WhizzML Reference Manual

The BigML Team

Version 0.48.7

MACHINE LEARNING MADE BEAUTIFULLY SIMPLE

info@bigml.com

Last updated March 28, 2024

About this Document

This document provides the full Reference Manual for the WhizzML language, including its standard library. Whizzml is a symbolic, functional language, with lambda abstractions, mapping constructs and an extensive standard library geared towards creation and manipulation of BigML resources to compose arbitrarily complex machine learning workflows.

Contents

CHAPTER 1

Basic Concepts

This document describes version 0.48.7 of the WhizzML language and runtime libraries.

1.1 Identifiers

WhizzML is case sensitive.

Most identifiers allowed in common, and not so common, programming languages are valid in WhizzML: any sequence of letters, digits and other characters that don't begin with a number are valid identifiers. The extended character set allowed in identifiers includes

```
! $ % & * + - . / < = > ? @ ^ _ ~
```

so all the following are valid WhizzML identifiers:

```
area
dataset-fields
model_23
removeStatusAndUpdateProgress!
number->string
percent%
pi^2
```

1.2 Whitespace and Comments

Whitespace characters are spaces and newlines, and a semicolon (;) indicates the start of a comment, which continues to the end of the line.

```
; this is a comment
;;; this too
(define a 3) ; starting at the semicolon, this is ignored
(define b (+ 1 a)) ;; ditto
```

1.3 Variables vs Syntax

An identifier may name a type of syntax or it may name a location where a value can be stored. If an identifier names syntax, it is called a *syntactic keyword* . If an identifier refers to a value's location, it is called a *variable* , and the value associated with the location the variable refers to is called the variable's *value* .

Some expressions are used to create new locations and bind variables to them: those expressions are called *binding constructs* . WhizzML's binding constructs are `lambda` expressions (Section 2.4), `let` expressions (Section 2.7), `map` expressions (Subsection 2.9.2), `for` expressions (Subsection 2.9.3), `loop`/`recur` expressions (Subsection 2.9.1) and definitions using the `define` keyword.

1.4 Types

No object satisfies more than one of the following predicates:

- `boolean?`
- `number?`
- `string?`
- `list?`
- `set?`
- `map?`
- `procedure?`

which define the corresponding types for WhizzML values. In addition, numbers can be further distinguished via the predicates:

- `integer?`
- `real?`

CHAPTER 2

Expressions

2.1 Variable References

An expression consisting of an identifier representing a variable is a *variable reference* , and its value is the value stored at the location the variable refers to.

2.2 Literal Expressions

2.2.1 Numbers

Constants of numeric types are expressed using the conventions and notation of Clojure for floating point values and integers

```
42
1.2e23
-2.1232E2
0x10    ;; hexadecimal 10 => 16
017     ;; octal 17 => 15
2r101   ;; binary 101 => 3
5r11    ;; base 5 11 => 6
2/3     ;; rational (/ 2 3) => 0.666666666
```

As seen in the examples, beyond the usual decimal and exponential literals, one can also write literal values using any base between 2 and 32 (with special notation for hexadecimal and octal) and use exact rational numbers, which behave as expected when arithmetically combined:

```
(+ 1/3 1/6 1/2)  ;; => 1
```

2.2.2 Strings and Booleans

Strings are quoted using " and must be single line (`"a string"`, `" another string."`).

Booleans have two literal values, `true` and `false` .

2.2.3 Lists

Literal lists can be written enclosing a list of space-separated literals in square brackets (`[]`), e.g.

```
[0 1 2 3]
["A" -42 true]          ;; lists can be heterogeneous
[["foo" 3] ["bar" 18]]  ;; and nested
[]                      ;; this is the empty list
```

2.2.4 Sets

Set literals are written by specifiying the literal list of the set's values, prepended by the symbol #:

```
#[1 2 3]
#["a" true [1 2] {"a" 3}]
#[a b]
```

Duplicate values in the list are automatically removed from the resulting set and the order of the elements in the list is irrelevant.

```
(= #[1 2 3] #[2 1 3]) ;; => true
(= #[1 2 3] #[2 1 3 1 2]) ;; => true
```

