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, Storm, or Akka), 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 raised. 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

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 unifying 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 structure 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 accommodate 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 six 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`**.
- Options are JSON object member values of the **`options`** top-level field and may be overridden by the PFA system. They all have well-defined defaults and unique, hierarchical names.

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.

Options may also be implemented atomically by name. If a named option is not implemented, the system should behave as though that option had its default value, regardless of whether the option is explicitly set in the PFA document. Options can in general be overridden by a host system, so if a host system doesn't implement an option, it is as though the system enforces the default.

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, though it is not strictly 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 carefully chosen ad hoc subset or complete conformance.

1.5 Reference implementations

Two open-source reference implementations are provided to help clarify this specification. They can be accessed on `http://scoringengine.org` or GitHub.

pfa-jvm: A PFA system written in Scala for the Java Virtual Machine (JVM). It performs all necessary checks and dynamically compiles PFA documents into JVM bytecode for fast execution. It has group-id **`org.scoringengine`** and artifact-id **`pfa`** and (will be) submitted to the Sonatype repository for use in common build tools.

pfa-py: A PFA system written Python. It performs all necessary checks, interprets PFA documents for testing, and transforms Python-based machine learning outputs into PFA. It (will be) submitted to the Python Package Index with package name **`pfa`**.

Although these implementations are intended to help clarify the intent of the specification for future implementations, they are not normative definitions of the standard. Only this document (and those that may supersede it) are normative. All versions of this document, including the latest version, can be found on the websites listed above.

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. 3.2). If absent, the default value is "map".

`input:` An Avro schema representing the data type of data provided to the scoring engine (see Sec. 3.3).

`output:` An Avro schema representing the data type of data produced by the scoring engine (see Sec. 3.3). 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. 3.1).

`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. 3.1).

`end:` An expression or JSON array of expressions that are executed in the end phase of the scoring engine's run (see Sec. 3.1).

`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. If **`method`** is "map" or "emit", this field is ignored.

`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 set of possible options and their representation is restricted:: see see Sec. 3.7.

