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.

This document is an early draft that has not been endorsed for recommendation by any organization. It describes a proposed specification that could, in the future, become a standard.

Contents

1 Introduction **9**

1.1 Motivation for PFA . . . 9

1.2 Terminology used in this specification . . . 10

1.3 PFA MIME type and file name extension . . . 10

1.4 Levels of PFA conformance and PFA subsets . . . 11

2 PFA document structure **13**

2.1 Cells and Pools . . . 14

2.2 Engine options . . . 15

3 Scoring engine execution model **16**

3.1 Scoring method: map, emit, and fold . . . 16

3.2 Input and output type specification . . . 16

3.3 Execution phases: begin, action, and end . . . 16

3.4 Persistent state: cells and pools . . . 16

3.5 Concurrent access to shared state . . . 16

3.6 Exceptions . . . 16

3.7 Execution control . . . 16

3.8 Pseudorandom number management . . . 16

4 Type system **17**

4.1 Avro types . . . 17

4.2 Type inference . . . 17

4.3 Type resolution, promotion, and covariance . . . 17

4.4 Function parameter patterns . . . 17

5 Symbols, scope, and data structures **18**

5.1 Immutable data, reassignable symbols . . . 18

5.2 Expression-level scope and mutation restrictions . . . 18

5.3 Data structure limitations . . . 18

6 User-defined functions **19**

6.1 Syntax and scope . . . 19

6.2 Anonymous callbacks and function references . . . 19

7 Expressions **20**

7.1 Function calls . . . 20

7.2 Symbol references . . . 20

7.3 Literal values . . . 20

7.4 Creating arrays, maps, and records . . . 20
7.5 Symbol assignment and reassignment . . . 20
7.6 Extracting from and updating arrays, maps, and records . . . 20
7.7 Extracting from and updating cells and pools . . . 20
7.8 Do blocks . . . 20
7.9 Conditionals: if and cond . . . 20
7.10 While loops: pretest and posttest . . . 20
7.11 For loops: by index, array element, and key-value . . . 20
7.12 Type-safe casting . . . 20
7.13 Inline documentation . . . 20
7.14 User-defined exceptions . . . 20
7.15 Log messages . . . 20

8 Core library **21**
8.1 Basic arithmetic . . . 21
8.1.1 Addition of two values (+) . . . 21
8.1.2 Subtraction (−) . . . 21
8.1.3 Multiplication of two values (*) . . . 21
8.1.4 Floating-point division (/) . . . 22
8.1.5 Integer division (//) . . . 22
8.1.6 Negation (u−) . . . 22
8.1.7 Modulo (%) . . . 23
8.1.8 Remainder (%%) . . . 23
8.1.9 Raising to a power (**) . . . 23
8.2 Comparison operators . . . 24
8.2.1 General comparision (cmp) . . . 24
8.2.2 Equality (==) . . . 24
8.2.3 Inequality (!=) . . . 24
8.2.4 Less than (<) . . . 24
8.2.5 Less than or equal to (<=) . . . 25
8.2.6 Greater than (>) . . . 25
8.2.7 Greater than or equal to (>=) . . . 25
8.2.8 Maximum of two values (max) . . . 25
8.2.9 Minimum of two values (min) . . . 26
8.3 Logical operators . . . 26
8.3.1 Logical and (and) . . . 26
8.3.2 Logical or (or) . . . 26
8.3.3 Logical xor (xor) . . . 26

8.3.4 Logical not (not) . . . 27
8.4 Bitwise arithmetic . . . 27
8.4.1 Bitwise and (&) . . . 27
8.4.2 Bitwise or (|) . . . 27
8.4.3 Bitwise xor (^) . . . 27
8.4.4 Bitwise not (~) . . . 28

9 Math library **29**
9.1 Constants . . . 29
9.1.1 Archimedes' constant π (m.pi) . . . 29
9.1.2 Euler's constant e (m.e) . . . 29
9.2 Common functions . . . 29
9.2.1 Square root (m.sqrt) . . . 29
9.2.2 Hypotnuse (m.hypot) . . . 29
9.2.3 Trigonometric sine (m.sin) . . . 30
9.2.4 Trigonometric cosine (m.cos) . . . 30
9.2.5 Trigonometric tangent (m.tan) . . . 30
9.2.6 Inverse trigonometric sine (m.asin) . . . 30
9.2.7 Inverse trigonometric cosine (m.acos) . . . 31
9.2.8 Inverse trigonometric tangent (m.atan) . . . 31
9.2.9 Robust inverse trigonometric tangent (m.atan2) . . . 31
9.2.10 Hyperbolic sine (m.sinh) . . . 32
9.2.11 Hyperbolic cosine (m.cosh) . . . 32
9.2.12 Hyperbolic tangent (m.tanh) . . . 32
9.2.13 Natural exponential (m.exp) . . . 32
9.2.14 Natural exponential minus one (m.expm1) . . . 33
9.2.15 Natural logarithm (m.ln) . . . 33
9.2.16 Logarithm base 10 (m.log10) . . . 33
9.2.17 Arbitrary logarithm (m.log) . . . 33
9.2.18 Natural logarithm of one plus square (m.ln1p) . . . 34
9.3 Rounding . . . 34
9.3.1 Absolute value (m.abs) . . . 34
9.3.2 Floor (m.floor) . . . 34
9.3.3 Ceiling (m.ceil) . . . 35
9.3.4 Simple rounding (m.round) . . . 35
9.3.5 Unbiased rounding (m.rint) . . . 35
9.3.6 Threshold function (m.signum) . . . 36
9.3.7 Copy sign (m.copysign) . . . 36

9.4 Linear algebra . . . 36

10 String manipulation **37**

10.1 Basic access . . . 37
10.1.1 Length (s.len) . . . 37
10.1.2 Extract substring (s.substr) . . . 37
10.1.3 Modify substring (s.substrto) . . . 37
10.2 Search and replace . . . 38
10.2.1 Contains (s.contains) . . . 38
10.2.2 Count instances (s.count) . . . 38
10.2.3 Find first index (s.index) . . . 38
10.2.4 Find last index (s.rindex) . . . 38
10.2.5 Check start (s.startswith) . . . 38
10.2.6 Check end (s.endswith) . . . 39
10.3 Conversions to or from other types . . . 39
10.3.1 Join an array of strings (s.join) . . . 39
10.3.2 Split into an array of strings (s.split) . . . 39
10.4 Conversions to or from other strings . . . 40
10.4.1 Concatenate two strings (s.concat) . . . 40
10.4.2 Repeat pattern (s.repeat) . . . 40
10.4.3 Lowercase (s.lower) . . . 40
10.4.4 Uppercase (s.upper) . . . 40
10.4.5 Left-strip (s.lstrip) . . . 40
10.4.6 Right-strip (s.rstrip) . . . 41
10.4.7 Strip both ends (s.strip) . . . 41
10.4.8 Replace all matches (s.replaceall) . . . 41
10.4.9 Replace first match (s.replacefirst) . . . 42
10.4.10 Replace last match (s.replacelast) . . . 42
10.4.11 Translate characters (s.translate) . . . 42
10.5 Regular Expressions . . . 42

11 Array Manipulation **43**

11.1 Basic access . . . 43
11.1.1 Length (a.len) . . . 43
11.1.2 Extract subsequence (a.subseq) . . . 43
11.1.3 Modify subsequence (a.subseqto) . . . 43
11.2 Search and replace . . . 44
11.2.1 Contains (a.contains) . . . 44
11.2.2 Count instances (a.count) . . . 44

11.2.3 Count instances by predicate (a.countPredicate) . . . 44
11.2.4 Find first index (a.index) . . . 44
11.2.5 Find last index (a.rindex) . . . 45
11.2.6 Check start (a.startswith) . . . 45
11.2.7 Check end (a.endswith) . . . 45
11.3 Manipulation . . . 46
11.3.1 Concatenate two arrays (a.concat) . . . 46
11.3.2 Append (a.append) . . . 46
11.3.3 Insert or prepend (a.insert) . . . 46
11.3.4 Replace item (a.replace) . . . 47
11.3.5 Remove item (a.remove) . . . 47
11.4 Reordering . . . 48
11.4.1 Sort (a.sort) . . . 48
11.4.2 Sort with a less-than function (a.sortLT) . . . 48
11.4.3 Randomly shuffle array (a.shuffle) . . . 48
11.4.4 Reverse order (a.reverse) . . . 48
11.5 Extreme values . . . 49
11.5.1 Maximum of all values (a.max) . . . 49
11.5.2 Minimum of all values (a.min) . . . 49
11.5.3 Maximum with a less-than function (a.maxLT) . . . 49
11.5.4 Minimum with a less-than function (a.minLT) . . . 49
11.5.5 Maximum N items (a.maxN) . . . 50
11.5.6 Minimum N items (a.minN) . . . 50
11.5.7 Maximum N with a less-than function (a.maxNLT) . . . 50
11.5.8 Minimum N with a less-than function (a.minNLT) . . . 51
11.5.9 Argument maximum (a.argmax) . . . 51
11.5.10 Argument minimum (a.argmin) . . . 51
11.5.11 Argument maximum with a less-than function (a.argmaxLT) . . . 52
11.5.12 Argument minimum with a less-than function (a.argminLT) . . . 52
11.5.13 Maximum N arguments (a.argmaxN) . . . 52
11.5.14 Minimum N arguments (a.argminN) . . . 53
11.5.15 Maximum N arguments with a less-than function (a.argmaxNLT) . . . 53
11.5.16 Minimum N arguments with a less-than function (a.argminNLT) . . . 53
11.6 Numerical combinations . . . 54
11.6.1 Add all array values (a.sum) . . . 54
11.6.2 Multiply all array values (a.product) . . . 54
11.6.3 Sum of logarithms (a.lnsum) . . . 54
11.6.4 Arithmetic mean (a.mean) . . . 55