2.2.5 Maps

Map literals are represented by enclosing a sequence of alternating keys and values in braces ({}):

```
{"key0" 3 "key1" 2}    ;; a map with keys "key0" and "key1"

{"age" 74
 "name" "Alan"
 "scores" [1 0 0 2]                          ;; a map with
 "address" {"street" "sesame" "number" 3}}   ;; nested values
```

Values in map entries can have any type, and, like keys, can also include non-literal expressions:

```
(let (size 123
      color "green"
      outer-key "a thing"
      k "key")
  {outer-key {k size (str k 2) color}
   (str outer-key 2) [size color]})
;; => {"a thing" {"key" 123 "key2" "green"}
;;     "a thing2" [123 "green"]}
```

Although arbitrary expressions are allowed for the keys in map literals, they must eventually evaluate to string values.

2.3 Procedure Calls

A *procedure call* is written by simply enclosing in parenthesis an expression for the procedure to be called followed by expressions for its arguments. WhizzML is an eager language: the expressions for the procedure and its arguments (in that order) are fully evaluated before the procedure call.

Oftentimes, the expression for the procedure to be called is simply a variable reference, as in the following examples:

```
(+ 2 3 4)
(* (/ 1 2) x)
(rand)
(create-dataset {"source" src-id})
(wait (create-source {"remote" "http://host.com/foo.csv"}) 1000)
```

where we call the procedures referenced by the identifiers `+`, `/`, `*`, `rand`, `create-dataset`, `create-source` and `wait`. More generally, any expression that evaluates to a procedure is allowed, e.g.:

```
((if (= x 0) * /) 42 x)
```

which either multiplies or divides 42 by `x`, depending on whether the latter is zero or not.

It is also possible to apply a procedure to a list of arguments constructed on the fly by means of the standard library procedure `apply`, as described in Section 4.4.

2.4 Procedures

User-defined *procedure* values are created using the `lambda` keyword, with the following syntax for *lambda expressions* :

```
(lambda [<name>] <formals> <body>)

<name> ::= optional identifier for recursive calls in <body>
<formals> ::= (<id_1> ... <id_n>) || (<id_1> ... <id_n> . <id_n+1>)
   where <id_i> are identifiers
<body> ::= list of valid whizzml expressions
```

As shown, `<formals>` must have one of the following forms:

- `(<id_1> ... <id_n>)`: The procedure takes a fixed number of arguments. The number of arguments can be zero, in which case `<formals>` is just the empty list, .
- `(<id_1> ... <id_n> . <id_n1>)+`: If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period (n can be zero). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

A lambda expression evaluates to a procedure that can be called directly:

```
((lambda (x) (+ x 1)) 41)    ;; => 42
```

The optional `<name>` identifier can be used to refer to the lambda procedure itself within its body, for recursive definitions. For instance, this lambda expression computes the factorial of its argument:

```
(lambda fact (x) (if (> x 1) (* x (fact (- x 1))) 1))
```

As mentioned, `fact` is in scope only within the lambda's body, where it's bound to the procedure value itself. So we have:

```
((lambda fact (x) (if (> x 1) (* x (fact (- x 1))) 1)) 5) ;; => 120
(let (f (lambda fact (x) (if (> x 1) (* x (fact (- x 1))) 1)))
  (f 5)) ;; => 120
(let (fact (lambda fact (x) (if (> x 1) (* x (fact (- x 1))) 1)))
  (fact 5)) ;; => 120
```

or assigned a name with `define`, and subsequently applied using that name:

```
(define inc (lambda (x) (+ 1 x)))
(inc 41)
```

The `<formals>` (possibly empty) list can contain only valid identifiers and can have no duplicates. Some examples:

```
(lambda (x y) (+ y x))
(lambda () (rand-int 23))
```

and using a period separator we can denote a variable number of arguments:

```
((lambda (x y . z) z) 1 2 3 4 56) ;; => [3 4 56]
((lambda (. z) z) 1 2 3 4 56) ;; => [1 2 3 4 56]
((lambda (x . z) z) 1) ;; => []
```

See also Subsection 3.2.3 for additional syntactic sugar used to name user-defined procedures.