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 small 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 record’s fields, and returns the string it finds at the tree’s leaf. (See the definition of the model.tree.simpleWalk function.)

```
{"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**.

rollback: An optional boolean specifying whether this cell should be rolled back to the state it had at the beginning of an **action** if an exception occurs during the **action**. The default is **false**, and **shared** and **rollback** are mutually incompatible: they cannot both be **true**.

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**.

rollback: An optional boolean specifying whether this pool should be rolled back to the state it had at the beginning of an **action** if an exception occurs during the **action**. The default is **false**, and **shared** and **rollback** are mutually incompatible: they cannot both be **true**.

Cell and pool names must match the following pattern: `[A-Za-z_][A-Za-z0-9_]*` (the first character must be a letter or underscore; subsequent characters, if they exist, may also be numbers). Cells and pools do not share a namespace with each other or with symbols or functions.

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

2.2 Locator marks

PFA documents are usually generated by an automated process, such as by encoding a machine learning decider into a scoring engine or by transforming user functions from an easily readable language into the terse PFA representation. In the latter case, there can be a cognitive disconnect between the language in which the user writes code, for instance a regression fit function written in Python, and the auto-generated PFA. In particular, if there is an error in the generated PFA due to an error in the original source code, reporting the error at the line and column number of the generated PFA would not be useful for the Python programmer. It would be much better if a PFA system could report such an error at the location of the offending line in the original source code.

To allow individual PFA systems to do this, the PFA specification allows for "@" as a JSON member name in any JSON object in the PFA document. The associated member value must be a string, and would ordinarily be a description of the line number of the original source that generated that JSON object in the PFA document. These "@" key-value pairs can appear in any object at any level, including Avro type schemae and embedded JSON data. If the library that interprets Avro type schemae and embedded JSON data does not ignore object members named “@”, then the PFA system must strip these objects before passing the JSON objects for interpretation.

The generation of and meaning of these locator marks are beyond the scope of the PFA specification—different PFA systems can generate and interpret them differently, or not at all. However, to ensure that the locator marks made by one system do not cause unnecessary errors in another system, the locator marks must abide by the following rules.

- JSON object members named “@” must not cause errors in any PFA system, even if they appear in Avro type schemae or embedded JSON objects.
- The value associated with this key must be a string.
- The system that generates these marks should place them at the beginning of each JSON object—that is, in the serialized form of the JSON data, the “@” member name should appear immediately after the “{“ character that starts the JSON object (apart from any whitespace). This is because some PFA readers may interpret the JSON data as it streams into a buffer, and they may encounter an error before reaching the last JSON object member. If the locator mark is not first, PFA systems cannot be expected to use it.

3 Scoring engine execution model

A PFA document (string of JSON-formatted text) describes a PFA scoring engine (executable routine) or a battery of initially identical engines. An engine behaves as a single-threaded executable with global state (cells and pools) and local variables. A battery of scoring engines may run in parallel and only share data if some cells or pools are explicitly marked as **`shared`**. Although a battery of scoring engines generated by a single PFA document start in exactly the same state, they may evolve into different states if they have any unshared cells or pools.

PFA engines are units of work that may fit into a pipeline system like Hadoop, Storm, or Akka. In a map-reduce framework such as Hadoop, for instance, one PFA document could describe the calculation performed by all of the mappers and another could describe the calculation performed by all of the reducers. The mappers are a battery of independent PFA engines, as are the reducers. In pure map-reduce, the mappers would not communicate with each other and the reducers would not communicate with each other, so none of the cells or pools should be marked as **`shared`**. With this separation of concerns, issues of transferring data, interpreting input file types, and formatting output should be handled by the pipeline system (Hadoop in this case) while the mathematical procedure is handled by PFA. Changing file formats would require an update to the pipeline code (and possibly a code review), but changing details of the analytic would only require a new PFA document (a JSON configuration file).

3.1 Execution phases of a PFA scoring engine

A PFA engine has a 7 phase lifecycle. These phases are the following, executed in this order:

1. reading the PFA document and performing a syntax check;
2. verifying PFA invariants and checking type consistency;
3. additional checks, constraints required by a particular PFA system;
4. initialization of the engine;
5. execution of the **`begin`** routine;
6. execution of the **`action`** routine for each input datum;
7. execution of the **`end`** routine.

In phase 1, JSON is decoded and may be used to build an abstract syntax tree of the whole document. At this stage, JSON types must be correctly matched (e.g. if a number is expected, a string cannot be provided instead) to build the syntax tree. Incorrectly formatted JSON should also be rejected, though we recommend that a dedicated JSON decoder is used for this task. Avro schemae should also be interpreted in this phase (see Sec. 4.1).

In phase 2, the loaded PFA document is interpreted as an executable. If the specific PFA implementation builds code with macros, compiles bytecode, or synthesizes a circuit for execution, that work should happen in this phase. Data types should be inferred and checked (see Sec. 4.3), especially if the executable is compiled.

Phase 3 is provided for optional checks. Due to limitations of a particular environment, some PFA systems may need to be more restrictive than the general specification and reject what would otherwise be a valid PFA document. Reasons include unimplemented function calls, inability to implement recursion, or data structures that are too large. The phase 3 checks may need to be performed concurrently with the phase 2 checks to build the executable.

Phase 4, initialization, is when data structures such as cells and pools are allocated and filled, network connections are established (if relevant for a particular PFA implementation), pseudorandom number generators are seeded, etc. These are actions that the engine must perform to work properly but are not a part of the **`begin`**, **`action`**, or **`end`** routines.

The actions performed in the last three phases, **`begin`**, **`action`**, and **`end`**, are explicitly defined in the PFA document. A PFA system must implement the **`action`** phase, since every PFA document must define an **`action`**. The **`action`** accepts input and returns output, though the way it does so depends on the **`method`** (Sec. 3.2).

The **`begin`** and **`end`** phases do not accept input and do not return output: they can only modify cells and pools, emit log messages, or raise exceptions. A PFA system is not required to implement **`begin`** and **`end`**. If a system that does not implement **`begin`** encounters a document that has a **`begin`** routine, it must fail with an error. If a system that does not implement **`end`** encounters a document that has an **`end`** routine, it need not fail with an error, though it may. This is because some PFA documents may use **`begin`** to initialize essential data structures and the **`action`** would only function properly if **`begin`** has been executed, but the **`end`** routine can only affect the state of a completed scoring engine whose interpretation is implementation-specific. Moreover, some data pipelines do not even have a concept of completion, such as Storm.

After all input data have been passed to the scoring engine and the last **`action`** or **`end`** routine has finished, the scoring engine is said to be completed. This may be considered an eighth phase of the engine, though its behavior at this point is not defined by this specification. A particular PFA system may extract aggregated results from a completed engine's state and it may even call functions defined in the document's **`fcn`** field, but this is beyond the scope of the standard PFA lifecycle. (Note: if the primary purpose of a scoring engine is to aggregate data, consider using the "fold" **`method`** instead of extracting from the engine's internal state.)

A completed scoring engine may be used to create a new PFA document, in which the final state of the cells and pools are used to define the **`cell init`** or **`pool init`** of the new document, such that a new scoring engine would start where the old one left off. A PFA system may even re-use an old scoring engine as a new scoring engine (repeating phase 4 onward), but a re-used engine must behave exactly like a new engine with copied state, such that the re-use is an implementation detail and does not affect behavior.

A PFA system may call functions defined in the document's **`fcn`** field at any time, but if the function modifies state (a "cell-to" or "pool-to" special form is reachable in its call graph) and the engine is not complete, the function call must not be allowed because it could affect the engine's behavior. A PFA system must not execute **`begin`**, **`action`**, or **`end`** outside of its lifecycle.

3.2 Scoring method: map, emit, and fold

PFA defines the following three methods for calling the **`action`** routine of a scoring engine.

"map": The **`action`** routine is given an **`input`** value, which it uses to construct and return an output. Barring exceptions, the output dataset would have exactly as many values as the input dataset.

"emit": The **`action`** routine is given an **`input`** value and an **`emit`** callback function, and the functional return value is ignored. The scoring engine returns results to the host system by calling **`emit`**. It can call **`emit`** any number of times, and thus the output dataset may be smaller or larger than the input dataset. For example, a filter would call **`emit`** zero or one times for each input.

"fold": The **action** routine is given an **input** value and a **tally** value, which it uses to construct and return an output. The first time **action** is invoked, **tally** is equal to **zero** (the top-level field). On the N^{th} time **action** is invoked, **tally** is equal to the $(N-1)^{\text{th}}$ return value. Thus, a "fold" scoring engine is an aggregator: transformed inputs may be counted, summed, maximized, or otherwise accumulated in the **tally**. The aggregate of the entire input dataset is the last return value of **action**, though the host system may make use of partial sums as well.

For all three methods, the **input** is available to expressions as a read-only symbol that can be accessed in an expression as the JSON string **"input"** (see Sec. 7.1). The **input** symbol's scope is limited to the **action** routine: it is not accessible in user-defined functions unless explicitly passed. The **input** symbol's data type is specified by the top-level field named **input**.

For the "map" and "fold" methods, the data type of the last expression in the **action** routine must be the type specified by the top-level field named **output**. For the "emit" method, there is no constraint on the type of the last expression in **action**, but the argument passed to the **emit** function must have **output** type.

For the "emit" method, the **emit** function is a globally accessible function. It may be called or referenced without qualification by any user-defined function or even in the **begin** and **end** routines.

For the "fold" method, the **tally** is available to expressions as a read-only symbol that can be accessed in an expression as the JSON string **"tally"** (see Sec. 7.1). The **tally** symbol's scope is limited to the **action** routine: it is not accessible in user-defined functions unless explicitly passed. The **tally** symbol's data type is specified by the **output** top-level field. The top-level field named **zero** must also have **output** type.

The means by which input values are provided to the scoring engine, output values are retrieved, and the **emit** function is set or changed are all unspecified. A PFA system may change the **emit** function at any time, even while an **action** is being processed (though we do not recommend this). However, the **emit** function must be defined and callable at all times during the **begin**, **action**, and **end** phases of the scoring engine's lifecycle.

3.3 Input and output type specification

The member values of the top-level fields **input** and **output** are Avro schemae (see Sec. 4.1). The way that these types constrain the input and output of scoring engines depends on the **method** and is described in Sec. 3.2.

The input data provided to the scoring engine must conform to the **input** type in the sense that there must be an unambiguous way to generate it from Avro-encoded data, though this conversion need not actually take place. For example, if the **input** is **{"type": "array", "items": "int"}**, then the values passed to the scoring engine must be ordered lists of integers, though they may be implemented as arrays, linked lists, immutable vectors, or any other functionally equivalent data structure that the PFA implementation is capable of using in calculations. The data source need not be Avro-encoded; the Avro schema is only used to specify the type, not to perform conversions. Similarly, the output data must conform to the **output** type in the sense that there must be an unambiguous way to convert it to Avro-encoded data, though this conversion need not actually take place.

Given that the input and output types are described by Avro schemae, the Avro binary and JSON data formats would be particularly convenient ways to read and write data. However, there is no requirement that a PFA system should have this capability. Data conversion and internal data format are both outside the scope of the PFA specification.

3.4 Persistent state: cells and pools

PFA defines two mechanisms to maintain state: cells and pools. Cells are global variables with a fixed name and type that must be initialized before the scoring engine's run begins. A cell's value can change to a new value of the same type, but the cell cannot be deleted and new cells cannot be created during the scoring engine's run. Pools are global namespaces with a fixed type. New named values can be created within a pool at runtime, as long as they have the correct type, and old values can be deleted at runtime.

A pool of type "**X**" would be equivalent to a cell whose type is "map of **X**" except for performance and concurrency issues. All special forms and library functions in the PFA specification treat data structures as immutable objects (see Sec. 5.1), but scoring engines often need to maintain very large key-value tables. If pools were not available, a PFA implementation would either incur a performance penalty if it maintained a large map in a cell as an immutable object or if it maintained all temporary variables as mutable objects. With both cells and pools, a PFA implementation may maintain all values as immutable objects, including cells, and maintain pools as mutable maps of immutable objects. See Sec. 3.5 for a discussion of concurrency issues in cells and pools.

Cells can only be accessed through the "cell" special form and can only be modified through the "cell-to" special form. Pools can only be accessed through the "pool" special form and modified through the "pool-to" special form (see Sec. 7.8).

One common use of persistent state is to represent a complex statistical model, such as a large decision tree. In most cases, such a model is constant during the scoring engine's run, and this constraint may be enforced through static analysis if the model is stored in a cell. Another common use is to represent recent or accumulated data in a table, indexed by key. In most cases, this table is updated frequently and new table entries may be added at any time. Furthermore, it is often useful to distribute the table-fill operation among a battery of concurrent scoring engines, with different engines modifying different keys at the same time. These cases are more easily implemented as pools or shared pools.

3.5 Concurrent access to shared state

If the **shared** member of a cell or pool's specification is **true**, the cell or pool is not assumed to be thread-local (see Sec. 2.1). It may be shared among a battery of identical scoring engines in a multi-threaded process, shared among identical scoring engines distributed across a network, shared in a database among different types of processes, or shared among components of an integrated circuit, etc. In any case, some rules must be followed to avoid simultaneous attempts to modify the data, and these rules must be standardized to ensure that the same scoring engine has the same behavior on different systems.

Shared cells and pools in PFA follow a read-copy-update rule for concurrent access: attempts to read the shared resource (through the "cell" or "pool" special forms) always succeed without blocking and attempts to write (through the "cell-to" or "pool-to" special forms) lock the resource or wait until another writer's lock is released. The writers must operate on a copy of the cell or pool's data (or on immutable data) so that readers can access the old version during the update process. The new value must be updated atomically at the end of the update process.

Although an update operation may only modify a part of the cell's structure (one value in an array, for instance), the granularity of the writers' lock is the entire cell: two writers must not be able to modify different parts of the same cell at the same time. The granularity of the writers' lock on pools is limited to a single named entity within the pool: two writers must be able to modify different entities in the pool at the same time, but not different parts of the same named entity.

The "cell-to" and "pool-to" special forms accept user-defined update functions (see Sec. 7.8) but these functions must not directly or indirectly call "cell-to" or "pool-to" because such a situation could lead to deadlock. This rule can be enforced by examining the call graph.

3.6 Exceptions

As much as is reasonably possible, PFA documents can be statically analyzed to avoid errors at runtime. However, some error states cannot be predicted without runtime information. These error states and their exact messages are explicitly defined for each susceptible special form and library function. If the specified error conditions are met in the `begin` routine of a scoring engine, processing stops and should not continue to the `action` routine. If error conditions occur when the `action` routine is processing an input datum, processing of that datum stops and may either continue to the next datum or stop the scoring engine entirely. The PFA host may choose to stop or continue on the basis of the standardized error message. If error conditions occur in the `end` routine, processing stops.

This abrupt end of processing may occur deep in an expression or array of expressions and behaves like an exception: control flow exits the routine immediately upon encountering the error condition and is either caught by the host system or it halts the process. In environments where this is difficult to implement, control flow may continue to the end of the routine, but side-effects such as modifications to persistent state and log messages must be avoided.

If the host system catches an exception in `action` and continues to the next datum, and if a cell or pool's `rollback` member is `true`, then that cell or pool should be reverted to the value that it had at the beginning of the `action` (see Sec. 2.1). If the `rollback` member is absent or `false`, then the cell or pool's value at the start of the next `action` should be the value it had at the time of the exception.

In addition to exceptions raised by special forms and library functions, a PFA document can raise custom exceptions with the "error" special form (see Sec. 7.13.2). The rules described above apply equally to custom exceptions, though we recommend that PFA systems differentiate between built-in exceptions (whose error messages are explicitly defined by this specification) and custom exceptions (whose error messages are free-form).

If a `timeout` is defined in the PFA document's `options` or is imposed by the PFA system, a `begin`, `action`, or `end` routine that exceeds this timeout raises an exception with the message "exceeded timeout of N milliseconds" where N is the relevant timeout. Timeout exceptions follow the same rules as built-in and custom exceptions.

If possible in a given system, no exceptions other than PFA exceptions should ever be raised while executing a `begin`, `action`, or `end` routine.

3.7 Execution options

The `options` top-level field allows PFA documents to request that they are executed in a particular way. However, the PFA system may override any of these options with its own values, or with the defaults. When overriding an option, the PFA system should somehow indicate that this is the case, possibly through a log message.

The complete set of options, their JSON types, and their default values are given below. If a PFA document attempts to set an option that is not in this list or attempts to set an option with the wrong type, it is a semantic error (phase 2 in see Sec. 3.1) and the scoring engine should not be started.

Option name	JSON type	Default value	Description
`timeout`	integer	-1	Number of milliseconds to let the `begin`, `action`, or `end` routine run; at or after this time, the PFA system may stop the routine with an exception (see Sec. 3.6). If negative, the execution has no timeout.

Option name	JSON type	Default value	Description
`timeout.begin`	integer	**`timeout`**	A specific timeout for the **`begin`** routine that overrides the general **`timeout`**.
`timeout.action`	integer	**`timeout`**	A specific timeout for the **`action`** routine that overrides the general **`timeout`**.
`timeout.end`	integer	**`timeout`**	A specific timeout for the **`end`** routine that overrides the general **`timeout`**.

3.8 Pseudorandom number management

The **`randseed`** top-level field specifies a seed for library functions that generate pseudorandom numbers. If the **`randseed`** is absent, the random number generator should be unpredictable: multiple runs of the same PFA document would yield different results if the output depends on pseudorandom numbers. If a **`randseed`** is provided, the random number generator should be predictable: multiple runs of the same PFA document would yield the same results on the same system. Explicitly setting a **`randseed`** is useful for tests.