11.6.5 Geometric mean (a.geomean) . . . 55
11.6.6 Median (a.median) . . . 55
11.6.7 Mode, or most common value (a.mode) . . . 55
11.7 Set or set-like functions . . . 56
11.7.1 Distinct items (a.distinct) . . . 56
11.7.2 Set equality (a.seteq) . . . 56
11.7.3 Union (a.union) . . . 56
11.7.4 Intersection (a.intersect) . . . 56
11.7.5 Set difference (a.diff) . . . 57
11.7.6 Symmetric set difference (a.symdiff) . . . 57
11.7.7 Subset check (a.subset) . . . 57
11.7.8 Disjointness check (a.disjoint) . . . 57
11.8 Functional programming . . . 58
11.8.1 Map array items with function (a.map) . . . 58
11.8.2 Filter array items with function (a.filter) . . . 58
11.8.3 Filter and map (a.filtermap) . . . 58
11.8.4 Map and flatten (a.flatmap) . . . 59
11.8.5 Reduce array items to a single value (a.reduce) . . . 59
11.8.6 Right-to-left reduce (a.reduceright) . . . 59
11.8.7 Fold array items to another type (a.fold) . . . 60
11.8.8 Right-to-left fold (a.foldright) . . . 60
11.8.9 Take items until predicate is false (a.takeWhile) . . . 60
11.8.10 Drop items until predicate is true (a.dropWhile) . . . 61
11.9 Functional tests . . . 61
11.9.1 Existential check, ∃ (a.any) . . . 61
11.9.2 Univeral check, ∀ (a.all) . . . 61
11.9.3 Pairwise check of two arrays (a.corresponds) . . . 61
11.10 Restructuring . . . 62
11.10.1 Sliding window (a.slidingWindow) . . . 62
11.10.2 Unique combinations of a fixed size (a.combinations) . . . 62
11.10.3 Permutations (a.permutations) . . . 63
11.10.4 Flatten array (a.flatten) . . . 63
11.10.5 Group items by category (a.groupby) . . . 63

12 Manipulation of other data structures **64**
12.1 Map . . . 64
12.2 Record . . . 64
12.3 Enum . . . 64

12.4 Fixed . . . 64

13 Missing data handling **65**

13.1 Impute library . . . 65

13.1.1 Skip record (impute.errorOnNull) . . . 65

13.1.2 Replace with default (impute.defaultOnNull) . . . 65

14 Aggregation **66**

15 Descriptive statistics libraries **67**

15.1 Sample statistics . . . 67

15.1.1 Update aggregated mean (stat.sample.updateMean) . . . 67

15.1.2 Compute aggregated mean (stat.sample.mean) . . . 67

16 Data mining models **68**

16.1 Decision and regression Trees . . . 68

16.1.1 Tree walk with simple predicates (model.tree.simpleWalk) . . . 68

16.1.2 Tree walk with user-defined predicates (model.tree.predicateWalk) . . . 68

16.2 Cluster models . . . 69

16.3 Regression . . . 69

16.4 Neural networks . . . 69

16.5 Support vector machines . . . 69

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 that flexibility is important, 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 an 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 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 provide generic algorithms, 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 for 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, all parts of the PFA engine, including control structures, type annotations, and embedded data are all expressed in one seamless 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 on 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…", "Specific implementations may…", 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", "string", "boolean", and "null" 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 recommended MIME type for PFA is "application/pfa+json", though this is not yet in the process of standardization.

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

1.4 Levels of PFA conformance and PFA subsets

PFA is a large specification with many modules, so some projects or vendors may wish to implement some but not all of the specification. However, interoperability is the reason PFA exists; if an implementation does not adhere to the standard, it has limited value. It is therefore useful to explicitly define what it means for a system to partially implement the standard.

JSON subtrees of a PFA document are interpreted in the following five contexts.

- Top-level fields are JSON object member name, value pairs in the outermost JSON object of the PFA document. They have unique member names and describe global aspects of the scoring engine.
- Special forms are JSON objects that specify executable expressions and function definitions. Each is associated with a unique name.
- Library functions are strings that specify routines not defined in the PFA document itself. Each is associated with a unique name that does not conflict with any of the special forms' names.
- Avro type schemae are JSON objects and strings that describe data types. The syntax and meaning of Avro types are specified in the Avro 1.7.6 specification.
- Embedded data are JSON objects, JSON arrays, numbers, integers, strings, booleans, and nulls that describe data structures. The syntax and meaning of these objects are also defined by Avro, as the format used by the `JSONEncoder` and `JSONDecoder`.

A system may be partially PFA compliant if it implements some but not all top-level fields, some but not all special forms, or some but not all library functions. Its coverage may be specified by listing the object member names of the top-level fields that it does implement, the names of the special forms that it does implement, and the names of the library functions that it does implement. Those top-level fields, special forms, and library functions that it does implement must be completely and correctly implemented. The coverage is therefore atomic and one can immediately determine if a particular system can execute a particular PFA document by checking the set of names used by the document against the set of names implemented by the system.

Some special forms and library functions make use of some top-level fields. For example, library functions that generate random numbers use the `randseed` field for configuration. These special forms and library functions cannot be considered implemented unless the corresponding top-level fields are also implemented. The dependencies are explicitly defined in this specification.

Avro type schemae and JSON-encoded data should be completely implemented, to the extent defined by the Avro specification. We suggest that implementations use language-specific Avro libraries as much as possible, rather than implementing Avro-related features in a PFA system.

The PFA standard is defined so that a PFA-compliant system can verify that the JSON types of a PFA document are correctly composed (syntax check), verify that the PFA invariants are maintained and Avro data types are correctly composed (semantic check), and impose additional constraints on the set of top-level fields, special forms, and library functions used (optional checks). A PFA-compliant system should perform the syntax and semantic checks, including all type inference and type checking, but it is not required. A PFA document that does not satisfy these invariants and type constraints is not valid and its behavior is not defined by this specification. The third set of checks, however, is completely optional and different systems may apply different constraints on the kinds of scoring engines they are willing to execute. For instance, an

implementation targeting a limited environment in which recursion is not possible may analyze the document and reject it if any recursive loops are found.

This specification does not define any standardized subsets of PFA. As stated above, partial conformance is defined by ad hoc subsets of atomic units. However, as experience develops, the community may define industry-standard subsets of PFA for specific purposes or special environments. Conforming to a standardized subset would provide better interoperability than defining ad hoc subsets, and we would recommend such a standard when it exists. At present, we can only recommend a well-chosen ad hoc subset or complete conformance.

2 PFA document structure

A PFA document is a serialized JSON object representing an executable scoring engine. Only the following JSON object member names may appear at this JSON nesting level. These are the top-level fields referred to in the conformance section of this specification. Three fields, **`action`**, **`input`**, and **`output`**, are required for every PFA document and are therefore required for every PFA implementation. The rest are optional for PFA documents and not strictly required for PFA implementations. As explained in the conformance section, not implementing some top-level fields can make some special forms and functions unimplementable.

`name`: A string used to identify the scoring engine (has no effect on calculations).

`method`: A string that may be "map", "emit", or "fold" (see Sec. 3.1). If absent, the default value is "map".

`input`: An Avro schema representing the data type of data provided to the scoring engine (see Sec. 3.2).

`output`: An Avro schema representing the data type of data produced by the scoring engine (see Sec. 3.2). The way that output is returned to the host system depends on the **`method`**.

`begin`: An expression or JSON array of expressions that are executed in the begin phase of the scoring engine's run (see Sec. 3.3).

`action`: An expression or JSON array of expressions that are executed for each input datum in the active phase of the scoring engine's run (see Sec. 3.3).

`end`: An expression or JSON array of expressions that are executed in the end phase of the scoring engine's run (see Sec. 3.3).

`fcns`: A JSON object whose member values are function definitions, defining routines that may be called by expressions in **`begin`**, **`action`**, **`end`**, or by expressions in other functions.

`zero`: Embedded JSON data whose type must match the **`output`** type of the engine. This is only used by the "fold" method to initialize the fold aggregation.

`cells`: A JSON object whose member values specify statically allocated, named, typed units of persistent state or embedded data (see Sec. 3.4). The format of this JSON object is restricted: see Sec. 2.1.

