, b, fcn]}
```
    a   Array(Wildcard(A,Set()))
    b   Array(Wildcard(B,Set()))
    fcn Fcn(List(Wildcard(A,Set()), Wildcard(B,Set())),Boolean)
         → Boolean

```
{"not": [x]}
```
    x Boolean
       → Boolean

```
{"a.minLT": [a, lessThan]}
```
    a   Array(Wildcard(A,Set()))
    lessThan Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
              → Wildcard(A,Set())

```
{"m.round": [x]}
```
    x Float
       → Int

or

    x Double
       → Long

```
{"a.sort": [a]}
```
    a Array(Wildcard(A,Set()))
       → Array(Wildcard(A,Set()))

```
{"a.mean": [a]}
```
    a Array(Double)
       → Double

```
{"a.count": [haystack, needle]}
```

`haystack` Array(Wildcard(A,Set()))
`needle` Array(Wildcard(A,Set()))
→ Int

or

`haystack` Array(Wildcard(A,Set()))
`needle` Wildcard(A,Set())
→ Int

`{"a.median": [a]}`
`a` Array(Wildcard(A,Set()))
→ Wildcard(A,Set())

`{"m.ceil": [x]}`
`x` Double
→ Double

`{"m.e": []}`
→ Double

`{"a.filter": [a, fcn]}`
`a` Array(Wildcard(A,Set()))
`fcn` Fcn(List(Wildcard(A,Set())),Boolean)
→ Array(Wildcard(A,Set()))

`{"m.asin": [x]}`
`x` Double
→ Double

`{"s.rindex": [haystack, needle]}`
`haystack` String
`needle` String
→ Int

`{"a.union": [a, b]}`
`a` Array(Wildcard(A,Set()))
`b` Array(Wildcard(A,Set()))
→ Array(Wildcard(A,Set()))

`{"m.sqrt": [x]}`
`x` Double
→ Double

`{"m.rint": [x]}`
`x` Double
→ Double

`{"//": [x, y]}`
`x` Wildcard(A,Set("int", "long"))
`y` Wildcard(A,Set())
→ Wildcard(A,Set())

`{"a.map": [a, fcn]}`
`a` Array(Wildcard(A,Set()))
`fcn` Fcn(List(Wildcard(A,Set())),Wildcard(B,Set()))
→ Array(Wildcard(B,Set()))

`{"+": [x, y]}`
`x` Wildcard(A,Set("int", "long", "float", "double"))
`y` Wildcard(A,Set())
→ Wildcard(A,Set())

`{"!=": [x, y]}`

```
    x   Wildcard(A,Set())
    y   Wildcard(A,Set())
        → Boolean
```

```
{"stat.sample.mean": [runningSum]}
    runningSum   WildRecord(A,Map(sum_w -> Double, sum_wx -> Double))
                 → Double
```

```
{"s.replacelast": [s, original, replacement]}
    s            String
    original     String
    replacement  String
                 → String
```

```
{"a.argmax": [a]}
    a   Array(Wildcard(A,Set()))
        → Int
```

```
{"m.ln1p": [x]}
    x   Double
        → Double
```

```
{"a.argmaxN": [a, n]}
    a   Array(Wildcard(A,Set()))
    n   Int
        → Array(Int)
```

```
{"m.acos": [x]}
    x   Double
        → Double
```

```
{"s.endswith": [haystack, needle]}
    haystack  String
    needle    String
              → Boolean
```

```
{"m.floor": [x]}
    x   Double
        → Double
```

```
{"a.startswith": [haystack, needle]}
    haystack  Array(Wildcard(A,Set()))
    needle    Array(Wildcard(A,Set()))
              → Boolean
```

or

```
    haystack  Array(Wildcard(A,Set()))
    needle    Wildcard(A,Set())
              → Boolean
```

```
{"a.argmaxLT": [a, lessThan]}
    a         Array(Wildcard(A,Set()))
    lessThan  Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
              → Int
```

```
{"s.rstrip": [s, chars]}
    s      String
    chars  String
           → String
```