The pseudorandom number generator maintains state between **`begin`**, **`action`**, and **`end`** invocations: the generator is not reseeded with each call. If a PFA document is used to create a battery of identical scoring engines, the **`randseed`** is used to generate different seeds for all of the scoring engines: they are not guaranteed to produce identical results.

The algorithm for generating pseudorandom numbers is not specified, so different PFA implementations may use different algorithms. Therefore, a PFA document with an explicit **`randseed`** is only guaranteed to yield identical results when rerun on the same system. On different systems, it may yield different results.

Every library function that depends on pseudorandom numbers should be seeded by the **`randseed`**. Pseudorandom functions are explicitly denoted by this specification.

4 Type system

Rather than invent a new type system, PFA uses Avro type schemae to describe its data types. Avro is a serialization format, but it also describes the types (sets of possible values) that are to be serialized or unserialized with JSON-based schemae. Feeding Avro-formatted data into and out of a PFA scoring engine is particularly easy, since the sets of possible values that can be Avro-serialized perfectly align with the sets of possible values that PFA can use in its calculations.

However, Avro serialization is by no means necessary to use with PFA: data that can be described by Avro types can be serialized many different ways. In fact, the Avro project provides two: a binary format and a JSON format. With the appropriate translations, CSV can be converted to and from a subset of Avro types, XML can be fully and reversibly transformed, as can many popular data formats. The transformation of data formats and the internal representation of data in a PFA implementation are beyond the scope of this specification and should be handled in any way that the designer of a PFA system sees fit.

4.1 Avro types

The normative definition of Avro 1.7.6 types and type schemae is provided online. However, the basics are duplicated here for convenience. This section is non-normative.

The set of all expressible types is the closure under the following primitives and parameterized types.

null: A type with only one value, `null`. This is a unit type, and is usually only useful when combined with other types in a union (see below) or as a return type for functions that do not have a meaningful value to return.

boolean: A type with only two values, `true` and `false`.

int: Signed whole numbers with a 32-bit range: from -2147483648 to 2147483647 inclusive.

long: Signed whole numbers with a 64-bit range: from -9223372036854775808 to 9223372036854775807 inclusive.

float: Signed fractional numbers with 32-bit binary precision as defined by IEEE 754.

double: Signed fractional numbers with 64-bit binary precision according to the same standard.

string: Strings of text characters that can be encoded in Unicode.

bytes: Arrays of uninterpreted bytes with any length.

fixed(`L`, `N`, `NS`): Named arrays of uninterpreted bytes with length `L` (integer), name `N` (string), and optional namespace `NS` (string).

enum(`S`, `N`, `NS`): Named enumeration of a finite set of symbols `S` (ordered list of strings), name `N` (string), and optional namespace `NS` (string).

array(`X`): Homogeneous array of type `X` (Avro type) with any length.

map(`X`): Homogeneous map from strings to type `X` (Avro type). The keys of all maps must be strings (just like JSON).

record(`F`, `N`, `NS`): Heterogeneous record of fields `F` (named slots with Avro types) with name `N` (string) and optional namespace `NS` (string). This is a product type; the possible values that it can have is the Cartesian product of the possible values that each field can have.

union(`T`): Union of types `T` (array of Avro types). This is a sum type; the possible values that it can have is union of the values of each type in `T`.

This type system has the following limitations and remediations.

- Arrays and maps must be homogeneous (all elements have the same, specified type). This is sufficient for most mathematical applications and the restriction helps to eliminate common mistakes. Also, it makes some significant optimizations possible that are difficult or impossible in dynamic languages.
- Map keys must be strings. If an application must represent a map whose keys are not strings, one can define a unique string representation for each key and look up items by first transforming to the string representation.
- There is no set or multiset type. This can be emulated with arrays and PFA's set-like functions or a map from string-valued keys to **`null`**.
- Circular references are not possible, as there are no pointers or references. However, data structures with conceptual loops can be emulated through weak references in a map. For example, an arbitrary directed graph can be described as a map of arrays of strings: each key is a node and each element of an array is a link to another node. These references are weak because there is no guarantee that a key exists for every array element.

  The lack of circular references is, in some ways, an advantage. Non-circular data can be more easily serialized without the possibility of infinite loops. Immutable data can take advantage of more structural sharing since data structures are purely tree-like.

  Note that recursively defined types *are* possible. A record **`X`** could have one or more fields that are unions of **`X`** and **`Y`**, or it could have a field that is an array or map of **`X`**. The first case would describe a tree of nodes **`X`** with a fixed number of named branches, terminating in leaves of type **`Y`**. (This is how decision trees are described in PFA; the scores have type **`Y`**.) The second case would describe a tree of nodes **`X`** with arbitrarily many branches at each node, terminating in empty arrays or maps.
- To make a type such as **string** nullable, one must construct a union of **string** and **null**. This union type cannot be passed into functions that expect a **string**.

  Again, this restriction can be an advantage. It is often known as a type-safe null: in the example above, string functions can still be used, but only after explicitly handling the **null** case. In PFA, one would use a cast-cases special form to split the program flow into a branch that handles the **string** case and a branch that handles the **null** case, usually by specifying a rule that replaces **`null`** with a string. This restriction eliminates the possibility of null pointer exceptions at runtime.

  Most library functions in PFA interpret **`null`** as a missing value. Missing value handling is an important consideration in many statistical analyses, so PFA has a suite of functions for addressing this case.

The advantages of this type system are that (1) it aligns well with types already used by major data pipeline tools (binary Avro and, with some transformation, Thrift and Protocol Buffers), (2) it is easy to represent as JSON or XML (the lack of circular references is particularly helpful), (3) any type is nullable, including primitives, (4) Avro's rules for schema resolution can be reinterpreted as type promotion for type inference (see Sec. 4.3), (5) all types have a strict ordering (see Avro sort order specification), and (6) the type schemae are JSON objects and strings, which fit seamlessly into a PFA document.

4.2 Type schemae in the PFA document

An Avro type schema can be a JSON object or a string. JSON objects construct parameterized types, while strings specify type primitives or reference previously defined types. Some PFA top-level fields and member values of special forms must be Avro schemae: these schema are simply included inline with the JSON representing the rest of the PFA document.

Below is a summary of the schema syntax that is relevant for PFA. All Avro schema elements must be accepted by a PFA reader, but these are the only ones that influence PFA.

The following strings are type primitives: “null”, “boolean”, “int”, “long”, “float”, “double”, “string”, “bytes”. Other strings are either previously defined named types or they are invalid. The form `{"type": "X"}` for string `X` is equivalent to the string on its own.

A byte array with name `N` and fixed length `L` is specified by the form `{"type": "fixed", "name": "N", "namespace": "NS", "size": L}`. The namespace is optional, but the name is not. The length `L` must be a JSON integer. Avro fixed types have additional object members, but they are not relevant for PFA.

An enumeration with name `N` and symbols `S` is specified by the form `{"type": "enum", "name": "N", "namespace": "NS", "symbols": S}`. The namespace is optional, but the name is not. The symbols `S` must be a JSON array of strings. Avro enumeration types have additional object members, but they are not relevant for PFA.

An array with elements of type `X` is specified by the form `{"type": "array", "items": X}`. An array does not accept a name or any other member values.

An map with values of type `X` is specified by the form `{"type": "map", "values": X}`. A map does not accept a name or any other member values.

A record with name `N` and fields `F` is specified by the form `{"type": "record", "name": "N", "namespace": "NS", "fields": F}`. The namespace is optional, but the name is not. The fields `F` must be JSON objects with the following form: `{"name": "FN", "type": "FT", "default": D, "order": O}` where `name` and `type` are required and `default` and `order` are not. The default `D` is encoded in the Avro-JSON format and provides a default value if the input data stream is missing one. The order `O` is one of these strings: “ascending”, “descending”, and “ignore”, and it defines the sort order for the record. Avro record types and field types have additional object members, but they are not relevant for PFA.

A union of types `T1` … `TN` is specified by a JSON array form `[T ... TN]`.

If the Avro implementation used by a PFA system supports aliases for schema resolution, the aliases should be used for type inference (see Sec. 4.3). Aliases only apply to named types and record fields.

Avro schema parsing is usually implemented as a stateful process, in which the parser remembers previously named types and recognizes its namespace-qualified name in a JSON string as representing the type. This is especially important for recursively defined types. A PFA document may have many type schemae embedded within it, often as member values of JSON objects. Systems that load JSON objects into hashtables cannot guarantee that the order of JSON object members is preserved, which could cause schemae to be read in any order.

Therefore, PFA implementations must pass Avro schemae to be parsed in an order that resolves dependencies or PFA implementations must parse the schemae themselves in an order that resolves dependencies. PFA documents must define named types (as a JSON object) exactly once and reference them (as a string) elsewhere.

4.3 Type inference

PFA uses a near-minimum of type annotations for static type analysis. Only the inputs to every calculation, which are function parameters, literal constants, inline arrays/maps/records, the symbols `input` and `tally`, and cell/pool definitions, and the outputs of every calculation, which are function return values and the scoring engine `output`, need to be specified. Unlike traditional languages (e.g. C or Java), the types of new variables are not specified: they are inferred through their initialization expressions (and would have been redundant if supplied).

With these annotations, the type check algorithm is simple. Every expression is a tree of subexpressions, whose leaves are either references to previously defined symbols, function parameters, cells, or pools (with known type) or constants (with specified type). Every function and special form has a type signature that may accept its arguments, in which case type-checking continues toward the root of the tree, or reject its arguments, in which case the PFA document fails with a semantic error. Every function and special form has a return type, which may depend on the types of its arguments (but not the values of its arguments, which are only known at runtime). Arguments and return types should be recursively checked until the root of the tree (the type of the expression as a whole) is reached. This derived type is checked against the declared function return type or `output`. If the declared type does not accept the derived type, the PFA document is rejected with a semantic error.

Some special forms take a JSON array of expressions and either apply no return type constraint or only constrain the last expression, which is used as a return value. Each case is explicitly specified in Sec. 7.

In passing, we note that the type annotations could have been more minimal if return types were not required, and some input types could, in principle, be inferred from their position in a function argument list. However, an explicit `output` allows a PFA system or casual observer to quickly determine if a scoring engine will fit into a given workflow, in which the input types and output types are constrained by data pipelines. Moreover, function return types cannot always be omitted, even in theory: recursive functions cannot determine their return type from parameters only, for instance. Also, inferring input types from parents or siblings in the expression tree unnecessarily complicates the type-check algorithm. The algorithm chosen for PFA is strictly local— only subexpressions and previously defined symbols are needed to infer an expression type— and uniform— the same rules apply regardless of the function's call graph.

4.4 Type resolution, promotion, and covariance

At each step in the type inference algorithm, the expected type or type pattern is checked against the actual or derived type. All types have a non-commutative, binary "accepts" relation, for which "A accepts B" means that B is an acceptable observed type for expected type A. For example, "**double** accepts **int**" because integers are a subset of double-precision floating point numbers, and any function that needs a **double** must be able to use an **int** instead.

Even though Avro is a serialization protocol, it defines a suite of type promotion rules for the sake of schema resolution. In Avro, these rules are used to determine if an old version of a schema is compatible with a new version of a schema: for instance, if the old schema defines a variable as an int and the new schema defines it as a double, the old serialized dataset is forward-compatible— it can be used in an application as though it had the new schema. PFA uses the same rules to promote data through an expression.

These rules are described in the Schema Resolution section of the Avro specification, but they are reviewed here with an emphasis on how the rules are used in PFA type inference.

Expected type	Accepts
null	Only **null** or a **union** of only **null** (union of exactly one type, which is not a useful union).
boolean	Only **boolean** or a **union** of only **boolean**.
int	Only **int** or a **union** of only **int**.
long	**int** or **long** or a **union** of any subset of {**int**, **long**}.
float	**int**, **long**, or **float** or a **union** of any subset of {**int**, **long**, **float**}.
double	**int**, **long**, **float** or **double** (all numeric types are promoted). Also accepts a **union** of any subset of {**int**, **long**, **float**, **double**}.

Expected type	**Accepts**
string	Only **string** or a **union** of only **string**.
bytes	Only **bytes** or a **union** of only **bytes**.
fixed(`L`, `N`, `NS`)	Only a **fixed** with the same length `L` and fully-qualified name given by `NS` and `N`. Also accepts a **union** of only this **fixed** type.
enum(`S`, `N`, `NS`)	An **enum** whose symbols `S` are a subset of the expected **enum**'s symbols with fully-qualified name given by `NS` and `N`. For example, if an **enum** with symbols "one", "two", and "three" is expected, it will accept an **enum** of the same name with symbols "two" and "one". Also accepts a **union** of only these **enum** types.
array(`X`)	An **array** with items `Y` for which `X` accepts `Y` (arrays are covariant). For example, an **array** of **double** accepts an **array** of **int**, but an an **array** of **int** does not accept an an **array** of **double**. Also accepts a **union** of only these **array** types.
map(`X`)	A **map** with values `Y` for which `X` accepts `Y` (maps are covariant). Also accepts a **union** of only these **map** types.
record(`F`, `N`, `NS`)	A **record** whose fields are a superset of fields `F` (in any order) with corresponding field types such that the expected field accepts the observed field (records are covariant). For example, an expected record with fields {"one": **double** and "two": **string**} accepts an observed record with fields {"one": **double**, "two": **string**, and "three": **bytes**}. It also accepts an observed record with fields {"one": **int** and "two": **string**}. The observed record must also have the same fully-qualified name given by `NS` and `N`. Also accepts a **union** of only these **record** types.
union(`T`)	Either a **union(`T′`)** such that for all `t′` in `T′`, there exists a `t` in `T` for which `t` accepts `t′`, or a single type `t″` such that there exists a `t` in `T` for which `t` accepts `t″`. For example, a **union** of {**string**, **bytes**, and **null**} accepts a **union** of {**string** and **bytes**}, and it also accepts a **string**. However, the reverse is not true: a narrow type or union cannot accept a wider union.

4.5 Narrowest supertype of a collection of types

Some circumstances (return type of a special form, solution of a generic type pattern) require a single type that accepts a given set of types. For example, an "if" conditional with both a "then" and an "else" clause returns a value that might have the type of the "then" clause or might have the type of the "else" clause. If both clauses have the same type `X`, then the return type of the "if" special form is `X`, but if "then" has type `Y` and "else" has type `Z`, the type of the "if" special form is something that could be (accepts) `Y` or `Z`. In these situations, the resultant type is the narrowest supertype of the possibilities.

The following rules define the narrowest supertype of a collection of at least one type. In cases where more than one rule matches, the first matching rule is applied.

#	**Collection of types**	**Narrowest supertype**
1.	all **null**	**null**
2.	all **boolean**	**boolean**
3.	all **int**	**int**
4.	all **int** or **long**	**long**
5.	all **int**, **long**, or **float**	**float**

#	Collection of types	Narrowest supertype
6.	all **int**, **long**, **float**, or **double**	**double**
7.	all **string**	**string**
8.	all **bytes**	**bytes**
9.	all **fixed** with the same length and fully-qualified name	that **fixed** type
10.	all **enum** types with the same symbols and fully-qualified name	that **enum** type
11.	all **arrays**	an **array** of the narrowest supertype of the items of each **array**
12.	all **maps**	a **map** of the narrowest supertype of the values of each **map**
13.	all **records** with the same fields and fully-qualified name	that **record** type
14.	any other collection of types, excluding those containing **fixed** and **enum**	a union of those types, merging any unions contained in the collection (e.g. **union(**`X, Y`**)** and **union(**`Y, Z`**)** combine into **union(**`X, Y, Z`**)**) and combining any types that can be combined with the rules above (e.g. **int** and **double** become **double**, rather than **union(int, double)**).
15.	any other case	is a type error.

In summary, any cases that cannot be promoted to the same type are combined into a union except for collections containing **fixed** or **enum**. These cases, had they been allowed, would introduce the need for PFA implementations to perform runtime type conversions: values of different **fixed** types would need to be converted into raw **bytes** and values of different **enum** types would need to be converted into **string** or an **enum** with a superset of symbols. These new, anonymous types would have potentially unexpected properties: the broadening of **enums** cannot maintain the order of a collection of symbols, which are used in some statistical applications as a finite ordinal set. It is better to raise a type error and force the PFA author to explicitly convert these types to **bytes**, **string**, or explicitly define an **enum** with a superset of symbols.

Distinct **records** are combined into a union of those **records**, however (rule 13 falls through to rule 14 when the **records** are not exactly the same).

4.6 Generic library function signatures

User-defined functions in the `fcns` top-level field have parameter lists and return types specified strictly by Avro type schemae (see Sec. 6). However, some library functions have more general type signatures so that they can be more broadly applied.

Library functions are never declared in a PFA document, and thus they are not bound to the same restrictions. It would be possible to define a JSON format for expressing generic function signatures, but that format would not be an Avro schema. Moreover, PFA documents are not intended for generic programming, but for auto-generated code. Since every PFA document is a specific solution to a specific problem, it should be written in terms of ungeneric functions. (The generality can be in the routine that generates the PFA document.) The PFA library functions, however, are intended for a wide variety of problems, and thus should be generic.

Library function type patterns are a superset of Avro types. They include the same primitives:

- null, boolean, int, long, float, double, string, bytes

and the same parameterized, product, and sum types, though names are optional:

- fixed (*size:* `L`)
- fixed (*size:* `L`, *name:* `N`)
- enum (*symbols:* `S`)
- enum (*symbols:* `S`, *name:* `N`)
- array of **`X`**
- map of **`X`**
- record (*fields:* {**`name`**$_1$**`:`** *type*$_1$*,* **`name`**$_2$**`:`** *type*$_2$*, …* **`name`**$_n$**`:`** *type*$_n$})
- record (*fields:* {**`name`**$_1$**`:`** *type*$_1$*,* **`name`**$_2$**`:`** *type*$_2$*, …* **`name`**$_n$**`:`** *type*$_n$}, *name:* `N`)
- union of {`T`}

When present, names are fully-qualified, rather than being split into namespace-name pairs. A type pattern of fixed, enum, or record without a name matches any fixed, enum, or record with the specified structure (structural typing, rather than nominative). For instance, a library function could require a record with integer, double, and string fields named "one", "two", and "three" like this:

record (*fields:* {**`one:`** int, **`two:`** double, **`three:`** string})

Since this pattern has no *name,* it would match any record that has exactly these fields with these types.

Unlike Avro types, the parameters of type patterns can be specified by wildcards. Wildcards are labeled and may be restricted to a set of Avro types (not patterns):

- any **`A`**
- any **`A`** of {*type*$_1$*, type*$_2$*, … type*$_n$}

Wildcards can appear anywhere that a pattern is expected. For instance,

array of any **`A`**

is an array with unspecified item type.

Wildcards with repeated labels constrain two types to be the same type. For instance,

record (*fields:* {**`one:`** any **`A`**, **`two:`** any **`B`**, **`three:`** any **`B`**})

is a record with three fields of unspecified type, though fields "two" and "three" have the same. If they are not exactly the same, they are promoted to the narrowest supertype of the matches. The following are examples of Avro types that would match this pattern.

Example 4.1. `R1` matches because "two" and "three" are both **string**.