`pools`: A JSON object whose member values specify dynamically allocated namespaces of typed persistent state (see Sec. 3.4). The format of this JSON object is restricted: see Sec. 2.1.

`randseed`: An integer which, if present, sets the seed for pseudorandom number generation (see Sec. 3.8).

`doc`: A string used to describe the scoring engine or its provenance (has no effect on calculations).

`metadata`: A JSON object, array, string, number, boolean, or null used to describe the scoring engine or its provenance (has no effect on calculations).

`options`: A JSON object of JSON objects, arrays, strings, numbers, booleans, or nulls used to control execution. The format of this JSON object is restricted: see see Sec. 2.2.

Example 2.1. This is the simplest possible PFA document. It only reads **`null`** values, returns **`null`** values, and performs no calculations.

```
{"input": "null", "output": "null", "action": null}
```

Example 2.2. This is a simple yet non-degenerate PFA document. It increments numerical input by 1.

```
{"input": "double", "output": "double", "action": {"+": ["input", 1]}}
```

Example 2.3. This example implements a simple decision tree. Input data are records with three fields: "one" (integer), "two" (double), and "three" (string). The decision tree is stored in a cell named "tree" with type "TreeNode". The tree has three binary splits (four leaves). The scoring engine walks from the root to a leaf for each input datum, choosing a path based on values found in the input data, and returns the string it finds at the leaf. (See the function definition of model.tree.simpleWalk for a complete explanation.)

```
{"input": {"type": "record", "name": "Datum", "fields":
   [{"name": "one", "type": "int"},
    {"name": "two", "type": "double"},
    {"name": "three", "type": "string"}]},
 "output": "string",
 "cells": {"tree":
             {"type":
               {"type": "record",
                "name": "TreeNode",
                "fields": [
                  {"name": "field", "type": "string"},
                  {"name": "operator", "type": "string"},
                  {"name": "value", "type": ["double", "string"]},
                  {"name": "pass", "type": ["string", "TreeNode"]},
                  {"name": "fail", "type": ["string", "TreeNode"]}]},
              "init":
                {"field": "one",
                 "operator": "<",
                 "value": {"double": 12},
                 "pass":
                   {"TreeNode":
                     {"field": "two",
                      "operator": ">",
                      "value": {"double": 3.5},
                      "pass": {"string": "yes-yes"},
                      "fail": {"string": "yes-no"}}},
                 "fail":
                   {"TreeNode":
                     {"field": "three",
                      "operator": "==",
                      "value": {"string": "TEST"},
                      "pass": {"string": "no-yes"},
                      "fail": {"string": "no-no"}}}}}},
 "action":
   {"model.tree.simpleWalk": ["input", {"cell": "tree"}]}}
```

2.1 Cells and Pools

The `cells` and `pools` top-level fields, if present, are JSON objects whose member values are cell-specifications or pool-specifications, respectively. A cell is a mutable, global data store that holds a single value with a specific type, and a pool is a mutable map from dynamically allocated names to values of a specific type (see Sec. 3.4).

A cell-specification is a JSON object with the following fields.

`type`: *(required)* An Avro schema representing the data type of this cell.

`init`: *(required)* Embedded JSON whose type must match **`type`**. This is the initial value of the cell (or constant value if it is never modified).

`shared`: An optional boolean specifying whether this cell is thread-local to one scoring engine or shared among a battery of similar engines (see Sec. 3.5). The default is false.

A pool-specification is a JSON object with the following fields.

`type`: *(required)* An Avro schema representing the data type of this pool.

`init`: JSON object whose member values are embedded JSON that must match **`type`**. Unlike a cell, a pool may be empty on initialization, in which case **`init`** is either unspecified or **`{}`**.

`shared`: An optional boolean specifying whether this pool is thread-local to one scoring engine or shared among a battery of similar engines (see Sec. 3.5). The default is false.

A complete explanation of cells and pools is given in Sec. 3.4.

2.2 Engine options

The top-level field **`options`**, if present, is a JSON object whose member names must be chosen from the following. Each member name is associated with a particular JSON type, also specified below. See Sec. 3.7 for a complete explanation of the overridable options framework.

`timeout`: *(integer)* If positive, the maximum number of milliseconds before a single execution of **`begin`**, **`action`**, or **`end`** throws a timeout exception. Default is -1 (no timeout).

`timeout.begin`: *(integer)* If positive, the maximum number of milliseconds before a single execution of **`begin`** throws a timeout exception. Overrides **`timeout`** if present and defaults to **`timeout`** if not.

`timeout.action`: *(integer)* If positive, the maximum number of milliseconds before a single execution of **`action`** throws a timeout exception. Overrides **`timeout`** if present and defaults to **`timeout`** if not.

`timeout.end`: *(integer)* If positive, the maximum number of milliseconds before a single execution of **`end`** throws a timeout exception. Overrides **`timeout`** if present and defaults to **`timeout`** if not.

3 Scoring engine execution model

3.1 Scoring method: map, emit, and fold

3.2 Input and output type specification

data funnel (how input and output are handled)

3.3 Execution phases: begin, action, and end

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

3.4 Persistent state: cells and pools

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

3.5 Concurrent access to shared state

multiple engines

3.6 Exceptions

3.7 Execution control

overridable options

timeouts

3.8 Pseudorandom number management

random numbers

4 Type system

4.1 Avro types

Type-safe null

4.2 Type inference

4.3 Type resolution, promotion, and covariance

4.4 Function parameter patterns

5 Symbols, scope, and data structures

garbage collector

5.1 Immutable data, reassignable symbols

5.2 Expression-level scope and mutation restrictions

5.3 Data structure limitations

No circular references

String-only map keys

6 User-defined functions

6.1 Syntax and scope

6.2 Anonymous callbacks and function references

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

8.1 Basic arithmetic

8.1.1 Addition of two values (+)

Signature: `{"+": [x, y]}`

`x`	any `A` of {int, long, float, double}
`y`	`A`
(returns)	`A`

Description: Add `x` and `y`.

Details:

Float and double overflows do not produce runtime errors but result in positive or negative infinity, which would be carried through any subsequent calculations (see IEEE 754). Use `impute.ensureFinite` to produce errors from infinite or NaN values.

Runtime Errors:

Integer results above or below -2147483648 and 2147483647 (inclusive) produce an "int overflow" runtime error.

Long-integer results above or below -9223372036854775808 and 9223372036854775807 (inclusive) produce a "long overflow" runtime error.

8.1.2 Subtraction (−)

Signature: `{"-": [x, y]}`

`x`	any `A` of {int, long, float, double}
`y`	`A`
(returns)	`A`

Description: Subtract `y` from `x`.

Details:

Float and double overflows do not produce runtime errors but result in positive or negative infinity, which would be carried through any subsequent calculations (see IEEE 754). Use `impute.ensureFinite` to produce errors from infinite or NaN values.

Runtime Errors:

Integer results above or below -2147483648 and 2147483647 (inclusive) produce an "int overflow" runtime error.

Long-integer results above or below -9223372036854775808 and 9223372036854775807 (inclusive) produce a "long overflow" runtime error.

8.1.3 Multiplication of two values (*)

Signature: `{"*": [x, y]}`

`x`	any `A` of {int, long, float, double}
`y`	`A`
(returns)	`A`

Description: Multiply `x` and `y`.

Details:

Float and double overflows do not produce runtime errors but result in positive or negative infinity, which would be carried through any subsequent calculations (see IEEE 754). Use `impute.ensureFinite` to produce errors from infinite or NaN values.

Runtime Errors:

Integer results above or below -2147483648 and 2147483647 (inclusive) produce an "int overflow" runtime error.

Long-integer results above or below -9223372036854775808 and 9223372036854775807 (inclusive) produce a "long overflow" runtime error.

8.1.4 Floating-point division (/)

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

`x`	double
`y`	double
(returns)	double

Description: Divide `y` from `x`, returning a floating-point number (even if `x` and `y` are integers).

8.1.5 Integer division (//)

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

`x`	any `A` of {int, long}
`y`	`A`
(returns)	`A`

Description: Divide `y` from `x`, returning the largest whole number `N` for which $N \leq x/y$ (integral floor division).

8.1.6 Negation (u−)

Signature: `{"u-": [x]}`

`x`	any `A` of {int, long, float, double}
(returns)	`A`

Description: Return the additive inverse of `x`.

Runtime Errors:

For exactly one integer value, -2147483648, this function produces an "int overflow" runtime error.

For exactly one long value, -9223372036854775808, this function produces a "long overflow" runtime error.

8.1.7 Modulo (%)

Signature: `{"%": [k, n]}`

`k`	any `A` of {int, long, float, double}
`n`	`A`
(returns)	`A`

Description: Return `k` modulo `n`; the result has the same sign as the modulus `n`.

Details:

This is the behavior of the `%` operator in Python, `mod`/`modulo` in Ada, Haskell, and Scheme.

8.1.8 Remainder (%%)

Signature: `{"%%": [k, n]}`

`k`	any `A` of {int, long, float, double}
`n`	`A`
(returns)	`A`

Description: Return the remainder of `k` divided by `n`; the result has the same sign as the dividend `k`.