```
{"m.sinh": [x]}
```

`x` Double
→ Double

`{"~": [x]}`
`x` Int
→ Int
or
`x` Long
→ Long

`{"a.disjoint": [a, b]}`
`a` Array(Wildcard(A,Set()))
`b` Array(Wildcard(A,Set()))
→ Boolean

`{"a.product": [a]}`
`a` Array(Wildcard(A,Set("int", "long", "float", "double")))
→ Wildcard(A,Set())

`{"a.minN": [a, n]}`
`a` Array(Wildcard(A,Set()))
`n` Int
→ Array(Wildcard(A,Set()))

`{"a.foldright": [a, zero, fcn]}`
`a` Array(Wildcard(A,Set()))
`zero` Wildcard(B,Set())
`fcn` Fcn(List(Wildcard(B,Set()), Wildcard(A,Set())),Wildcard(B,Set()))
→ Wildcard(B,Set())

`{"s.strip": [s, chars]}`
`s` String
`chars` String
→ String

`{"^": [x, y]}`
`x` Int
`y` Int
→ Int
or
`x` Long
`y` Long
→ Long

`{"a.subseqto": [a, start, end, replacement]}`
`a` Array(Wildcard(A,Set()))
`start` Int
`end` Int
`replacement` Array(Wildcard(A,Set()))
→ Array(Wildcard(A,Set()))

`{"a.flatten": [a]}`
`a` Array(Array(Wildcard(A,Set())))
→ Array(Wildcard(A,Set()))

`{"and": [x, y]}`
`x` Boolean
`y` Boolean
→ Boolean

`{"/": [x, y]}`

```
    x   Double
    y   Double
        → Double
{">": [x, y]}
    x   Wildcard(A,Set())
    y   Wildcard(A,Set())
        → Boolean
{"a.maxNLT": [a, n, lessThan]}
    a           Array(Wildcard(A,Set()))
    n           Int
    lessThan    Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
                → Array(Wildcard(A,Set()))
{"%%": [k, n]}
    k   Wildcard(A,Set("int", "long", "float", "double"))
    n   Wildcard(A,Set())
        → Wildcard(A,Set())
{"a.symdiff": [a, b]}
    a   Array(Wildcard(A,Set()))
    b   Array(Wildcard(A,Set()))
        → Array(Wildcard(A,Set()))
{"model.tree.predicateWalk": [datum, treeNode, predicate]}
    datum       WildRecord(D,Map())
    treeNode    WildRecord(T,Map(pass -> Union(List(WildRecord(T,Map()), Wildcard(S,Set()))),
                fail -> Union(List(WildRecord(T,Map()), Wildcard(S,Set())))))
    predicate   Fcn(List(WildRecord(D,Map()), WildRecord(T,Map())),Boolean)
                → Wildcard(S,Set())
{"s.count": [haystack, needle]}
    haystack   String
    needle     String
               → Int
{"a.sortLT": [a, lessThan]}
    a           Array(Wildcard(A,Set()))
    lessThan    Fcn(List(Wildcard(A,Set()), Wildcard(A,Set())),Boolean)
                → Array(Wildcard(A,Set()))
{"a.dropWhile": [a, fcn]}
    a     Array(Wildcard(A,Set()))
    fcn   Fcn(List(Wildcard(A,Set())),Boolean)
          → Array(Wildcard(A,Set()))
```

8.1 Basic Arithmetic, Logical, and Bitwise Operators

8.2 Universal Comparison Operators

8.3 Basic Math Library

8.4 Linear Algebra

including named row/col matrices

8.5 String Manipulation

8.6 Regular Expressions

and stemming

8.7 Array Manipulation

8.8 Map, Record, Enum, and Fixed Manipulation

8.9 Missing Data Handling

8.10 Aggregation

SQL-like functions

- group-by tables
- CUSUM

9 Descriptive Statistics Libraries

9.1 Sample Statistics

accumulated mean, median(?)

10 Data Mining Models

10.1 Decision and Regression Trees

10.2 Cluster Models

10.3 Regression

10.4 Neural Networks

10.5 Support Vector Machines