```
{"type": "record", "name": "R1", "fields": [
    {"name": "one", "type": "int"},
    {"name": "two", "type": "string"},
    {"name": "three", "type": "string"}]}
```

Example 4.2. `R2` matches because "two" and "three" are both **int**. The type that matches wildcard `B` does not need to be different from the type that matches wildcard `A` (also "int").

```
{"type": "record", "name": "R2", "fields": [
    {"name": "one", "type": "int"},
    {"name": "two", "type": "int"},
    {"name": "three", "type": "int"}]}
```

Example 4.3. `R3` matches because "two" and "three" can both be promoted to **double**.

```
{"type": "record", "name": "R3", "fields": [
    {"name": "one", "type": "string"},
    {"name": "two", "type": "int"},
    {"name": "three", "type": "double"}]}
```

The scope of wildcard labels is the entire function signature, including all parameters and return type.

Example 4.4. As an example, the signature of "+" (the library function that adds two numbers) is

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

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

The two parameters, `x` and `y`, can be any type in the set {int, long, float, double} and the return type is the narrowest supertype of `x` and `y`. Thus, int + int → int, int + double → double, etc. The restriction on `A` in the pattern for `x` applies equally to `y` because all constraints must be satisfied for a pattern to match.

Example 4.5. Shared labels are primarily used to carry types from a parameter to the return type. They allow a function like "a.subseq" (extract a subsequence of an array) to be defined once for any type of items.

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

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

Record substructure can also be matched with wildcards. Such a pattern has one of these two forms:

- any record `A`
- any record `A` with {$\texttt{name}_1$: $type_1$, $\texttt{name}_2$: $type_2$, … $\texttt{name}_n$: $type_n$}

The first matches any record, regardless of what fields it contains, and the second matches a record with at least the specified fields. Wildcards with and without specifying record substructure have labels in the same namespace.

Example 4.6. For example, the function "stat.sample.mean" computes a mean from a record containing components of the running sum. Its signature requires that the record has double-valued fields `sum_w` and `sum_wx`, but it does not specify anything else about the record. This `runningSum` could be overloaded with additional, application-specific functionality.

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

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

Example 4.7. As another example, the "model.tree.simpleWalk" function for evaluating decision trees requires that **datum** is a record and that **treeNode** is a record with **field**, **operator**, **value**, **pass**, and **fail** fields. Since it does not specify anything else about the record, the **treeNodes** could also carry metadata describing how the tree was made.

{"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**

Finally, some library functions can accept functions as arguments, even though functions are not first-class objects in the language. A function cannot be assigned to a symbol, but it can appear in an argument list. The following type pattern matches functions:

- function ($type_1$, $type_2$, … $type_n$) → *type*

The pattern does not specify the names of parameters, only their number, order, and types (as patterns). Only non-generic functions can be matched, and since most library functions are generic, the matched function would usually be a user-defined function. (Notable exceptions are mathematical special functions, probability distributions, and clustering metrics.)

Example 4.8. The parameter and return types of a function pattern can be wildcarded, as in this example of "a.filter", which filters an array.

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

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

Example 4.9. Library functions with function arguments are primarily used to override the default behavior of a statistical routine through callbacks. The "model.tree.predicateWalk" function illustrates how a decision tree traversal can be given an arbitrary or even user-defined predicate.

{"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**

Function arguments can only be provided using the fcndef or fcnref special forms, which require explicit functions to be known during static analysis. Therefore, the platform on which PFA is being implemented does not need first class functions or the equivalent (e.g. objects with a pre-specified "apply" method). It does not even need functions in the normal sense: if the platform requires all functions to be expanded inline, the PFA system can generate specific code for each case. That is, the “a.filter” and “model.tree.predicateWalk” functions can be expanded into special-case bytecode for each call that takes a different function argument, possibly mixing the library function implementation with the user-defined callback. This would not be possible if function arguments could only be resolved at runtime.

5 Symbols, scope, and data structures

Calculations in PFA are performed by nesting function calls in argument lists (purely functional programming) and assigning values to symbols, cells, and pools, possibly overwriting previous values (imperative programming). Some mathematical algorithms are more easily expressed in a functional style while others are more easily expressed in an imperative style. Cells and pools (global state) are discussed in Sec. 3.4. Symbols provide temporary local state.

Much like a cell, a symbol is a named container with fixed type whose value can be replaced: it is a variable. Unlike a cell, a symbol can only be referenced within a limited scope. Symbol scopes in PFA are

- lexical: scopes are defined in terms of character ranges in the text of the PFA document, and
- block-level: they range from the declaration of the symbol to the end of the containing block, which is either a single expression of a JSON array of expressions.

Since blocks are deeply nested in a moderately complex PFA document, symbol scopes may be deeply nested as well. A symbol must not be shadowed; that is, it must not be declared twice in the same scope or in two scopes such that one is nested within the other. The same symbol name may be used in non-overlapping scopes.

Symbols are declared with the "let" special form and reassigned with the "set" special form (see Sec. 7.6). This distinction between declaration and reassignment should be enforced, as it allows an observer to determine which symbols are constant and which algorithms are purely functional at a glance.

Although a symbol is readable anywhere below the point at which it is defined, some special forms limit the scopes in which a symbol can be reassigned. For example, symbols declared outside of an anonymous function cannot be modified by the function, since that would be hard to implement in a system that does not have closures. Looping special forms for which **`seq`** is **`false`** may be evaluated in any order to allow for systems that might parallelize the loop— in this case, symbols declared outside the loop cannot be modified in the loop, since that would lead to race conditions. Special forms that prohibit modification of symbols declared outside the form are said to be "sealed from above".

Some forms also do not allow new variable declarations. One such example is the argument list of a function call, since such a declaration could never be used (each argument of the function call is in a different scope). These scopes are said to be "sealed within". In a functional programming style, it is sometimes desirable to insert whole algorithms in what would otherwise be a restricted block like a function argument. To allow for this usage, a "do" special form can be placed within such a scope, creating a sub-scope that allows variable declarations and expands a single-expression block into a JSON array of expressions. The "do" form is required to ensure that this usage is intentional.

The sealed-from-above or sealed-within status of scopes within each special form is specified in Sec. 7.

Symbol names must match the following pattern: **`[A-Za-z_][A-Za-z0-9_]*`** (the first character must be a letter or underscore; subsequent characters, if they exist, may also be numbers). Symbol names do not share a namespace with cells, pools, or functions.

5.1 Immutable data structures

All values in PFA, including complex data structures like arrays, maps, and records, are immutable. That is, there are no special forms or library functions that can change the structure of a value in-place, so in-place operations cannot be expressed in any routine constructed from those calls. The PFA specification puts no constraints on how values are implemented, however: they may be immutable in the context of PFA operations and yet mutable in the context of the host system. We suggest that complex data structures be implemented with structural sharing to optimize speed and memory usage, but this is an implementation detail.

One consequence of all data structures being immutable is that there is no distinction between a value and a reference to a value or between copying and linking. In languages with mutable values and references, this distinction is important because linking a reference to the same value in multiple places creates an unseen connection between them. For instance (in Python),

```
y.child = x
z.child = x
...
y.child.modify()
```

modifies `z.child` as well as `y.child`, while

```
y.child = deepcopy(x)
z.child = deepcopy(x)
...
y.child.modify()
```

only modifies `y.child`. Since PFA lacks the equivalent of `modify()`, the above distinction is irrelevant.

Another consequence is that it is impossible to create a circular reference in PFA. In Python, one can create a circular reference in a list `x` by

```
x.append(x)
```

Attempts to depth-first walk through this list of lists would result in an infinite loop, and modifications of `x` would silently change this nested element (as above). In PFA, the equivalent statement

```
{"set": {"x": {"a.append": ["x", "x"]}}}
```

creates a new array that contains all of the old items of `x` with the entire old array `x` appended. This new array is assigned to symbol `x`. Since PFA has no mutable structures (and no lazy evaluation), it is impossible to create a circular reference.

Example 5.1. In fact, assignments with the same symbol on the left-hand side as the right-hand side behave like "`x = x + 1`" in that the right-hand side always deals with the old value of the symbol and the left-hand side may change the meaning of the symbol to a new value. For instance,

```
{"do": [
    {"let": {"x": {
        "type": {"type": "record", "name": "R", "fields": [
            {"name": "child", ["R", "null"]}]},
        "value": {"child": null}}}},
    {"set": {"x": {"attr": "x", "path": ["child"], "to": "x"}}},
    {"set": {"x": {"attr": "x", "path": ["child"], "to": "x"}}},
    {"set": {"x": {"attr": "x", "path": ["child"], "to": "x"}}},
    "x"]}
```

results in