Details:

This is the behavior of the `%` operator in Fortran, C/C++, and Java, `rem`/`remainder` in Ada, Haskell, and Scheme.

8.1.9 Raising to a power (**)

Signature: `{"**": [x, y]}`

`x`	any `A` of {int, long, float, double}
`y`	`A`
(returns)	`A`

Description: Raise `x` to the power `n`.

Details:

Float and double overflows do not produce runtime errors but result in positive or negative infinity, which would be carried through any subsequent calculations (see IEEE 754). Use `impute.ensureFinite` to produce errors from infinite or NaN values.

Runtime Errors:

Integer results above or below -2147483648 and 2147483647 (inclusive) produce an "int overflow" runtime error.

Long-integer results above or below -9223372036854775808 and 9223372036854775807 (inclusive) produce a "long overflow" runtime error.

8.2 Comparison operators

Avro defines a sort order for every pair of values with a compatible type, so any two objects of compatible type can be compared in PFA.

8.2.1 General comparision (cmp)

Signature: `{"cmp": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	int

Description: Return `1` if `x` is greater than `y`, `-1` if `x` is less than `y`, and `0` if `x` and `y` are equal.

8.2.2 Equality (==)

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

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is equal to `y`, `false` otherwise.

8.2.3 Inequality (!=)

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

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is not equal to `y`, `false` otherwise.

8.2.4 Less than (<)

Signature: `{"<": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is less than `y`, `false` otherwise.

8.2.5 Less than or equal to (<=)

Signature: `{"<=": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is less than or equal to `y`, `false` otherwise.

8.2.6 Greater than (>)

Signature: `{">": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is greater than `y`, `false` otherwise.

8.2.7 Greater than or equal to (>=)

Signature: `{">=": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	boolean

Description: Return `true` if `x` is greater than or equal to `y`, `false` otherwise.

8.2.8 Maximum of two values (max)

Signature: `{"max": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	`A`

Description: Return `x` if $x \geq y$, `y` otherwise.

Details:

For the maximum of more than two values, see `a.max`

8.2.9 Minimum of two values (min)

Signature: `{"min": [x, y]}`

`x`	any `A`
`y`	`A`
(returns)	`A`

Description: Return `x` if `x` < `y`, `y` otherwise.

Details:

For the minimum of more than two values, see `a.min`

8.3 Logical operators

8.3.1 Logical and (and)

Signature: `{"and": [x, y]}`

`x`	boolean
`y`	boolean
(returns)	boolean

Description: Return `true` if `x` and `y` are both `true`, `false` otherwise.

Details:

If `x` is `false`, `y` won't be evaluated. (Only relevant for arguments with side effects.)

8.3.2 Logical or (or)

Signature: `{"or": [x, y]}`

`x`	boolean
`y`	boolean
(returns)	boolean

Description: Return `true` if either `x` or `y` (or both) are `true`, `false` otherwise.

Details:

If `x` is `true`, `y` won't be evaluated. (Only relevant for arguments with side effects.)

8.3.3 Logical xor (xor)

Signature: `{"xor": [x, y]}`

`x`	boolean
`y`	boolean
(returns)	boolean

Description: Return `true` if `x` is `true` and `y` is `false` or if `x` is `false` and `y` is `true`, but return `false` for any other case.

8.3.4 Logical not (not)

Signature: `{"not": [x]}`

`x`	boolean
(returns)	boolean

Description: Return `true` if `x` is `false` and `false` if `x` is `true`.

8.4 Bitwise arithmetic

8.4.1 Bitwise and (&)

Signature: `{"&": [x, y]}`

`x`	int
`y`	int
(returns)	int

or

`x`	long
`y`	long
(returns)	long

Description: Calculate the bitwise-and of `x` and `y`.

8.4.2 Bitwise or (|)

Signature: `{"|": [x, y]}`

`x`	int
`y`	int
(returns)	int

or

`x`	long
`y`	long
(returns)	long

Description: Calculate the bitwise-or of `x` and `y`.

8.4.3 Bitwise xor (^)

Signature: `{"^": [x, y]}`

`x`	int
`y`	int
(returns)	int

or

`x`	long
`y`	long
(returns)	long

Description: Calculate the bitwise-exclusive-or of `x` and `y`.

8.4.4 Bitwise not (~)

Signature: `{"~": [x]}`

`x`	int
(returns)	int

or

`x`	long
(returns)	long

Description: Calculate the bitwise-not of `x`.

9 Math library

9.1 Constants

Constants such as π and e are represented as stateless functions with no arguments. Specific implementations may choose to replace the function call with its inline value.

9.1.1 Archimedes' constant π (m.pi)

Signature: `{"m.pi": []}`

(returns) double

Description: The double-precision number that is closer than any other to π, the ratio of a circumference of a circle to its diameter.

9.1.2 Euler's constant e (m.e)

Signature: `{"m.e": []}`

(returns) double

Description: The double-precision number that is closer than any other to e, the base of natural logarithms.

9.2 Common functions

9.2.1 Square root (m.sqrt)

Signature: `{"m.sqrt": [x]}`

`x` double
(returns) double

Description: Return the positive square root of `x`.

Details:

The domain of this function is from 0 (inclusive) to infinity. Beyond this domain, the result is Use

9.2.2 Hypotnuse (m.hypot)

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

`x` double
`y` double
(returns) double

Description: Return $\sqrt{x^2 + y^2}$.

Details:

Avoids round-off or overflow errors in the intermediate steps.

The domain of this function is the whole real line; no input is invalid.

9.2.3 Trigonometric sine (m.sin)

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

`x`	double
(returns)	double

Description: Return the trigonometric sine of `x`, which is assumed to be in radians.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.4 Trigonometric cosine (m.cos)

Signature: `{"m.cos": [x]}`

`x`	double
(returns)	double

Description: Return the trigonometric cosine of `x`, which is assumed to be in radians.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.5 Trigonometric tangent (m.tan)

Signature: `{"m.tan": [x]}`

`x`	double
(returns)	double

Description: Return the trigonometric tangent of `x`, which is assumed to be in radians.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.6 Inverse trigonometric sine (m.asin)

Signature: `{"m.asin": [x]}`

`x`	double
(returns)	double

Description: Return the arc-sine (inverse of the sine function) of `x` as an angle in radians between $-\pi/2$ and $\pi/2$.

Details:

The domain of this function is from -1 to 1 (inclusive). Beyond this domain, the result is Use

9.2.7 Inverse trigonometric cosine (m.acos)

Signature: `{"m.acos": [x]}`

`x`	double
(returns)	double

Description: Return the arc-cosine (inverse of the cosine function) of `x` as an angle in radians between 0 and π.

Details:

The domain of this function is from -1 to 1 (inclusive). Beyond this domain, the result is Use

9.2.8 Inverse trigonometric tangent (m.atan)

Signature: `{"m.atan": [x]}`

`x`	double
(returns)	double

Description: Return the arc-tangent (inverse of the tangent function) of `x` as an angle in radians between $-\pi/2$ and $\pi/2$.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.9 Robust inverse trigonometric tangent (m.atan2)

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

`y`	double
`x`	double
(returns)	double

Description: Return the arc-tangent (inverse of the tangent function) of `y`/`x` without loss of precision for small `x`.

Details:

The domain of this function is the whole real plane; no pair of inputs is invalid.

Note that `y` is the first parameter and `x` is the second parameter.

9.2.10 Hyperbolic sine (m.sinh)

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

`x`	double
(returns)	double

Description: Return the hyperbolic sine of `x`, which is equal to $\frac{e^x - e^{-x}}{2}$.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.11 Hyperbolic cosine (m.cosh)

Signature: `{"m.cosh": [x]}`

`x`	double
(returns)	double

Description: Return the hyperbolic cosine of `x`, which is equal to $\frac{e^x + e^{-x}}{2}$

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.12 Hyperbolic tangent (m.tanh)

Signature: `{"m.tanh": [x]}`

`x`	double
(returns)	double

Description: Return the hyperbolic tangent of `x`, which is equal to $\frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.13 Natural exponential (m.exp)

Signature: `{"m.exp": [x]}`

`x`	double
(returns)	double

Description: Return `m.e` raised to the power of `x`.

Details:

The domain of this function is the whole real line; no input is invalid.

9.2.14 Natural exponential minus one (m.expm1)

Signature: `{"m.expm1": [x]}`

`x`	double
(returns)	double

Description: Return $e^x - 1$.

Details:

Avoids round-off or overflow errors in the intermediate steps.

The domain of this function is the whole real line; no input is invalid.

9.2.15 Natural logarithm (m.ln)

Signature: `{"m.ln": [x]}`

`x`	double
(returns)	double

Description: Return the natural logarithm of `x`.

Details:

The domain of this function is from 0 to infinity (exclusive). Given zero, the result is negative infinity, and below zero, the result is Use

9.2.16 Logarithm base 10 (m.log10)

Signature: `{"m.log10": [x]}`

`x`	double
(returns)	double

Description: Return the logarithm base 10 of `x`.