2.5 Maps and Lists as Procedures

List values can be used as procedures that, when applied to an integer value, return the value in the list at that position. In other words, lists can be interpreted as a procedure that maps integers to values. For instance:

```
(["a" "b" "c"] 0) ;; => "a"
(let (l [1 2 3]) [(l 2) (l 0) (l 1) (l 2)]) ;; => [3 1 2 3]
```

Besides the index, one can provide a default value to return if the position is out of bounds:

```
(["a" "b" "c"] 0 "d") ;; => "a"
(["a" "b" "c"] 3 "d") ;; => "d"
```

The action of lists as procedures is easily described in terms of the `nth` primitive (see Subsection 4.7.2). If `<list>` is an arbitrary list, `<n>` an integer and `<x>` any WhizzML value, we have the identities:

```
(<list> <n>) := (nth <list> <n>)
(<list> <n> <x>) := (nth <list> <n> <x>)
```

In a similar way, map values can be used as procedures that perform lookups of the key or key paths passed as first argument, with a second optional argument denoting the value to return if the given key or key path is not found. When a default value is not given, the map signals an error with code -15. For instance:

```
(let (m {"age" 24238 "name" "Treebeard"}
      p {"ent" m "other" 0})
  (m "age")  ;; => 24238
  (m "size") ;; => false
  (m "size" "huge")  ;; => "huge"
  (p ["ent" "name"]) ;; => "Treebeard"
  (p ["ent" "name"] "Lev") ;; => "Treebeard"
  (p ["a" "name"]) ;; => * Error (code -15) *
  (m ["age" "day"])) ;; => * Error (code -15) *
```

Note that if `<s_n>` are arbitrary string values, `<map>` a map and `<x>` an arbitrary default value, the following identities hold:

```
(<map> (list <s_0> <s_1> ... <s_n>)) := ((<map> <s_0>) (list <s_1> .. <s_n>))
(<map> (list <s_0> <s_1> ... <s_n>) <x>) := ((<map> <s_0>) (list <s_1> .. <s_n>) <x>)
```

The action of maps as procedures can also be described in terms of the primitives `get` and `get-in` (see Subsection 4.9.2) as follows:

```
(<map> <s>) := (get <map> <s>)
(<map> <s> <x>) := (get <map> <s> <x>)
(<map> (list <s_0> ... <s_n>)) := (get-in <map> (list <s_0> ... <s_n>))
(<map> (list <s_0> ... <s_n>) <x>) := (get-in <map> (list <s_0> ... <s_n>) <x>)
```

Lookups using a list of keys generalize to mixed lists of keys and positions, so that a composite value consisting of maps and lists can be traversed in a single call, using it in the function position. That behavior is inherited from the `get-in` primitive. For instance:

```
(let (m {"kind" "split"
         "children" [{"kind" "number" "value" 42}
                     {"kind" "list" "value" [1 2 3]}]}
      ls [0 m [1 2 3] m])
  (m ["children" 0]) ;; => {"kind" "number" "value" 42}
  (m ["children" 0 "value"]) ;; => 42
  (m ["children" 1 "value" 2]) ;; => 3
  (m ["children" 2 "value" 2]) ;; => * Error (code -15) *
  (ls [1 "kind"]) ;; => "split"
  (ls [2 2]) ;; => 3
  (ls [3 "children" 1 "value" 2])) ;; => 3
```

Despite the fact that maps and lists can behave as procedures, the primitive `procedure?` returns `false` when applied to map and list values.

2.6 Conditionals

WhizzML offers two conditional expressions, `if` and `cond`.