```
{"child": {"child": {"child": {"child": null}}}}
```

just as

```
{"do": [
    {"let": {"x": 0}},
    {"set": {"x": {"+": ["x", 1]}}},
    {"set": {"x": {"+": ["x", 1]}}},
    {"set": {"x": {"+": ["x", 1]}}},
    "x"]}
```

results in 3. In languages with mutable data structures, some updates behave like "`x = x + 1`" while others create circular references.

Similarly, there is no distinction between passing a function argument by reference versus passing it by value. Such a distinction would be seen when the function modifies the object that it is given, but in PFA, these modifications are not possible.

To get this behavior in a language or context that allows in-place modifications, one must make deep copies of the objects because copies are not linked the way that references are. However, purely immutable objects can be safely passed without copying and structurally shared when modified. In the examples above, a PFA implementation may use the same object in memory wherever its value is needed and may always pass a pointer to the object as a function argument. Since values and references are equivalent, one can choose the computationally least expensive operation.

Our decision to make all data structures immutable was driven by two needs: (1) to make them fully expressible by Avro type schemae, which do not allow for circular references or any non-treelike graph, and (2) to simplify the read-copy-update concurrency algorithm. If objects were mutable, then an explicit copy would be needed to allow read operations to see the old version of the object while write operations create a new version. In addition to making implementations more complicated, this would make all write operations more expensive, since they must do a deep copy on every update. With immutable objects, the copy step is unnecessary.

5.2 Memory management

PFA does not define any particular memory management technique. The language has constructs for creating objects but not for deleting them, so some sort of garbage collector will be needed to release objects that are out of scope. This implicit garbage collector may be the same as the one used by PFA host's environment (e.g. the JVM and Python have built-in garbage collectors) or it may be a library used to collect garbage only in the PFA system (e.g. the boehmgc and boost libraries provide garbage collectors for C++).

6 User-defined functions

A PFA document may define new functions that are called as though they were library functions. There are two differences between these "user-defined" functions and the library functions specified by PFA:

- user-defined functions are prefixed by "`u.`" when called (in the "u" branch of the module tree);
- their signatures must consist of specific Avro types, since only library functions can have generic signatures (see Sec. 4.6).

The scope of all functions, including user-defined functions, is global. Functions are also static; they cannot be declared at runtime. They may be declared in the **`fcns`** top-level field, which makes their global, static nature manifest, or they may be declared inline in the argument list of a special form or library function that accepts a function as an argument.

In the latter case, the user-defined function has no name (an "anonymous function") and some features of a lexical closure. It has access to symbols defined in an enclosing scope (it "closes over" those symbols), but this access is limited to reading only. A PFA system may implement this by internally adding the closed symbols to its parameter list. In all other aspects, an anonymous function behaves like a globally defined function and may be implemented as one (with an auto-generated name).

Function names must match the following: **`[A-Za-z_]([A-Za-z0-9_]|\.[A-Za-z][A-Za-z0-9_]*)*`** (consists of dot-delimited words in which each word starts with a letter or underscore; subsequent characters, if they exist, may also be numbers; there must be at least one word). Functions do not share a namespace with cells, pools, or symbols.

6.1 No first-class functions

Some languages have first-class functions, meaning that functions are "first-class citizens" and can be treated like any other data structure. In PFA, functions are second-class, but not coach. Similar to first-class functions, they may be passed as arguments to other functions (specific library functions that accept functions as arguments) and they may be defined inline as anonymous functions that close over local variables. Unlike first-class functions, they cannot be assigned to symbols or be returned from functions, and the closures have read-only access to the variables they close over.

These restrictions make it possible to implement PFA functions in very limited environments and to be able to fully analyze the function call graph without executing the scoring engine (thereby statically detecting recursion, if necessary). If, instead, a function could be assigned to a symbol, then it could be changed at runtime and not be predicted statically. If an inline function could reassign symbols in its scope, then it could not be internally implemented as a simple function.

These rules provide enough flexibility to implement callbacks in library functions, but not so much as to make the call graph unpredictable or make the scoring engine impossible to implement in environments that only accept inline-expanded functions. Pascal and Algol are two historical examples of languages with the same set of rules.

6.2 Syntax for declaring new functions

User functions are declared with the "fcndef" special form. If "fcndef" appears in an argument list, it defines an anonymous function. If it appears as a member value for a name-value pair in the **`fcns`** top-level field, it defines a named function.

6.2.1 Defining function: the "fcndef" special form

The "fcndef" special form has the following syntax.

```
{"params": [{par1: type1}, {par2: type2}, ...], "ret": retType, "do": expr}
```

`par1`	string, name of first parameter
`type1`	Avro schema, type of first parameter
`par2`	string, name of second parameter
`type2`	Avro schema, type of second parameter
. . .	
`retType`	Avro schema, return type
`expr`	expression or JSON array of expressions

The **params**, **ret**, and **do** fields must be present, though **params** could be an empty JSON array (no parameters). The individual parameters are always represented by single-member JSON objects: the name is the parameter name (used in the function body), and the value is its type.

The **do** field is the function body, and its return value is the last or only expression it contains. The **do** field must contain at least one expression (it must not be an empty JSON array).

If this form appears in the **fcns** top-level field, the function name (without its namespace qualifier, "**u.**") is given by the associated member name.

Example 6.1. For example,

```
"fcns": {
    "square": {"params": [{"x": "double"}], "ret": "double", "do": {"**": ["x", 2]}},
    "cube": {"params": [{"x": "double"}], "ret": "double", "do": {"**": ["x", 3]}}
}
```

defines **u.square** (which squares the input value) and **u.cube** (which cubes the input value). The number 5 may now be squared by **{"u.square": 5}** or **{"u.square": [5]}**.

Example 6.2. This Fibonacci number algorithm demonstrates recursion.

```
"fcns": {
    "fib": {"params": [{"n": "int"}], "ret": "int", "do":
        {"cond": {"if": {"==": ["n", 0]}, "then": 0},
                 {"if": {"==": ["n", 1]}, "then": 1},
         "else": {"+": [
             {"u.fib": [{"-": ["n", 1]}]},
             {"u.fib": [{"-": ["n", 2]}]}
         ]}}}
}
```

Example 6.3. This iterative Fibonacci number algorithm illustrates a multi-line **do** field.

```
"fcns": {
    "fib": {"params": [{"n": "int"}], "ret": "int", "do": [
        {"let": {"now": 0,
                 "next": 1}},
        {"for": {"i": "n"},
         "until": {"<": ["i", 0]},
         "step": {"i": {"-": ["i", 1]}},
         "seq": true,
         "do": [
             {"let": {"tmp": {"+": ["now", "next"]}}},
             {"set": {"now": "next",
                      "next": "tmp"}}
         ]},
         {"if": {"==": ["n", 0]},
          "then": 0,
          "else": "next"}
    ]}
}
```

6.3 Syntax for referencing functions

Named functions are referenced with the "fcnref" special form, and anonymous functions are both declared and referenced with an inline "fcndef" special form (see above).

6.3.1 Referencing a globally defined function: the "fcnref" special form

The "fcnref" special form has the following syntax.

```
{"fcnref": name}
```

`name`	string, the name of the function

The **`name`** string must be an exact name of the function, not an expression that can be evaluated at runtime.

Example 6.4. Here is a complete PFA document that increments and returns a counter every time it is pinged (the scoring engine is executed with `null` input).

```
{"input": "null",
 "output": "int",
 "action": [
     {"cell": "counter", "to": {"fcnref": "u.increment"}},
     {"cell": "counter"}],
 "cells":
     {"counter": {"type": "int", "init": 0, "shared": true}},
 "fcns":
     {"increment":
         {"params": [{"x": "int"}],
          "ret": "int",
          "do": {"+": ["x", 1]}}}
}
```

The increment is defined in a function so that getting the counter value, incrementing it, and putting the new value in the cell are all one atomic action— other update attempts block until this one is done. Without ensuring atomicity, the counter could miss an update when a second update attempt reads `counter` before the first writes its new value.

Example 6.5. The same operation could be performed with an inline “fcndef”. In the example below, the scoring engine adds the input value to `counter` every time it is called. Because of its location, the anonymous function can close over the `input` symbol, which exists in the `action` block but not in `fcns`.

```
{"input": "int",
 "output": "int",
 "action": [
     {"cell": "counter", "to": {"params": [{"x": "int"}],
                                "ret": "int",
                                "do": {"+": ["x", "input"]}}},
     {"cell": "counter"}],
 "cells":
     {"counter": {"type": "int", "init": 0, "shared": true}}
}
```

7 Expressions

PFA documents may contain simple programs formed by composing expressions. There are four types of expressions: symbol references, literal values, function calls, and special forms.

7.1 Symbol references

A symbol reference yields the current value of a predefined symbol. For example, the **`action`** routine has a **`input`** symbol pre-defined: wherever this appears, the input value is inserted. New symbols can be created with "let" and old symbols can be changed with "set", if the current context allows it (see Sec. 5). Cells and pools are not symbols; they are referenced through a pair of special forms.

A symbol reference is simply a JSON string. JSON strings in contexts where an expression is expected are interpreted as symbol references (with one exception); JSON strings in other contexts have other meanings. The exception is the following: a valid symbol name must not contain dots (".") as described at the end of Sec. 5, and if a JSON string where an expression is expected contains dots, it should be interpreted as a shortcut for the "attr" special form.

7.2 Literal values

Literal values are constants embedded in an expression. Simple examples are numbers, such as the "2" in this expression that squares **`x`**: **`{"**": ["x", 2]}`**.

Cells without "cell-to" can also be used to define constants in a PFA document. The differences are: (1) a literal appears in the midst of an expression, while cells are in the **`cells`** top-level field, (2) cells are named and can be referenced in many places throughout a PFA document, including functions, but literals are unnamed unless assigned to a symbol of limited scope, (3) constant cells are constructed once, in the initialization phase of the engine, but literals may or may not be constructed every time the program flow reaches them (this is an implementation detail). Typically, large data structures like the representation of a statistical model would be stored in a cell, rather than an inline literal.

Literals are represented by the following special forms:

- **`{"int": NUMBER}`** where **`NUMBER`** is a JSON integer. The return type is **int**; a value that is too large to be represented as a 32-bit number is a syntax error.
- **`{"long": NUMBER}`** where **`NUMBER`** is a JSON integer. The return type is **long**; a value that is too large to be represented as a 64-bit number is a syntax error.
- **`{"float": NUMBER}`** where **`NUMBER`** is a JSON floating-point number. The return type is **float**; a value that is too large, too small, or too precise to be represented as a IEEE 754 32-bit floating point number is a syntax error.
- **`{"double": NUMBER}`** where **`NUMBER`** is a JSON floating-point number. The return type is **double**; a value that is too large, too small, or too precise to be represented as a IEEE 754 64-bit floating point number is a syntax error.
- **`{"string": STRING}`** where **`STRING`** is a quoted string. The return type is **string**.
- **`{"base64": STRING}`** where **`STRING`** is a base-64 representation of a binary input. The return type is **bytes**.
- **`{"type": TYPE, "value": VALUE}`** where **`TYPE`** is an Avro schema and **`VALUE`** is a JSON literal (*not* an expression) whose type matches **`TYPE`**. This form can be used to construct arrays, maps, records, etc.

Additionally, if an integer appears where an expression is expected, that number is an **int** literal if 32-bit, a **long** literal if 64-bit, and a syntax error if larger. If a floating point number appears where an expression is expected, that number is a **double** literal (not a **float**, even if small enough). A floating-point number that is too large, too small, or too precise to be a IEEE 754 64-bit floating point number causes a syntax error.

There is also a shortcut for making **string** literals: `[STRING]` (single-element JSON array containing a string). Note that a string, by itself is interpreted as a symbol reference (because symbol references are more common than string literals). In some contexts, such as the argument list of a function, an expression or an array of expressions is expected. If a `[STRING]` appears in one of these contexts, it is interpreted as a one-element array containing a symbol reference, not a literal string.

7.3 Function calls

Most of the functionality of PFA is provided through function calls; data-centric scoring engines, such as statistical models, would often involve only one function call. Library functions are part of the PFA definition (see Sec. 8 et seq.); user functions are defined in a PFA document (see Sec. 6).

A function call is expressed in JSON as a single-member object, like this

```
{"functionName": [argument1, argument2, ... argumentN]}
```

for a function `functionName` of `N` (zero or more) arguments or

```
{"functionName": argument1}
```

for a function of exactly one argument.

The arguments of a function call can either be an expression or a function reference (assuming that the called function accepts functions as arguments).

Functions have strict requirements on the number of arguments (always fixed; there are no optional arguments or varargs) and their types. Functions return a value with a specific type, though that type may depend on the argument types (for library functions).

Function call expressions evaluate all of the function's arguments, from left to right, before calling the function, unless documented otherwise. For instance, the `and` function only evaluates its second argument if the first does not evaluate to `false` and the `or` function only evaluates its second argument if the first does not evaluate to `true`.

The expression in each argument of a function call is evaluated in a separate, sealed-from-above and sealed-within scope (see Sec. 5). Thus, one cannot declare or reassign symbols as one progresses through the argument list.

7.4 Special forms

Special forms, are expressions that take arguments, perform an operation, and return a value, much like function calls. Unlike function calls, however, they may have irregular syntax. Whereas a function call is always a JSON object with one member name (the function name) and one member value (its arguments, interpreted as expressions), some special forms have multiple JSON object members and some special forms have custom interpretations for their member values. These object members may appear in any order. In addition, special forms have custom scoping rules when they do interpret values as expressions.

7.5 Creating arrays, maps, and records

7.5.1 Creating arrays/maps/records from expressions: the "new" special form

Arrays, maps, and records can be created with the `{"type": TYPE, "value": VALUE}` literal form described above, but the `VALUE` must be a JSON literal and not an expression. Thus, the literal form cannot depend on any inputs. The "new" special form exists to create arrays, maps, and records from expressions:

```
{"new": ARRAY, "type": ARRAY-TYPE}
{"new": OBJECT, "type": MAP-OR-RECORD-TYPE}
```

where `ARRAY` is a JSON array of expressions whose type is accepted by Avro schema `ARRAY-TYPE`, and `OBJECT` is a JSON object whose member values are expressions matching Avro schema `MAP-OR-RECORD-TYPE`.

Example 7.1. The following returns an array of powers of `x`, which has type **double**.