Details:

The domain of this function is from 0 to infinity (exclusive). Given zero, the result is negative infinity, and below zero, the result is Use

9.2.17 Arbitrary logarithm (m.log)

Signature: `{"m.log": [x, base]}`

`x`	double
`base`	int
(returns)	double

Description: Return the logarithm of `x` with a given `base`.

Details:

The domain of this function is from 0 to infinity (exclusive). Given zero, the result is negative infinity, and below zero, the result is Use

Runtime Errors:

If `base` is less than or equal to zero, this function produces a "base must be positive" runtime error.

9.2.18 Natural logarithm of one plus square (m.ln1p)

Signature: `{"m.ln1p": [x]}`

`x`	double
(returns)	double

Description: Return $ln(x^2 + 1)$.

Details:

Avoids round-off or overflow errors in the intermediate steps.

The domain of this function is from -1 to infinity (exclusive). Given -1, the result is negative infinity, and below -1, the result is Use

9.3 Rounding

9.3.1 Absolute value (m.abs)

Signature: `{"m.abs": [x]}`

`x`	any `A` of {int, long, float, double}
(returns)	`A`

Description: Return the absolute value of `x`.

Details:

The domain of this function is the whole real line; no input is invalid.

Runtime Errors:

For exactly one integer value, -2147483648, this function produces an "int overflow" runtime error.

For exactly one long value, -9223372036854775808, this function produces a "long overflow" runtime error.

9.3.2 Floor (m.floor)

Signature: `{"m.floor": [x]}`

`x`	double
(returns)	double

Description: Return the largest (closest to positive infinity) whole number that is less than or equal to

the input.

Details:

The domain of this function is the whole real line; no input is invalid.

9.3.3 Ceiling (m.ceil)

Signature: `{"m.ceil": [x]}`

`x`	double
(returns)	double

Description: Return the smallest (closest to negative infinity, not closest to zero) whole number that is greater than or equal to the input.

Details:

The domain of this function is the whole real line; no input is invalid.

9.3.4 Simple rounding (m.round)

Signature: `{"m.round": [x]}`

`x`	float
(returns)	int

or

`x`	double
(returns)	long

Description: Return the closest whole number to `x`, rounding up if the fractional part is exactly one-half.

Details:

Equal to `m.floor` of (`x` + 0.5).

Runtime Errors:

Integer results outside of -2147483648 and 2147483647 (inclusive) produce an "int overflow" runtime error.

Long-integer results outside of -9223372036854775808 and 9223372036854775807 (inclusive) produce a "long overflow" runtime error.

9.3.5 Unbiased rounding (m.rint)

Signature: `{"m.rint": [x]}`

`x`	double
(returns)	double

Description: Return the closest whole number to `x`, rounding toward the nearest even number if the fractional part is exactly one-half.

9.3.6 Threshold function (m.signum)

Signature: `{"m.signum": [x]}`

`x`	double
(returns)	int

Description: Return 0 if `x` is zero, 1 if `x` is positive, and -1 if `x` is negative.

Details:

The domain of this function is the whole real line; no input is invalid.

9.3.7 Copy sign (m.copysign)

Signature: `{"m.copysign": [mag, sign]}`

`mag`	any `A` of {int, long, float, double}
`sign`	`A`
(returns)	`A`

Description: Return a number with the magnitude of `mag` and the sign of `sign`.

Details:

The domain of this function is the whole real or integer plane; no pair of inputs is invalid.

9.4 Linear algebra

including named row/col matrices

10 String manipulation

Strings are immutable, so none of the following functions modifies a string in-place. Some return a modified version of the original string.

10.1 Basic access

10.1.1 Length (s.len)

Signature: `{"s.len": [s]}`

`s`	string
(returns)	int

Description: Return the length of string `s`.

10.1.2 Extract substring (s.substr)

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

`s`	string
`start`	int
`end`	int
(returns)	string

Description: Return the substring of `s` from `start` (inclusive) until `end` (exclusive).

Details:

Negative indexes count from the right (-1 is just before the last item), indexes beyond the legal range are truncated, and `end` $\leq$ `start` specifies a zero-length subsequence just before the `start` character. All of these rules follow Python's slice behavior.

10.1.3 Modify substring (s.substrto)

Signature: `{"s.substrto": [s, start, end, replacement]}`

`s`	string
`start`	int
`end`	int
`replacement`	string
(returns)	string

Description: Replace `s` from `start` (inclusive) until `end` (exclusive) with `replacement`.

Details:

Negative indexes count from the right (-1 is just before the last item), indexes beyond the legal range are truncated, and `end` $\leq$ `start` specifies a zero-length subsequence just before the `start` character. All of these rules follow Python's slice behavior.

10.2 Search and replace

10.2.1 Contains (s.contains)

Signature: `{"s.contains": [haystack, needle]}`

`haystack`	string
`needle`	string
(returns)	boolean

Description: Return `true` if `haystack` contains `needle`, `false` otherwise.

10.2.2 Count instances (s.count)

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

`haystack`	string
`needle`	string
(returns)	int

Description: Count the number of times `needle` appears in `haystack`.

10.2.3 Find first index (s.index)

Signature: `{"s.index": [haystack, needle]}`

`haystack`	string
`needle`	string
(returns)	int

Description: Return the lowest index where `haystack` contains `needle` or -1 if `haystack` does not contain `needle`.

10.2.4 Find last index (s.rindex)

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

`haystack`	string
`needle`	string
(returns)	int

Description: Return the highest index where `haystack` contains `needle` or -1 if `haystack` does not contain `needle`.

10.2.5 Check start (s.startswith)

Signature: `{"s.startswith": [haystack, needle]}`

`haystack`	string
`needle`	string
(returns)	boolean

Description: Return `true` if the first (leftmost) subseqence of `haystack` is equal to `needle`, false otherwise.

10.2.6 Check end (s.endswith)

Signature: `{"s.endswith": [haystack, needle]}`

`haystack`	string
`needle`	string
(returns)	boolean

Description: Return `true` if the last (rightmost) subseqence of `haystack` is equal to `needle`, false otherwise.

10.3 Conversions to or from other types

10.3.1 Join an array of strings (s.join)

Signature: `{"s.join": [array, sep]}`

`array`	array of string
`sep`	string
(returns)	string

Description: Combine strings from `array` into a single string, delimited by `sep`.

10.3.2 Split into an array of strings (s.split)

Signature: `{"s.split": [s, sep]}`

`s`	string
`sep`	string
(returns)	array of string

Description: Divide a string into an array of substrings, splitting at and removing delimiters `sep`.

Details:

If `s` does not contain `sep`, this function returns an array whose only element is `s`. If `sep` appears at the beginning or end of `s`, the array begins with or ends with an empty string. These conventions match Python's behavior.

10.4 Conversions to or from other strings

10.4.1 Concatenate two strings (s.concat)

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

`x`	string
`y`	string
(returns)	string

Description: Append `y` to `x` to form a single string.

Details:

To concatenate an array of strings, use s.join with an empty string as `sep`.

10.4.2 Repeat pattern (s.repeat)

Signature: `{"s.repeat": [s, n]}`

`s`	string
`n`	int
(returns)	string

Description: Create a string by concatenating `s` with itself `n` times.

10.4.3 Lowercase (s.lower)

Signature: `{"s.lower": [s]}`

`s`	string
(returns)	string

Description: Convert `s` to lower-case.

10.4.4 Uppercase (s.upper)

Signature: `{"s.upper": [s]}`

`s`	string
(returns)	string

Description: Convert `s` to upper-case.

10.4.5 Left-strip (s.lstrip)

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

`s`	string
`chars`	string
(returns)	string

Description: Remove any characters found in `chars` from the beginning (left) of `s`.

Details:

The order of characters in `chars` is irrelevant.

10.4.6 Right-strip (s.rstrip)

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

`s`	string
`chars`	string
(returns)	string

Description: Remove any characters found in `chars` from the end (right) of `s`.

Details:

The order of characters in `chars` is irrelevant.

10.4.7 Strip both ends (s.strip)

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

`s`	string
`chars`	string
(returns)	string

Description: Remove any characters found in `chars` from the beginning or end of `s`.

Details:

The order of characters in `chars` is irrelevant.

10.4.8 Replace all matches (s.replaceall)

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

`s`	string
`original`	string
`replacement`	string
(returns)	string

Description: Replace every instance of the substring `original` from `s` with `replacement`.

10.4.9 Replace first match (s.replacefirst)

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

`s`	string
`original`	string
`replacement`	string
(returns)	string

Description: Replace the first (leftmost) instance of the substring `original` from `s` with `replacement`.

10.4.10 Replace last match (s.replacelast)

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

`s`	string
`original`	string
`replacement`	string
(returns)	string

Description: Replace the last (rightmost) instance of the substring `original` from `s` with `replacement`.

10.4.11 Translate characters (s.translate)

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

`s`	string
`oldchars`	string
`newchars`	string
(returns)	string

Description: For each character in `s` that is also in `oldchars` with some index `i`, replace it with the character at index `i` in `newchars`. Any character in `s` that is not in `oldchars` is unchanged. Any index `i` that is greater than the length of `newchars` is replaced with nothing.