2.6.1 Conditionals with *if* and *when*

The `if` form takes a test expression as its first argument. If it evaluates to any value other than `false`, the consequent subexpression is evaluated, otherwise the (optional) alternate subexpression is evaluated.

```
(if <test> <consequent> <alternate>)
```

If `<alternate>` is not provided and `<test>` is `false`, the whole `if` expression evaluates to `false`. But when no alternate is used, it is better to use the special form `when`, which takes a test and a body (possibly consisting of multiple expressions) that is evaluated only if the test succeeds. Its general form is

```
(when <test> <body>)
```

which is equivalent to

```
(if <test> (prog <body>))
```

Examples:

```
(if (> x 3)
    (prog (log-info "Creating evaluation")
          (create-evaluation {"model" id}))
    (log-warn "No evaluation available"))

(if (> x 0)
```

```
    (/ 42 x)
    (if (< x 0)
        (/ -33 x)
        x))

(when (> x 0)
  (log-info "Positive case")
  (handle-positive-x x))
```

2.6.2 Conditionals with *cond*

WhizzML also offers a compact way of performing conditional code execution based on an arbitrary list of tests, using the `cond` keyword. The general syntax follows the pattern:

```
(cond <test1> <consequent1>
      <test2> <consequent2>
      ...
      <testn> <consequentn>
      <alternate>)
```

which is equivalent to a sequence of `if` expressions performing the same tests in the same order:

```
(if <test1>
    <consequent1>
    (if <test2>
        <consequent2>
        (if ....
            ....
               (if <testn>
                   <consequentn>
                   <alternate>) ...) ...))
```

In words, every test is evaluated in order until one of them yields a value different from `false`, in which case the full `cond` evaluates to the associated `<consequent>` subexpression. If all tests fail, the result of the expression is the result of evaluating the final `<alternate>` subexpression.

Examples:

```
(cond false 3
      false 4
      42)         ;; => 42

(cond (> x 3) "big"
      (< x 3) "small"
      "medium")
```

2.6.3 Logical *and*

```
(and <exp1> ... <expn>)
```

The `and` special syntax evaluates each of its arguments in turn until one of them yields a false value, which is then the result of the whole expression. The rest of the arguments are not evaluated, i.e., the form is short-circuited (which is the reason this is a special form, not definable as a user procedure).

```
(and (> 23 0) "foo" "bar") ;; => "bar"
(and (> x 0) (< (/ y x) 0.002))
```

2.6.4 Logical *or*

```
(or <exp1> ... <expn>)
```

The `or` special syntax evaluates each of its arguments in turn until one of them yields a truish value, which is then the result of the whole expression. The rest of the arguments are not evaluated, i.e., the form is short-circuited (which is the reason this is a special form, not definable as a user procedure).

2.7 Binding Constructs (*let*)

In order to define variables at local scope in whizzml, one uses `let` expressions, which have the following form:

```
(let (<id1> <val1>
      <id2> <val2>
      ...
      <idn> <valn>)
  <body>)
```

All left-hand expressions `<id1>` ... `<idn>` must be identifiers. Value expressions `<val1>` ... `<valn>` are evaluated in turn and their value bound to the corresponding identifier, which is then usable as a variable in subsequent value expressions and in `<body>`.

WhizzML is a statically scoped language. When evaluating `<valj>`, `<id1>` ... `<idj-1>` are visible (i.e., *in scope*), but not `<idj>`, and in subsequent value expressions, the computed value for `<idj>` will shadow any variable of the same name in an outer scope.

```
(define x "a string")

(let (x 42)
  (+ x 3))  ;; => 45

(let (x 0
      x (+ x 1)
      y (+ x 2))
  (+ x y))  ;; => 4

x ;; => "a string"
```

2.7.1 Binding List Destructuring with *let*

When an expression evaluates to a list, it is common to access its individual elements afterwards. That is easily accomplished using list accessors (see Subsection 4.7.2), but `let` bindings support special syntax for `sequential bind destructuring`, which allows concise assignment of list elements to named variables as follows.