```
{"new": [1, "x", {"**": ["x", 2]}], "type": {"type": "array", "items": "double"}}
```

Example 7.2. The following returns a map from names to powers of `x`, which has type **double**.

```
{"new": {"p0": 1, "p1": "x", "p2": {"**": ["x", 2]}},
 "type": {"type": "array", "items": "double"}}
```

Example 7.3. The following initializes a record of type `R` with a **string** stored in `x` and its length.

```
{"new": {"theString": "x", "theLength": {"s.len": "x"}},
 "type": {"type": "record", "name": "R", "fields":
     [{"name": "theString", "type": "string"},
      {"name": "theLength", "type": "int"}]}}
```

7.6 Symbol assignment and reassignment

7.6.1 Creating symbols: the "let" special form

The "let" special form creates new symbols and assigns an initial value. The "set" special form changes the value associated with a set of symbols. There are no type declarations for new symbols because its types are inferred from the initial value.

A "let" has the following syntax:

```
{"let": {"name1": VALUE1, "name2": VALUE2, ...}}
```

where `VALUE1`, `VALUE2`, etc. are expressions. These expressions must not depend on symbols defined in the same "let" form, and they may be evaluated in any order. (The symbols can be referenced after the end of the "let" form and before the end of its containing form.) The `VALUE` expressions are sealed-within: they may not declare new symbols unless wrapped in a "do" special form, and cannot change externally declared symbols.

A "let" that only declares one symbol still requires a nested JSON object. The "let" form has return type **null**. It must declare at least one symbol.

7.6.2 Changing symbol bindings: the "set" special form

A "set" has the following syntax:

```
{"set": {"name1": VALUE1, "name2": VALUE2, ...}}
```

where `VALUE1`, `VALUE2`, etc. are expressions. These expressions may depend on symbols that are rebound in the same "set" form, but if so, they are provided with the old values of those symbols, as defined immediately before the "set" form. The assignments within a "set" may be evaluated in any order. The `VALUE` expressions are sealed-within, just like the "let" form.

A "set" that only changes one symbol still requires a nested JSON object. The "set" form has return type **null**. It must reassign at least one symbol.

Example 7.4. If the following appears in a JSON array of expressions, the final values of `x` and `y` are 2 and 2 (not 2 and 3 or 3 and 2).

```
{"let": {"x": 1, "y": 1}},
{"set": {"x": {"+": ["x", "y"]}, "y": {"+": ["x", "y"]}}}
```

Regardless of whether `x` is reassigned first or `y` is reassigned first, each sum sees both `x` and `y` as having a value of 1 at the time of assignment.

7.7 Extracting from and updating arrays, maps, and records

7.7.1 Retrieving nested values: the "attr" special form

The "attr" special form extracts a value from an array, a map, a record, or any combination of these three. The "attr-to" special form returns an object with one element changed (leaving the original untouched).

The form of "attr" is

```
{"attr": EXPRESSION, "path": INDEXES}
```

where `EXPRESSION` is the array, map, or record to extract from and `INDEXES` is a JSON array of expressions with **int** type or **string** type and string literals. These `INDEXES` specify a path through the nested objects. If nesting is only one level deep, `INDEXES` must have exactly one element. It is a syntax error for `INDEXES` to have zero elements.

If the type at a particular nesting level is an **array**, the path index must resolve to an **int**, though it can perform arbitrary calculations to produce the **int**. If the type at a particular nesting level is a **map**, the path index must resolve to a **string**. If the type at a particular nesting level is a **record**, the path index must be a literal **string** (not just an expression that resolves to a **string**), and that string must be a field name of the record. Since it is a literal string, static analysis can verify that the field name exists and a PFA system should raise a type error if no such field exists. The nesting level is increased for each array element of the path, and the return type of "attr" is the type of the deepest object referenced by the path.

As a convenience, "attr" forms may also be expressed as a dot-delimited string resembling a symbol reference. The first substring is taken to be a symbol reference `EXPRESSION` and the rest are taken to be elements of the `INDEXES` array (literal expressions only). PFA systems should implement this short-cut.

Example 7.5. The following are equivalent:

```
"x.4.key.field"
```

and

```
{"attr": "x", "path": [4, ["key"], ["field"]]}
```

and

```
{"attr": {"attr": {"attr": "x", "path": [4]}, "path": [["key"]]}, "path": [["field"]]}
```

If the type of `x` is

```
{"type": "array", "items": {"type": "map", "values": {"type": "record", "name": "R",
     "fields": [{"name": "field", "type": X}]}}}
```

then the example is valid and the return type is `X`.

(Note: `["key"]` and `["field"]` are short-cuts for `{"string": "key"}` and `{"string": "field"}`, respectively; see Sec. 7.2.)

7.7.2 Copy with different nested values: the "attr-to" special form

The form of "attr-to" is:

```
{"attr": EXPRESSION, "path": INDEXES, "to": VALUE-OR-FUNCTION}
```

with the same meaning for `EXPRESSION` and `INDEXES` as in "attr". If the attribute evaluates to `X` at the end of its path, the `VALUE-OR-FUNCTION` argument must either be an expression of type `X` or a function that maps `X` to `X`. If is an expression, then the form will return a structure like the original, but with the specified subelement changed to the new value. If `VALUE-OR-FUNCTION` is a function, then the function is evaluated, passing in the old value of the subelement and updating the structure with the function's return value.

The `INDEXES` must contain at least item. There is no equivalent of using the dot-delimited string as a short-cut. The "attr-to" form returns a whole new structure with one subelement changed; it has the same type as `EXPRESSION`. It must be emphasized that "attr-to" does not change the `EXPRESSION` or the object it refers to, nor does it reassign any symbols; it returns a new value.

Example 7.6. If `x` is an array, we can effectively change one element of `x` to `xn` by

```
{"set": {"x": {"attr": "x", "path": ["n"], "to": "xn"}}}
```

Example 7.7. If the `"x.4.key.field"` subelement is a `int`, following are equivalent:

```
{"attr": "x", "path": [4, ["key"], ["field"]], "to": {"+": ["x.4.key.field", 1]}}
```

and

```
{"attr": "x", "path": [4, ["key"], ["field"]], "to":
     {"params": [{"z": "int"}], "ret": "int", "do": {"+": ["z", 1]}}}
```

The first version represents the same path twice (the second time with dot-notation for brevity), while the second version represents the path once and applies an updator function when it gets to the end of the path. This updator function could be referenced by name with fcnref for code re-use.

7.8 Extracting from and updating cells and pools

Cells and pools (see Sec. 2.1) are referenced with "cell" and "pool" special forms and updated with "cell-to" and "pool-to" special forms. Subelements in a cell or pool are referenced/updated with a path of the same form as "attr", though the "cell" and "cell-to" forms may be given without any path to specify the cell as a whole.

7.8.1 Retrieving cell values: the “cell” special form

The “cell” special form has the following syntax.

```
{"cell": NAME}
```

or

```
{"cell": NAME, "path": INDEXES}
```

where `NAME` is the name of the cell and `INDEXES` is a JSON array of **int** and **string** valued expressions (for arrays and maps) and **string** literals (for records). If a `path` is not given, this form returns the value of the specified cell. If a `path` is given, then it walks down the `INDEXES` of the `path` in the same way as “attr”. Unlike “attr”, `INDEXES` may be an empty JSON array, in which case the behavior is the same as if `path` had not been specified.

7.8.2 Changing cell values: the “cell-to” special form

The “cell-to” special form has the following syntax.

```
{"cell": NAME, "to": VALUE-OR-FUNCTION}
```

or

```
{"cell": NAME, "path": INDEXES, "to": VALUE-OR-FUNCTION}
```

where `VALUE-OR-FUNCTION` is expression of type `X` or a function that maps `X` to `X`, assuming that the cell or subelement has type `X`. Like “attr-to”, the `to` field replaces the cell or subelement at the end of the path with `VALUE-OR-FUNCTION` if it is a value or evaluates `VALUE-OR-FUNCTION` on the subelement if it is a function. Unlike “attr-to”, the “cell-to” form changes the value of the cell and returns `null`. Any references to the old value of the cell (or its parts) are unchanged.

Example 7.8. For example, suppose that `myCell` is a cell containing an array of **string** and `n` is an index within the bounds of the array.

```
{"let": {"x": {"cell": "myCell"}, "y": {"cell": "myCell", "path": ["n"]}}},
{"cell": "myCell", "path": ["n"], "to": {"string": "hello"}}
```

The first line retrieves the value of the cell and a subelement of the cell, putting them in local symbols `x` and `y`. The second line changes `myCell` in such a way that element `n` (only) is now `"hello"`. However, both `x` and `y` are unchanged— they continue to reference the old value of `myCell` and its components.

For “attr-to”, the difference between the value form and the function form of `VALUE-OR-FUNCTION` is just a matter of style (whether the path is repeated or the function can be re-used). For shared cells, choosing between the value form and the function form of `VALUE-OR-FUNCTION` in “cell-to” could make a difference in the behavior of the scoring engine. If a cell is extracted and replaced in two steps, then it is possible for another scoring engine sharing that value to modify it between the extraction step and the replacement step. The function form, however, is atomic: all other attempts to modify the value wait until it is done.

Example 7.9. Suppose that `myCell` is a shared cell that we want to increment by 1. If we use

```
{"cell": "myCell", "to": {"+": [{"cell": "myCell"}, 1]}}
```

in multiple scoring engines, then it is possible that their calls to “cell” and “cell-to” might interleave. In this sequence of operations:

1. value of **myCell** is 5
2. scoring engine A gets the value of **myCell** (5)
3. scoring engine B gets the value of **myCell** (5)
4. scoring engine A adds 1 to 5 and sets the value of **myCell** to the result (6)
5. scoring engine B adds 1 to 5 and sets the value of **myCell** to the result (6)

myCell is only incremented once, though two scoring engines attempted to increment it. If steps 3 and 4 were reversed (a race condition), the final result would be 7, rather than 6.

Instead, we should increment a cell with the function form:

```
{"cell": "myCell", "to":
     {"params": [{"x": "int"}], "ret": "int", "do": {"+": ["x", 1]}}}
```

The same applies to subelements of a cell specified by a **path**. If a **path** is used, the granularity of the writers lock is at the level of the whole cell: two scoring engines cannot modify different parts of the same cell at the same time. Calls to “cell” (readers) are never blocked: they always see the old version of the cell until the “cell-to” function finishes (see Sec. 3.5).

The functions passed to “cell-to” are restricted: they should not be allowed to call any “cell-to” or “path-to” at any level of their call graphs. (That is, they cannot modify cells or pools and the functions that they call cannot modify cells or pools, including any functions those functions call, etc.) This constraint should be enforced by a PFA system that supports shared cells, since it excludes the possibility of deadlock.

7.8.3 Retrieving pool values: the “pool” special form

The “pool” special form is similar to “cell”, except that a **path** is always required.