Details:

This is the behavior of the the Posix command `tr`, where `s` takes the place of standard input and `oldchars` and `newchars` are the `tr` commandline options.

10.5 Regular Expressions

and stemming

11 Array Manipulation

11.1 Basic access

11.1.1 Length (a.len)

Signature: `{"a.len": [a]}`

`a`	array of any `A`
(returns)	int

Description: Return the length of array `a`.

11.1.2 Extract subsequence (a.subseq)

Signature: `{"a.subseq": [a, start, end]}`

`a`	array of any `A`
`start`	int
`end`	int
(returns)	array of `A`

Description: Return the subsequence of `a` from `start` (inclusive) until `end` (exclusive).

Details:

Negative indexes count from the right (-1 is just before the last item), indexes beyond the legal range are truncated, and `end` $\leq$ `start` specifies a zero-length subsequence just before the `start` character. All of these rules follow Python's slice behavior.

11.1.3 Modify subsequence (a.subseqto)

Signature: `{"a.subseqto": [a, start, end, replacement]}`

`a`	array of any `A`
`start`	int
`end`	int
`replacement`	array of `A`
(returns)	array of `A`

Description: Return a new array by replacing `a` from `start` (inclusive) until `end` (exclusive) with `replacement`.

Details:

Negative indexes count from the right (-1 is just before the last item), indexes beyond the legal range are truncated, and `end` $\leq$ `start` specifies a zero-length subsequence just before the `start` character. All of these rules follow Python's slice behavior.

Note: `a` is not changed in-place; this is a side-effect-free function.

11.2 Search and replace

11.2.1 Contains (a.contains)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	boolean

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	boolean

Description: Return `true` if `haystack` contains `needle`, `false` otherwise.

11.2.2 Count instances (a.count)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	int

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	int

Description: Count the number of times `needle` appears in `haystack`.

11.2.3 Count instances by predicate (a.countPredicate)

libfcna.countPredicate

11.2.4 Find first index (a.index)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	int

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	int

Description: Return the lowest index where `haystack` contains `needle` or -1 if `haystack` does not contain `needle`.

11.2.5 Find last index (a.rindex)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	int

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	int

Description: Return the highest index where `haystack` contains `needle` or -1 if `haystack` does not contain `needle`.

11.2.6 Check start (a.startswith)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	boolean

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	boolean

Description: Return `true` if the first (leftmost) subseqence of `haystack` is equal to `needle`, false otherwise.

11.2.7 Check end (a.endswith)

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

`haystack`	array of any `A`
`needle`	array of `A`
(returns)	boolean

or

`haystack`	array of any `A`
`needle`	`A`
(returns)	boolean

Description: Return `true` if the last (rightmost) subseqence of `haystack` is equal to `needle`, false otherwise.

11.3 Manipulation

11.3.1 Concatenate two arrays (a.concat)

Signature: `{"a.concat": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	array of `A`

Description: Concatenate `a` and `b` to make a new array of the same type.

Details:

The length of the returned array is the sum of the lengths of `a` and `b`.

11.3.2 Append (a.append)

Signature: `{"a.append": [a, item]}`

`a`	array of any `A`
`item`	`A`
(returns)	array of `A`

Description: Return a new array by adding `item` at the end of `a`.

Details:

Note: `a` is not changed in-place; this is a side-effect-free function.

The length of the returned array is one more than `a`.

11.3.3 Insert or prepend (a.insert)

Signature: `{"a.insert": [a, index, item]}`

`a`	array of any `A`
`index`	int
`item`	`A`
(returns)	array of `A`

Description: Return a new array by inserting `item` at `index` of `a`.

Details:

Negative indexes count from the right (-1 is just before the last item), following Python's index behavior.

Note: `a` is not changed in-place; this is a side-effect-free function.

The length of the returned array is one more than `a`.

Runtime Errors:

If `index` is beyond the range of `a`, an "array out of range" runtime error is raised.

11.3.4 Replace item (a.replace)

Signature: `{"a.replace": [a, index, item]}`

`a`	array of any `A`
`index`	int
`item`	`A`
(returns)	array of `A`

Description: Return a new array by replacing `index` of `a` with `item`.

Details:

Negative indexes count from the right (-1 is just before the last item), following Python's index behavior.

Note: `a` is not changed in-place; this is a side-effect-free function.

The length of the returned array is equal to that of `a`.

Runtime Errors:

If `index` is beyond the range of `a`, an "array out of range" runtime error is raised.

11.3.5 Remove item (a.remove)

Signature: `{"a.remove": [a, start, end]}` or `{"a.remove": [a, index]}`

`a`	array of any `A`
`start`	int
`end`	int
(returns)	array of `A`

or

`a`	array of any `A`
`index`	int
(returns)	array of `A`

Description: Return a new array by removing elements from `a` from `start` (inclusive) until `end` (exclusive) or just a single `index`.

Details:

Negative indexes count from the right (-1 is just before the last item), indexes beyond the legal range are truncated, and `end` $\leq$ `start` specifies a zero-length subsequence just before the `start` character. All of these rules follow Python's slice behavior.

Note: `a` is not changed in-place; this is a side-effect-free function.

The length of the returned array is one less than `a`.

Runtime Errors:

If `index` is beyond the range of `a`, an "array out of range" runtime error is raised.

11.4 Reordering

11.4.1 Sort (a.sort)

Signature: `{"a.sort": [a]}`

`a`	array of any `A`
(returns)	array of `A`

Description: Return an array with the same elements as `a` but in ascending order (as defined by Avro's sort order).

Details:

Note: `a` is not changed in-place; this is a side-effect-free function.

11.4.2 Sort with a less-than function (a.sortLT)

Signature: `{"a.sortLT": [a, lessThan]}`

`a`	array of any `A`
`lessThan`	function (`A`, `A`) → boolean
(returns)	array of `A`

Description: Return an array with the same elements as `a` but in ascending order as defined by the `lessThan` function.

Details:

Note: `a` is not changed in-place; this is a side-effect-free function.

11.4.3 Randomly shuffle array (a.shuffle)

Signature: `{"a.shuffle": [a]}`

`a`	array of any `A`
(returns)	array of `A`

Description: Return an array with the same elements as `a` but in a random order.

Details:

Note: `a` is not changed in-place; this is a side-effect-free function (except for updating the random number generator).

11.4.4 Reverse order (a.reverse)

Signature: `{"a.reverse": [a]}`

`a`	array of any `A`
(returns)	array of `A`

Description: Return the elements of `a` in reversed order.

11.5 Extreme values

11.5.1 Maximum of all values (a.max)

Signature: `{"a.max": [a]}`

`a`	array of any `A`
(returns)	`A`

Description: Return the maximum value in `a` (as defined by Avro's sort order).

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.2 Minimum of all values (a.min)

Signature: `{"a.min": [a]}`

`a`	array of any `A`
(returns)	`A`

Description: Return the minimum value in `a` (as defined by Avro's sort order).

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.3 Maximum with a less-than function (a.maxLT)

Signature: `{"a.maxLT": [a, lessThan]}`

`a`	array of any `A`
`lessThan`	function (`A`, `A`) → boolean
(returns)	`A`

Description: Return the maximum value in `a` as defined by the `lessThan` function.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.4 Minimum with a less-than function (a.minLT)

Signature: `{"a.minLT": [a, lessThan]}`

`a`	array of any `A`
`lessThan`	function (`A`, `A`) → boolean
(returns)	`A`

Description: Return the minimum value in `a` as defined by the `lessThan` function.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.5 Maximum N items (a.maxN)

Signature: `{"a.maxN": [a, n]}`

`a`	array of any `A`
`n`	int
(returns)	array of `A`

Description: Return the `n` highest values in `a` (as defined by Avro's sort order).

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.6 Minimum N items (a.minN)

Signature: `{"a.minN": [a, n]}`

`a`	array of any `A`
`n`	int
(returns)	array of `A`

Description: Return the `n` lowest values in `a` (as defined by Avro's sort order).

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.7 Maximum N with a less-than function (a.maxNLT)

Signature: `{"a.maxNLT": [a, n, lessThan]}`

`a`	array of any `A`
`n`	int
`lessThan`	function (`A`, `A`) → boolean
(returns)	array of `A`

Description: Return the `n` highest values in `a` as defined by the `lessThan` function.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.8 Minimum N with a less-than function (a.minNLT)

Signature: `{"a.minNLT": [a, n, lessThan]}`

`a`	array of any `A`
`n`	int
`lessThan`	function (`A`, `A`) → boolean
(returns)	array of `A`

Description: Return the `n` lowest values in `a` as defined by the `lessThan` function.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.9 Argument maximum (a.argmax)

Signature: `{"a.argmax": [a]}`

`a`	array of any `A`
(returns)	int

Description: Return the index of the maximum value in `a` (as defined by Avro's sort order).

Details:

If the maximum is not unique, this function returns the index of the first maximal value.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.10 Argument minimum (a.argmin)

Signature: `{"a.argmin": [a]}`

`a`	array of any `A`
(returns)	int

Description: Return the index of the minimum value in `a` (as defined by Avro's sort order).