Instead of using a single identifier in a `let` binding to specify a variable, one can use a literal list of identifiers. The right hand value must then be a list with, at least, as many values as there are identifiers. The form then binds each identifier to successive elements in the list value. For instance:

```
(let ([x y] [0 1])
  [y x])   ;; => [1 0]
(let ([x y] [0 1 2 3])
  [y x])   ;; => [1 0]
(let ([x] (range 3 4)
      [y z] [x (* 2 x)])
  [x y z])   ;; => [3 3 6]
```

More formally, a binding of the form:

```
(let ([<id0> ... <idn>] <v>
      ...)
 ...)
```

where `<id0>` ... `<idn>` are variable names and `<v>` is an arbitrary expression that evaluates to a list, is equivalent to the list of bindings:

```
(let (v <v>
      <id0> (nth v 0)
      <id1> (nth v 1)
      ...
      <idn> (nth v n)
      ...)
  ...)
```

with `v` an identifier that does not occur free in the expression `<v>`.

The dot notation used to denote rest-arguments in procedure parameter declarations (see Section 2.4) is available in sequence destructure to bind the tail of the list to a name. For instance:

```
(let ([x y . z] [0 1 2 3 4])
  z)   ;; => [2 3 4]
(let ([x y . z] [0 1])
  z)   ;; => []
(let ([. r] (range 3))
  r)   ;; => [0 1 2]
```

That is, a binding of the form:

```
(let ([<id0> ... <idn> . <y>] <v>
      ...)
  ...)
```

where `<id1>` ... `<idn>` and `<y>` are variable names and `<v>` is an arbitrary expression that evaluates to a list, is equivalent to the list of bindings:

```
(let (v <v>
      <id0> (nth v 0)
      <id1> (nth v 1)
      ...
      <idn> (nth v n)
      <y> (drop n v)
      ...)
   ...)
```

where `drop` is a primitive that discards the first `n` elements of a list, `v` an identifier that does not occur free in the expression `<v>`.

2.8 Sequencing

`prog` can be used to group together a list of expressions, which are evaluated unconditionally. The value of the last one returned as the value of the whole expression.

```
(prog <expression1> ... <expressionn>)
```

This form is useful in contexts where you can only write a single expression, such as conditional branches, but one to perform some side-effecting operation before returning the actual value of the expression:

```
(cond (> x 0) (prog (log-info "Positive value!") x)
      (< x 0) (prog (log-info "Negative value!") 0)
      (prog (log-info "Zero, returning -1") -1))
```

2.9 Iteration

Iteration is accomplished explicitly using the `loop` and `recur` keywords, or implicitly using either `map` or `for`, as well as the standard library procedure `reduce` and the special syntax `iterate` (see Subsection 4.7.7 for details on the latter two).

2.9.1 Iteration with *loop/recur*

Generic iteration can be attained using the loop-recur form, which has the structure:

```
(loop (<id0> <val0> ... <idn> <valn>) <body>)

<id0> ... <idn> := identifiers bound as variables in <body> with
                   values <val0> ... <valn> in the first iteration

<body> := an arbitrary body that can contain calls of the form
          (recur <val0'> ... <valn'>), which cause the loop body
          to be re-entered with <id0> ... <idn> bound to the new
          values <val0'> ... <valn'>
```

For instance, this loop reverses the list `[1 2 3 4]`:

```
(loop (in [1 2 3 4]
       out [])
  (if (= [] in)
      out                   ;; we're done iterating, return the result
      (recur (tail in)      ;; we iterate with in -> (tail in)
             (cons (head in) out)) ;;    and  out -> (cons (head in) out)
```

and in this other example we compute the number of even values in an input list:

```
(loop (in input-list
       cnt 0)
  (cond (= [] input-list) cnt
        (= 0 (rem (head input-list) 2)) (recur (tail input-list) (+ 1 cnt))
        (recur (tail input-list) cnt)))
```

The special syntactic form `iterate` and the standard procedure `reduce`, which are fully described in Subsection 4.7.7, simplify writing loops that compute a final value by traversing several lists.