```
{"pool": NAME, "path": INDEXES}
```

The **INDEXES** must not be empty, and the first item must be a literal string naming the desired object in the pool’s namespace.

7.8.4 Creating or changing pool values: the “pool-to” special form

The “pool-to” special form is similar to “cell-to”, except that a **path** is always required and **init** is required if and only if **to** specifies a function.

```
{"pool": NAME, "path": INDEXES, "to": VALUE}

{"pool": NAME, "path": INDEXES, "to": FUNCTION, "init": VALUE}
```

Unlike a cell, a pool object might not exist at runtime. If it does not exist, the **FUNCTION** that maps an old value to a new value cannot be applied, so the value specified by **init** is used instead. The **VALUE** specified by **init** has the same type constraints as the **VALUE** specified by **to**. The “pool-to” form changes the value of the pool and returns **null**.

The same concurrency issues apply to shared pools as to shared cells, except that **init** is used if a specified object does not exist. The check for existence and update are both in the same atomic operation. The granularity of the writers lock is on a single object in the pool, not the whole pool and not a part of the object.

Some PFA systems may implement shared cells and pools with a networked database. If so, they can take advantage of the **path** in the "cell" special form or the "pool" special form to only transfer the relevant subelement of the cell or pool over the network, rather than sending the whole data structure and extracting the subelement afterward.

7.9 Tree-like structures in the program flow

7.9.1 Expanding an expression into a mini-program: the "do" special form

A "do" special form allows the PFA document author to insert a series of expressions where one expression is expected. It has the following syntax.

```
{"do": ARRAY-OF-EXPRESSIONS}
```

where **ARRAY-OF-EXPRESSIONS** is a JSON array of expressions and the whole form is one expression. The return value is the value of the last expression in the JSON array, which must not be empty.

It is also possible for **ARRAY-OF-EXPRESSIONS** to be replaced with a single expression, though doing so would defeat the purpose of a "do" block. This option exists for symmetry with other, similar forms.

Example 7.10. A "do" form is also useful for declaring symbols in a sealed-within scope (see Sec. 5). For instance, new symbols usually cannot be declared in a function's argument list, but sometimes it is useful to insert a mini-program as an argument (in a purely functional style). For instance,

```
{"someFunction": [
     "simpleArgument",
     {"do": [
          {"let": {"x": 0}},
          {"while": {"notDone": "x"}, "do": {"set": {"x": {"iterate": "x"}}}}
          "x"
     ]},
     "simpleArgument"]}
```

The first and third arguments to **someFunction** are just references to a symbol named **simpleArgument**, but the second argument is a mini-program that calls **iterate** on a value until **notDone** is **false**.

Example 7.11. Although a "do" block can loosen a sealed-within scope, it cannot loosen its sealed-from-above attribute. Symbols declared outside of a sealed-from-above scope cannot be modified in that scope, even within a "do" block. For example, the following is invalid (a semantic error).

```
{"let": {"outerSymbol": 0}},
{"someFunction": [
     {"do": [
          {"set": {"outerSymbol": 1}},
          "outerSymbol"
     ]}]}
```

because the scope of function arguments are sealed-from-above.

Example 7.12. A “do” block does not apply any additional constraints on the scope of symbols. The following attempt to modify an outside symbols is valid.

```
{"let": {"outerSymbol": 0}},
{"do": [
     {"set": {"outerSymbol": 1}},
     "outerSymbol"
]}
```

7.10 Branching the program flow

Conditionals cause a branch in the program flow, in which the expressions that are evaluated depend on values at runtime.

7.10.1 Conditional with one or two cases: the “if” special form

The “if” special form has the following syntax:

```
{"if": CONDITION, "then": EXPRESSION-OR-EXPRESSIONS}
```

or

```
{"if": CONDITION, "then": EXPRESSION-OR-EXPRESSIONS, "else": EXPRESSION-OR-EXPRESSIONS}
```

where `CONDITION` is a single expression that evaluates to **boolean** and `EXPRESSION-OR-EXPRESSIONS` is either a single expression or a JSON array of expressions. The form without an `else` clause returns `null`, but the form with an `else` clause returns the narrowest supertype of the `then` and `else` clauses.

The `CONDITION` expression is evaluated in a sealed-from-above, sealed-within scope, but the `then` and `else` clauses are unsealed (see Sec. 5). That is, an outside symbol cannot be modified in the `CONDITION`, but it can be modified in the `then` and `else` clauses. In an imperative programming style, `then` and `else` are primarily used to assign values to symbols defined outside the “if”. In a functional programming style, the return value of the entire “if” form would be assigned to a symbol.

7.10.2 Conditional with many cases: the “cond” special form

The “cond” special form allows one to chain a series of conditionals.

```
{"cond": [{"if": CONDITION1, "then": EXPRS1}, {"if": CONDITION2, "then": EXPRS2}, ...]}
```

or

```
{"cond": [{"if": CONDITION1, "then": EXPRS1}, {"if": CONDITION2, "then": EXPRS2}, ...],
  "else": EXPRS}
```

The “if” forms within a “cond” are like stand-alone “if” forms except that they cannot have `else` clauses. The “cond” form can have a single `else` clause. Much like an “if”, the return value of “cond” is `null` if `else` is absent, and it is the narrowest supertype of all `then` clauses and the `else` clause if the `else` is present. A “cond” must have at least one “if”. The `CONDITION1`, `CONDITION2`, … are all single expressions that are sealed-from-above and sealed-within, and the `EXPRS1`, `EXPRS2`, … and `EXPRS` are either single expressions or JSON arrays of expressions that are unsealed. The `then` clause corresponding to the first (and only the first) successful `if` condition is evaluated, or the `else` clause is evaluated if no `if` conditions are successful.

7.11 Loops in the program flow

Loops repeatedly evaluate a set of expressions until some condition is met at runtime.

7.11.1 Generic pre-test loop: the "while" special form

The pre-test "while" special form has the following syntax.

```
{"while": CONDITION, "do": EXPRESSION-OR-EXPRESSIONS}
```

where `CONDITION` is a single expression that evaluates to **boolean** and `EXPRESSION-OR-EXPRESSIONS` is either a single expression or a JSON array of expressions. The return value of the "while" form is `null`: it can only be used to modify state. The scope of the `CONDITION` is sealed-from-above and sealed-within, but the scope of the `EXPRESSION-OR-EXPRESSIONS` is unsealed (see Sec. 5).

The `CONDITION` is evaluated before the `EXPRESSION-OR-EXPRESSIONS` and the two are evaluated in alternation until `CONDITION` returns `false`. If `CONDITION` returns `false` the first time it is called, the `EXPRESSION-OR-EXPRESSIONS` would never be evaluated.

7.11.2 Generic post-test loop: the "do-until" special form

The post-test "do-until" special form has the following syntax.

```
{"do": EXPRESSION-OR-EXPRESSIONS, "until": CONDITION}
```

Like "while", the `CONDITION` is a single expression that evaluates to **boolean** and `EXPRESSION-OR-EXPRESSIONS` is either a single expression or a JSON array of expressions. The return value is `null`. The difference is that the alternation between evaluating `EXPRESSION-OR-EXPRESSIONS` and `CONDITION` starts with the `EXPRESSION-OR-EXPRESSIONS` and continues until `CONDITION` is `true`. Thus, the `EXPRESSION-OR-EXPRESSIONS` is evaluated at least once, even if `CONDITION` is always `true`.

7.11.3 Iteration with dummy variables: the "for" special form

For loops are specialized loops that declare new symbols for use in the loop body. There are three basic types: "for" loops, which are usually used to increment a numerical index, "foreach" loops, which iterate over values of an array, and "forkey-forval" loops, which iterate over key-value pairs of a map.

The "for" special form has the following syntax.

```
{"for": NAME-TO-EXPRESSION, "while": CONDITION, "step": NAME-TO-EXPRESSION,
 "do": EXPRESSION-OR-EXPRESSIONS}
```

The `for` clause's `NAME-TO-EXPRESSION` is a JSON object whose members are new symbols to declare, the `step` clause's `NAME-TO-EXPRESSION` is a JSON object whose members are symbols to modify, and `while` is a **boolean**-valued single expression that stops the iteration when it becomes `false`. The `do` expression or expressions is the body of the loop, and the return value of the "for" form is `null`.

The `for` clause's `NAME-TO-EXPRESSION` is similar to a let expression in that it declares and initializes new symbols. The names of the symbols are the member names of the `NAME-TO-EXPRESSION` and the initial values are the evaluated results of the member values. The scope of the initialization expressions are sealed-from-above and sealed-within (see Sec. 5), and they are evaluated before anything else in the "if" form but the order of initialization expressions is not guaranteed. The symbols declared by the `for` clause can only be referenced in the `while`, `step`, and `do` clauses.

The `step` clause's `NAME-TO-EXPRESSION` is similar to a set expression in that it changes symbols, usually by incrementing them. The names of the symbols to modify are the member names of the `NAME-TO-EXPRESSION` and their new values are the evaluated results of the member values. The scope of the updator expressions are sealed-from-above and sealed-within, and if any updator expressions depend on a symbol that is also being updated, they will see the old value of the symbol. That way, they can be evaluated in any order without conflict, which is important because their order is not guaranteed.

The `while` expression is sealed-from-above and sealed-within. It is evaluated before the loop body (`do`), so it is possible for the loop body to never be evaluated.

The `do` expression or expressions is evaluated while the `CONDITION` is `true`. The symbols declared by the `for` clause may be referenced and even modified, though it is good practice to modify these loop variables only in the `step` clause. The body of the loop is unsealed, meaning that it can modify symbols defined outside of its scope.

Example 7.13. This is a basic for loop that indexes the loop iteration with an integer `i`.

```
{"for": {"i": 0}, "while": {"<": ["i", 10]}, "step": {"i": {"+": ["i", 1]}},
 "do": [
     {"something": i}
     {"somethingElse": i}
 ]}
```

The `something` and `somethingElse` functions are called with arguments from 0 (inclusive) until 10 (exclusive).

7.11.4 Iteration over arrays: the "foreach" special form

The "foreach" special form iterates over elements of an array. It has the following syntax.

```
{"foreach": NAME, "in": ARRAY-EXPRESSION, "do": EXPRESSION-OR-EXPRESSIONS}
```

or

```
{"foreach": NAME, "in": ARRAY-EXPRESSION, "do": EXPRESSION-OR-EXPRESSIONS,
 "seq": TRUE-OR-FALSE}
```

where `NAME` is a string naming the new symbol to declare, `ARRAY-EXPRESSION` is a single expression whose type is an `array`, the `do` expression or expressions is the body of the loop, and the return value of the "for" form is `null`. The `seq` flag, if present and set to `true`, ensures that the loop order through the array is sequential. This flag affects the scope of symbols, so that a PFA system has the option to evaluate the loop body in parallel if the `seq` flag is absent or false.

The symbol declared by the `foreach` clause can be referenced only in the body of the loop. The `ARRAY-EXPRESSION` is sealed-from-above and sealed-within. If the return type of the `ARRAY-EXPRESSION` is an array of `X`, the symbol declared by `foreach` has type `X`.

The loop body is evaluated once for every element in the array specified by the `in` clause. Since the array might be empty, the loop body might never be evaluated. If `seq` is absent or `false`, the order of elements is not guaranteed and the loop scope is sealed-from-above. Some PFA implementations may take advantage of this fact to parallelize the loop. (If so, all cells and pools referenced within the loop must be treated as though they are shared.) If `seq` is present and `true`, then the elements of the array are processed in order and the loop scope is not sealed-from-above.

7.11.5 Iteration over maps: the "forkey-forval" special form

The "forkey-forval" special form iterates over key-value pairs of a map. It has the following syntax.