Details:

If the minimum is not unique, this function returns the index of the first minimal value.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.11 Argument maximum with a less-than function (a.argmaxLT)

Signature: `{"a.argmaxLT": [a, lessThan]}`

`a`	array of any `A`
`lessThan`	function (`A`, `A`) → boolean
(returns)	int

Description: Return the index of the maximum value in `a` as defined by the `lessThan` function.

Details:

If the maximum is not unique, this function returns the index of the first maximal value.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.12 Argument minimum with a less-than function (a.argminLT)

Signature: `{"a.argminLT": [a, lessThan]}`

`a`	array of any `A`
`lessThan`	function (`A`, `A`) → boolean
(returns)	int

Description: Return the index of the minimum value in `a` as defined by the `lessThan` function.

Details:

If the minimum is not unique, this function returns the index of the first minimal value.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.5.13 Maximum N arguments (a.argmaxN)

Signature: `{"a.argmaxN": [a, n]}`

`a`	array of any `A`
`n`	int
(returns)	array of int

Description: Return the indexes of the `n` highest values in `a` (as defined by Avro's sort order).

Details:

If any values are not unique, their indexes will be returned in ascending order.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.14 Minimum N arguments (a.argminN)

Signature: `{"a.argminN": [a, n]}`

`a`	array of any `A`
`n`	int
(returns)	array of int

Description: Return the indexes of the `n` lowest values in `a` (as defined by Avro's sort order).

Details:

If any values are not unique, their indexes will be returned in ascending order.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.15 Maximum N arguments with a less-than function (a.argmaxNLT)

Signature: `{"a.argmaxNLT": [a, n, lessThan]}`

`a`	array of any `A`
`n`	int
`lessThan`	function (`A`, `A`) → boolean
(returns)	array of int

Description: Return the indexes of the `n` highest values in `a` as defined by the `lessThan` function.

Details:

If any values are not unique, their indexes will be returned in ascending order.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.5.16 Minimum N arguments with a less-than function (a.argminNLT)

Signature: `{"a.argminNLT": [a, n, lessThan]}`

`a`	array of any `A`
`n`	int
`lessThan`	function (`A`, `A`) → boolean
(returns)	array of int

Description: Return the indexes of the `n` lowest values in `a` as defined by the `lessThan` function.

Details:

If any values are not unique, their indexes will be returned in ascending order.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

If `n` is negative, an "n < 0" runtime error is raised.

11.6 Numerical combinations

11.6.1 Add all array values (a.sum)

Signature: `{"a.sum": [a]}`

`a`	array of any `A` of {int, long, float, double}
(returns)	`A`

Description: Return the sum of numbers in `a`.

Details:

Returns zero if the array is empty.

11.6.2 Multiply all array values (a.product)

Signature: `{"a.product": [a]}`

`a`	array of any `A` of {int, long, float, double}
(returns)	`A`

Description: Return the product of numbers in `a`.

Details:

Returns one if the array is empty.

11.6.3 Sum of logarithms (a.lnsum)

Signature: `{"a.lnsum": [a]}`

`a`	array of double
(returns)	double

Description: Return the sum of the natural logarithm of numbers in `a`.

Details:

Returns zero if the array is empty and `NaN` if any value in the array is zero or negative.

11.6.4 Arithmetic mean (a.mean)

Signature: `{"a.mean": [a]}`

`a`	array of double
(returns)	double

Description: Return the arithmetic mean of numbers in `a`.

Details:

Returns `NaN` if the array is empty.

11.6.5 Geometric mean (a.geomean)

Signature: `{"a.geomean": [a]}`

`a`	array of double
(returns)	double

Description: Return the geometric mean of numbers in `a`.

Details:

Returns `NaN` if the array is empty.

11.6.6 Median (a.median)

Signature: `{"a.median": [a]}`

`a`	array of any `A`
(returns)	`A`

Description: Return the value that is in the center of a sorted version of `a`.

Details:

If `a` has an odd number of elements, the median is the exact center of the sorted array. If `a` has an even number of elements and is a `float` or `double`, the median is the average of the two elements closest to the center of the sorted array. For any other type, the median is the left (first) of the two elements closest to the center of the sorted array.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.6.7 Mode, or most common value (a.mode)

Signature: `{"a.mode": [a]}`

`a`	array of any `A`
(returns)	`A`

Description: Return the mode (most common) value of `a`.

Details:

If several different values are equally common, the median of these is returned.

Runtime Errors:

If `a` is empty, an "empty array" runtime error is raised.

11.7 Set or set-like functions

PFA does not have a set datatype, but arrays can be interpreted as sets with the following functions.

11.7.1 Distinct items (a.distinct)

Signature: `{"a.distinct": [a]}`

`a`	array of any `A`
(returns)	array of `A`

Description: Return an array with the same contents as `a` but with duplicates removed.

11.7.2 Set equality (a.seteq)

Signature: `{"a.seteq": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	boolean

Description: Return `true` if `a` and `b` are equivalent, ignoring order and duplicates, `false` otherwise.

11.7.3 Union (a.union)

Signature: `{"a.union": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	array of `A`

Description: Return an array that represents the union of `a` and `b`, treated as sets (ignoring order and duplicates).

11.7.4 Intersection (a.intersect)

Signature: `{"a.intersect": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	array of `A`

Description: Return an array that represents the intersection of `a` and `b`, treated as sets (ignoring order and duplicates).

11.7.5 Set difference (a.diff)

Signature: `{"a.diff": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	array of `A`

Description: Return an array that represents the difference of `a` and `b`, treated as sets (ignoring order and duplicates).

11.7.6 Symmetric set difference (a.symdiff)

Signature: `{"a.symdiff": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	array of `A`

Description: Return an array that represents the symmetric difference of `a` and `b`, treated as sets (ignoring order and duplicates).

Details:

The symmetric difference is (`a` diff `b`) union (`b` diff `a`).

11.7.7 Subset check (a.subset)

Signature: `{"a.subset": [little, big]}`

`little`	array of any `A`
`big`	array of `A`
(returns)	boolean

Description: Return `true` if `little` is a subset of `big`, `false` otherwise.

11.7.8 Disjointness check (a.disjoint)

Signature: `{"a.disjoint": [a, b]}`

`a`	array of any `A`
`b`	array of `A`
(returns)	boolean

Description: Return `true` if `a` and `b` are disjoint, `false` otherwise.

11.8 Functional programming

11.8.1 Map array items with function (a.map)

Signature: `{"a.map": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → any `B`
(returns)	array of `B`

Description: Apply `fcn` to each element of `a` and return an array of the results.

Details:

The order in which `fcn` is called on elements of `a` is not guaranteed, though it will be called exactly once for each element.

11.8.2 Filter array items with function (a.filter)

Signature: `{"a.filter": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → boolean
(returns)	array of `A`

Description: Apply `fcn` to each element of `a` and return an array of the elements for which `fcn` returns `true`.

Details:

The order in which `fcn` is called on elements of `a` is not guaranteed, though it will be called exactly once for each element.

11.8.3 Filter and map (a.filtermap)

Signature: `{"a.filtermap": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → union of {any `B`, null}
(returns)	array of `B`

Description: Apply `fcn` to each element of `a` and return an array of the results that are not `null`.

Details:

The order in which `fcn` is called on elements of `a` is not guaranteed, though it will be called exactly once for each element.

11.8.4 Map and flatten (a.flatmap)

Signature: `{"a.flatmap": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → array of any `B`
(returns)	array of `B`

Description: Apply `fcn` to each element of `a` and flatten the resulting arrays into a single array.

Details:

The order in which `fcn` is called on elements of `a` is not guaranteed, though it will be called exactly once for each element.

11.8.5 Reduce array items to a single value (a.reduce)

Signature: `{"a.reduce": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`, `A`) → `A`
(returns)	`A`

Description: Apply `fcn` to each element of `a` and accumulate a tally.

Details:

The first parameter of `fcn` is the running tally and the second parameter is an element from `a`.

The order in which `fcn` is called on elements of `a` is not guaranteed, though it accumulates from left (beginning) to right (end), called exactly once for each element. For predictable results, `fcn` should be associative. It need not be commutative.

11.8.6 Right-to-left reduce (a.reduceright)

Signature: `{"a.reduceright": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`, `A`) → `A`
(returns)	`A`

Description: Apply `fcn` to each element of `a` and accumulate a tally.

Details:

The first parameter of `fcn` is an element from `a` and the second parameter is the running tally.

The order in which `fcn` is called on elements of `a` is not guaranteed, though it accumulates from right (end) to left (beginning), called exactly once for each element. For predictable results, `fcn` should be associative. It need not be commutative.

11.8.7 Fold array items to another type (a.fold)

Signature: `{"a.fold": [a, zero, fcn]}`

`a`	array of any `A`
`zero`	any `B`
`fcn`	function (`B`, `A`) → `B`
(returns)	`B`

Description: Apply `fcn` to each element of `a` and accumulate a tally, starting with `zero`.