```
{"forkey": NAME1, "forval": NAME2, "in": MAP-EXPRESSION,
 "do": EXPRESSION-OR-EXPRESSIONS}
```

where `NAME1` is a string naming a new symbol that iterates through map keys, `NAME2` is a string naming a new symbol that iterates through map values, `MAP-EXPRESSION` is a single expression whose type is a **map**, and the `do` expression or expressions is the body of the loop.

The symbols declared by the `forkey` and `forval` clauses can be referenced only in the body of the loop. The `MAP-EXPRESSION` is sealed-from-above and sealed-within. If the return type of the `MAP-EXPRESSION` is a map of `X`, the symbol declared by `forval` has type `X`. The symbol declared by `forkey` has type **string**.

The loop body is evaluated once for every key-value pair in the map specified by the `in` clause. Since the map might be empty, the loop body might never be evaluated. The order of elements is not guaranteed, but the loop scope is unsealed. An order-dependent calculation may yield different results every time it is evaluated, so it is good practice to only perform order-independent operations in a "forkey-forval" loop.

TODO: Add for-comprehensions for functional-style looping.

7.12 Type-safe casting

Unrestricted type-casting would invalidate the guarantees that a static type check provides. It is allowed in many popular programming languages, and if an incorrect type-cast is encountered at runtime, an exception (Java) or undefined behavior (C++) ensues.

However, it is possible to provide the advantages of type-casting without invalidating the static type check. Instead of returning the type-cast object as a single expression, we split the program flow into branches, one for each possible down-cast type. For instance, if a possibly-missing input datum has type **union(null, double)**, we split the program flow into a branch that handles the **null** case and a branch that handles the **double** case. Thus, the type-cast is a special kind of conditional.

7.12.1 Narrowing a type: the "cast-cases" special form

Down-casting (making a value's type more specific) is handled by the "cast-cases" special form, which has the following syntax.

```
{"cast": EXPRESSION, "cases": [
     {"as": TYPE1, "named": NAME1, "do": EXPRESSION-OR-EXPRESSIONS1},
     {"as": TYPE2, "named": NAME2, "do": EXPRESSION-OR-EXPRESSIONS2},
     ...
]}
```

or

```
{"cast": EXPRESSION, "cases": [
     {"as": TYPE1, "named": NAME1, "do": EXPRESSION-OR-EXPRESSIONS1},
     {"as": TYPE2, "named": NAME2, "do": EXPRESSION-OR-EXPRESSIONS2},
     ...],
 "partial": TRUE-OR-FALSE}
```

The `cast` clause evaluates a single `EXPRESSION` with some type `X`. The `as` clauses enumerate possible subtypes $\texttt{Y}_i$ of `X`. If `X` does not accept $\texttt{Y}_i$, then branch i can never be reached, which should result in a semantic exception.

If **partial** is absent or **false**, then the Avro type schemae named by **as** clauses should be exhaustive, there must be at least two cases, and the "cast-cases" form returns the last value of the followed branch. If **partial** is present and **true**, then the types named by **as** clauses do not need to be exhaustive, there must be at least one case, and the "cast-cases" form returns **null**. Only one branch is followed: the one with the first matching type schema.

The **named** clause takes a string that declares a new symbol with the given name. The new symbol can be referenced only within the corresponding **do** expression or expressions. Since the scopes of the **do** clauses do not overlap, cases may re-use names without shadowing.

The **cast** expression is sealed-from-above and sealed-within, but the **do** expressions are unsealed (see Sec. 5).

Example 7.14. This replaces a missing value in the **input** symbol by a default value, assuming that the PFA document's input type **[null, double]** (union of **null** and **double**). The "cast-cases" form is used as an expression that initializes **normalizedInput**.

```
{"let": {"normalizedInput":
     {"cast": "input", "cases": [
          {"as": "null", "named": "x", "do": -1000.0},
          {"as": "double", "named": "x", "do": "x"}
     ]}}}
```

Example 7.15. Alternatively, we could use an imperative programming style and overwrite a default value, like the following.

```
{"let": {"normalizedInput": -1000.0}},
{"cast": "input", "cases":
     [{"as: "double", "named": "x", "do": {"set": {"normalizedInput": "x"}}}]
 "partial": true}
```

7.12.2 Widening a type: the "upcast" special form

It is sometimes (rarely) useful to up-cast a value, making it less specific. This is handled by the "upcast" expression, which simply returns the casted value.

```
{"upcast": EXPRESSION, "as": TYPE}
```

The Avro type schema specified by **as** should accept the return type of **EXPRESSION** and raise a semantic error if it does not.

7.12.3 Checking missing values: the "ifnotnull" special form

A special case that is frequently used in PFA is to cast a union of **X** and **null** to **X**. Null is used to represent missing values, so this would branch the program flow based on whether the value is missing or not. Moreover, some datasets have many fields that could be missing and it is useful to isolate the case in which none of them are, in fact, missing. This could be accomplished with nested "cast-cases" forms, but the nesting level would be very deep and unwieldy, even for automated processing.

To simplify the PFA document, the "ifnotnull" special form only performs a **null**-removing cast, but it applies to one or more symbols simultaneously. It has the following syntax.

```
{"ifnotnull": {SYM1: EXPR1, ... SYMN: EXPRN}, "then": EXPRESSION-OR-EXPRESSIONS}
```

or

```
{"ifnotnull": {SYM1: EXPR1, ... SYMN: EXPRN}, "then": EXPRESSION-OR-EXPRESSIONS,
 "else": EXPRESSION-OR-EXPRESSIONS}
```

where `SYM1` . . . `SYMN` are as-yet undefined symbols, `EXPR1` . . . `EXPRN` are single expressions, and `EXPRESSION-OR-EXPRESSIONS` are expressions or JSON arrays of expressions.

All of the `EXPR1` . . . `EXPRN` expressions are evaluated, in an unspecified order, with sealed-from-above and sealed-within scope. Each of these expressions must have a union type that includes **null**, though they may have different types from one another.

If all of the `EXPR1` . . . `EXPRN` expressions are not null, their values are assigned to new symbols `SYM1` . . . `SYMN`, respectively. If the type of some `EXPR` is **union(X, null)**, the corresponding `SYM` has type **X**. If the type of some `EXPR` is **union(X, Y, . . . null)**, the corresponding `SYM` has type **union(X, Y, . . .)**. These symbols are only defined in the scope of the `then` clause.

If all of the `EXPR1` . . . `EXPRN` expressions are not null, the `then` clause is executed with the new symbols in scope. If any of the `EXPR1` . . . `EXPRN` expressions are null and the `else` clause exists, the `else` clause is executed without any new symbols in scope. The `then` and `else` clauses have no scope constraints.

The "ifnotnull" form without an `else` clause returns `null`, but the form with an `else` clause returns the narrowest supertype of the `then` and `else` clauses.

7.13 Miscellaneous special forms

7.13.1 Inline documentation: the "doc" special form

JSON has no means of including comments. Auto-generated code usually doesn't need comments, but there may be occasions in which it is useful to embed inactive statements in a PFA document.

The "doc" special form has the following syntax.

```
{"doc": STRING}
```

Unlike a comment in a programming language, it occupies a slot in the syntax tree. If inactive expressions are ever needed (the equivalent of Python's `pass` keyword), one could use a "doc" with an empty `STRING`. This form returns `null`.

7.13.2 User-defined exceptions: the "error" special form

PFA provides non-local exits to stop the evaluation of a `begin`, `action`, or `end` routine that behave like exceptions (see Sec. 3.6). Some library functions raise exceptions, but the PFA author can also raise user-defined exceptions. The PFA engine is free to halt execution or just skip to the next input datum upon encountering an exception.

The "error" special form raises user-defined exceptions. It has the following syntax.

```
{"error": STRING}
```

or

```
{"error": STRING, "code": INTEGER}
```

The `STRING` is the error message, and if a `code` is provided, it provides a safer way to identify specific errors than a human-readable message. The "error" form return type is **null**. Even though the "error" form can never return normally, this return type might be used to determine the type of an enclosing form, depending on its position.

7.13.3 Log messages: the "log" special form

The only three outputs that a PFA scoring engine can emit are (1) normal output, the result of a calculation, (2) an exception, and (3) log messages. Log messages should not be used for normal output, as they may or may not be handled by the host PFA system. The PFA system has full discretion to ignore, filter, collate, forward, or merge log messages.

The "log" special form emits a log message. It has the following syntax.

```
{"log": EXPRESSION-OR-EXPRESSIONS}
```

or

```
{"log": EXPRESSION-OR-EXPRESSIONS, "namespace": NAME}
```

The `EXPRESSION-OR-EXPRESSIONS` are the expressions to dump to the log (remember to wrap plain strings in `{"string": " ... "}` or `[" ... "]`); the optional `NAMESPACE` is a token that may be used for filtering or collating. The "log" form returns `null`.

8 Core library

The core library contains functions that would, in most languages, be infix operators. Following a LISP style of defining them as functions makes it easier to manipulate the expression tree with an automated algorithm.

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 comparison (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 (ˆ)

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 Hypotenuse (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

TODO: Define linear algebra functions. In addition to the standard operations on numerically indexed vectors and matrices, there should be ways to manipulate vectors and matrices whose rows and columns are named with strings (maps, rather than arrays).

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

Search and replace functions in the basic string library do not use regular expressions.

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

TODO: Define regular expression functions. Regular expressions should use the Perl-compatible (PCRE) syntax. Perhaps there should be some basic stemming, too (similar to PMML 4.2).

11 Array Manipulation

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

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
or	
`haystack`	array of any **`A`**
`needle`	function (**`A`**) $\rightarrow$ boolean
(returns)	int

Description: Count the number of times **`needle`** appears in **`haystack`** or the number of times the **`needle`** function evaluates to **`true`**.

11.2.3 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`**.

11.2.4 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`**.

11.2.5 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.

11.2.6 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` ≤ `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

The functions listed here provide the Cartesian product of the following features: (1) minimization and maximization, (2) using the natural Avro sort order or a custom less-than function, (3) returning only the most extreme value or an array of the N most extreme values, (4) returning the values themselves or the indexes of the values in the array. Each combination is provided as a separate function to avoid complicating the type signatures of the functions.

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`) $\rightarrow$ 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`) $\rightarrow$ 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**) $\rightarrow$ 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

These are the standard functors found in most functional programming contexts.

11.8.1 Map array items with function (a.map)

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

`a`	array of any `A`
`fcn`	function (`A`) $\rightarrow$ 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

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

TODO: Define map manipulation functions.

12.2 Record

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

TODO: Define record manipulation functions.

12.3 Enum

TODO: Define enum manipulation functions.

12.4 Bytes and fixed

TODO: Define bytes and fixed manipulation functions.

13 Missing data handling

Some methods for dealing with missing data are inseparable from the statistical model in question, such as adjustment factors for clustering or surrogate predicates for decision trees. Those that can be separated have been collected here.

"To impute" means to replace missing data with substituted values.

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`.

TODO: Define more imputers.

14 Aggregation

TODO: Define aggregation functions— SQL-like functions, group-by tables, CUSUM, etc.

15 Descriptive statistics libraries

This library contains methods for characterizing a dataset empirically. These are generally more lightweight than data mining models.

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.

TODO: Define more functions, accumulated mean, median or medoid(?); Tony Finch's algorithms...

16 Data mining models

This library contains methods of analyzing data using trained models. Many of these are usually the outputs of machine learning algorithms.

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` (score).

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` (score).

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.

TODO: Define more tree methods, including the integrated missing value methods from PMML (weightedConfidence, aggregateNodes). Some of those will require ScoreDistributions on the nodes.

16.2 Cluster models

TODO: Define cluster model functions. The metrics should all be separate functions, possibly in the math library.

16.3 Regression

TODO: Define regression functions. The fit function could be one of a few special cases (linear, polynomial, logistic) or a completely arbitrary user-defined function.

16.4 Neural networks

TODO: Define neural network functions. Separate the functions for simple perceptrons from the functions for neural network topologies, so that they're interchangeable. Backpropagation modifies a cell or pool in memory.

16.5 Support vector machines

TODO: Define support vector machine functions. The kernel trick can be implemented by passing a user-defined function.