Details:

The first parameter of `fcn` is the running tally and the second parameter is an element from `a`.

The order in which `fcn` is called on elements of `a` is not guaranteed, though it accumulates from left (beginning) to right (end), called exactly once for each element. For predictable results, `fcn` should be associative with `zero` as its identity; that is, `fcn(zero, zero) = zero`. It need not be commutative.

11.8.8 Right-to-left fold (a.foldright)

Signature: `{"a.foldright": [a, zero, fcn]}`

`a`	array of any `A`
`zero`	any `B`
`fcn`	function (`B`, `A`) → `B`
(returns)	`B`

Description: Apply `fcn` to each element of `a` and accumulate a tally, starting with `zero`.

Details:

The first parameter of `fcn` is an element from `a` and the second parameter is the running tally.

The order in which `fcn` is called on elements of `a` is not guaranteed, though it accumulates from right (end) to left (beginning), called exactly once for each element. For predictable results, `fcn` should be associative with `zero` as its identity; that is, `fcn(zero, zero) = zero`. It need not be commutative.

11.8.9 Take items until predicate is false (a.takeWhile)

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

`a`	array of any `A`
`fcn`	function (`A`) → boolean
(returns)	array of `A`

Description: Apply `fcn` to elements of `a` and create an array of the longest prefix that returns `true`, stopping with the first `false`.

Details:

Beyond the prefix, the number of `fcn` calls is not guaranteed.

11.8.10 Drop items until predicate is true (a.dropWhile)

Signature: `{"a.dropWhile": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → boolean
(returns)	array of `A`

Description: Apply `fcn` to elements of `a` and create an array of all elements after the longest prefix that returns `true`.

Details:

Beyond the prefix, the number of `fcn` calls is not guaranteed.

11.9 Functional tests

11.9.1 Existential check, ∃ (a.any)

Signature: `{"a.any": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → boolean
(returns)	boolean

Description: Return `true` if `fcn` is `true` for any element in `a` (logical or).

Details:

The number of `fcn` calls is not guaranteed.

11.9.2 Univeral check, ∀ (a.all)

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

`a`	array of any `A`
`fcn`	function (`A`) → boolean
(returns)	boolean

Description: Return `true` if `fcn` is `true` for all elements in `a` (logical and).

Details:

The number of `fcn` calls is not guaranteed.

11.9.3 Pairwise check of two arrays (a.corresponds)

Signature: `{"a.corresponds": [a, b, fcn]}`

`a`	array of any `A`
`b`	array of any `B`
`fcn`	function (`A`, `B`) → boolean
(returns)	boolean

Description: Return `true` if `fcn` is `true` when applied to all pairs of elements, one from `a` and the other from `b` (logical relation).

Details:

The number of `fcn` calls is not guaranteed.

If the lengths of `a` and `b` are not equal, this function returns `false`.

11.10 Restructuring

11.10.1 Sliding window (a.slidingWindow)

Signature: `{"a.slidingWindow": [a, size, step, allowIncomplete]}`

`a`	array of any `A`
`size`	int
`step`	int
`allowIncomplete`	boolean
(returns)	array of array of `A`

Description: Return an array of subsequences of `a` with length `size` that slide through `a` in steps of length `step` from left to right.

Details:

If `allowIncomplete` is `true`, the last window may be smaller than `size`. If `false`, the last window may be skipped.

Runtime Errors:

If `size` is non-positive, a "size < 1" runtime error is raised.

If `step` is non-positive, a "step < 1" runtime error is raised.

11.10.2 Unique combinations of a fixed size (a.combinations)

Signature: `{"a.combinations": [a, size]}`

`a`	array of any `A`
`size`	int
(returns)	array of array of `A`

Description: Return the unique combinations of `a` with length `size`.

Runtime Errors:

If `size` is non-positive, a "size < 1" runtime error is raised.

11.10.3 Permutations (a.permutations)

Signature: `{"a.permutations": [a]}`

`a`	array of any `A`
(returns)	array of array of `A`

Description: Return the permutations of `a`.

Details:

This function scales rapidly with the length of the array. For reasonably large arrays, it will result in timeout exceptions.

11.10.4 Flatten array (a.flatten)

Signature: `{"a.flatten": [a]}`

`a`	array of array of any `A`
(returns)	array of `A`

Description: Concatenate the arrays in `a`.

11.10.5 Group items by category (a.groupby)

Signature: `{"a.groupby": [a, fcn]}`

`a`	array of any `A`
`fcn`	function (`A`) → string
(returns)	map of array of `A`

Description: Groups elements of `a` by the string that `fcn` maps them to.

12 Manipulation of other data structures

12.1 Map

12.2 Record

12.3 Enum

12.4 Fixed

13 Missing data handling

13.1 Impute library

13.1.1 Skip record (impute.errorOnNull)

Signature: `{"impute.errorOnNull": [x]}`

`x`	union of {any `A`, null}
(returns)	`A`

Description: Skip an action by raising an "encountered null" runtime error when `x` is `null`.

13.1.2 Replace with default (impute.defaultOnNull)

Signature: `{"impute.defaultOnNull": [x, default]}`

`x`	union of {any `A`, null}
`default`	`A`
(returns)	`A`

Description: Replace `null` values in `x` with `default`.

14 Aggregation

SQL-like functions

group-by tables

CUSUM

15 Descriptive statistics libraries

15.1 Sample statistics

15.1.1 Update aggregated mean (stat.sample.updateMean)

Signature: `{"stat.sample.updateMean": [runningSum, w, x]}`

`runningSum`	any record `A` with {`sum_w:` double, `sum_wx:` double}
`w`	double
`x`	double
(returns)	`A`

Description: Update a record containing running sums for computing a sample mean.

Parameters:

`runningSum`	Record of partial sums: `sum_w` is the sum of weights, `sum_wx` is the sum of weights times sample values.
`w`	Weight for this sample, which should be 1 for an unweighted mean.
`x`	Sample value.

Details:

Use `stat.sample.mean` to get the mean.

15.1.2 Compute aggregated mean (stat.sample.mean)

Signature: `{"stat.sample.mean": [runningSum]}`

`runningSum`	any record `A` with {`sum_w:` double, `sum_wx:` double}
(returns)	double

Description: Compute the mean from a `runningSum` record.

Details:

Use `stat.sample.updateMean` to fill the record.

accumulated mean, median(?)

16 Data mining models

16.1 Decision and regression Trees

16.1.1 Tree walk with simple predicates (model.tree.simpleWalk)

Signature: `{"model.tree.simpleWalk": [datum, treeNode]}`

`datum`	any record `D`
`treeNode`	any record `T` with {`field:` string, `operator:` string, `value:` any `V`, `pass:` union of {`T`, any `S`}, `fail:` union of {`T`, `S`}}
(returns)	`S`

Description: Descend through a tree comparing `datum` to each branch with a simple predicate, stopping at a leaf of type `S`.

Parameters:

`datum`	An element of the dataset to score with the tree.
`treeNode`	A node of the decision or regression tree.
	`field:` Indicates the field of `datum` to test. Fields may have any type.
	`operator:` One of "==" (equal), "!=" (not equal), "<" (less than), "<=" (less or equal), ">" (greater than), or ">=" (greater or equal).
	`value:` Value for comparison. Should be the union of or otherwise broader than all `datum` fields under consideration.
	`pass:` Branch to return if field `field` of `datum` `operator` `value` yields `true`.
	`fail:` Branch to return if field `field` of `datum` `operator` `value` yields `false`.
(return value)	The score associated with the destination leaf, which may be any type `S`. If `S` is a `string`, this is generally called a decision tree; if a `double`, it is a regression tree; if an `array` of `double`, a multivariate regression tree, etc.

Runtime Errors:

Raises a "no such field" error if `field` is not a field of `datum`.

Raises an "invalid comparison operator" error if `operator` is not one of "==", "!=", "<", "<=", ">", or ">=".

Raises a "bad value type" error if the `field` of `datum` cannot be upcast to `V`.

16.1.2 Tree walk with user-defined predicates (model.tree.predicateWalk)

Signature: `{"model.tree.predicateWalk": [datum, treeNode, predicate]}`

`datum`	any record `D`
`treeNode`	any record `T` with {`pass:` union of {`T`, any `S`}, `fail:` union of {`T`, `S`}}
`predicate`	function (`D`, `T`) → boolean
(returns)	`S`

Description: Descend through a tree comparing `datum` to each branch with a user-defined predicate, stopping at a leaf of type `S`.

Parameters:

`datum`	An element of the dataset to score with the tree.
`treeNode`	A node of the decision or regression tree.
	`pass:` Branch to return if `"predicate": ["datum", "treeNode"]` yields `true`.
	`fail:` Branch to return if `"predicate": ["datum", "treeNode"]` yields `false`.
(return value)	The score associated with the destination leaf, which may be any type `S`. If `S` is a `string`, this is generally called a decision tree; if a `double`, it is a regression tree; if an `array` of `double`, a multivariate regression tree, etc.

16.2 Cluster models

16.3 Regression

16.4 Neural networks

16.5 Support vector machines