2.9.2 List Value Mapping with *map*

Although it could be defined as a user procedure, WhizzML provides a built-in `map` that applies a given procedure (its first argument) to each of the elements of the lists provided as a second and subsequent arguments, and returns the list of results.

```
(map <proc> <list0> <list1> ...)
```

For example:

```
(map (lambda (x) (+ x 1)) [2 4 6])   ;; => [3 5 7]
(map (lambda (x y) (+ x y)) [1 2 3] [2 3 4])   ;; => [3 5 7]
(map list [true false] [1 2] ["a" "b"]) ;; => [[true 1 "a"] [false 2 "b"]]
```

For a single list, `map` is functionally equivalent to the following naive definition:

```
(define (map fn lst)
  (cond (empty? lst) lst
        (cons (fn (head lst)) (map fn (tail lst)))))
```

For multiple lists, `<proc>` must take as many arguments as lists are passed in the call, and the mapping stops when the shortest of the given lists is exhausted. For example:

```
(map (lambda (x y z) (+ x y z)) (range 10) (range 20) (range 2))  ;; => [0 3]
(map (lambda (x y) (if (> x y) x y)) [1 2 3 4] [-1 5])  ;; => [1 5]
```

Despite its being a special form, `map` can be used as a procedural value, i.e., as an argument to other higher order functions, but it's worth noting that in those cases the performance of the multiple-lists version will be noticeably worse than the built in invoked when `map` is used in call position.

2.9.3 List Value Mapping with *for*

The `for` form is an alternative form of expressing mapping over lists where we use a body for the values to be computed in terms of a variable that iterates over a given list.

```
(for (<id> <list>) <body>)
```

which is equivalent to:

```
(map (lambda (<id>) <body>) <list>)
```

Examples:

```
(for (x [1 2 3]) (+ x 1))    ;; => [2 3 4]

(for (x (range 0 10))
  (if (= 0 (rem x 3)) x 0))  ;; => [0 0 0 3 0 0 6 0 0 9]
```

2.10 Error Handling

Error handling in whizzml is based on exceptions raised either by the runtime or by the user via the `raise` procedure, which throws a given value as an error.

Objects thrown by `raise` can be of any type, but when the error comes from a predefined function, it will always be a map with, at least, the keys "message" and "code".

There are two mechanisms For capturing exceptions: a basic, lower-level one based on the form `handle`, and the familiar `try`/`catch` construct (which is built as syntactic sugar over the basic handle functionality).

2.10.1 Signaling Errors with *raise*

To signal an error in user code, use the `raise` keyword.

```
(raise <error>)
```

The above expression throws the value `<error>` as an exception, to be captured by an exception handler (see below), or, if none is active, to cause the program to stop.

As mentioned, you can throw any valid value as an error:

```
(raise "Connection problems")
(raise -23)
(raise {"message" "Empty dataset" "code" 42})
```

However, it is common to use a map to signal errors, as do by default all built-in procedures (if needed). The latter always contain the keys "message", "code" and "instruction"; here's an example of an error raised by the / operator:

```
{"message" "Error computing primitive operation '/': Divide by zero"
 "code" -1
 "instruction" {"source" {"lines" [1 1] "columns" [0 6]} "instruction" "apply"}}
```

But that is just a convention: `raise` will accept any WhizzML value and propagate it as an error to the currently active error handler.

2.10.2 Capturing Errors with *handle*

Here is an example that captures a divide by zero exception using the `handle` syntactic form:

```
(handle (lambda (e) (log-error e) 42)
  (/ 1 0))  ;; => 42
```

The first argument of `handle` is a function of one argument that is called when the body of the handle form (the s-expressions following the error handling function) signals an error. The above program will log the captured error `e` (a divide by zero exception) and return 24: the value of the whole `handle` expression is the value returned by the handle function (when an error is thrown) or the value of its body if no error is raised.

In general,

```
(handle <handler> <body>)
```

registers the single-argument procedure `<handler>` as the active error handler while the forms in `<body>` are being executed. If the evaluation of `<body>` raises an error, the value thrown will be passed to the `<handler>` procedure and the value of the `handle` expression will be the value that the call to `<handler>` returns.

The handler can also be a variable whose value is a single-argument procedure, as in the following example:

```
(define (on-error e)
  (let (code (get e "code"))
    (cond (= e -1) 0
          (= e 1) -5
          (raise e))))

(handle on-error
  (when (negative? (get-x))
    (raise {"code" -1 "message" "Error: negative x"}))
  (do-something (get-x)))
```

As you can see, the body of `handle` can contain more than one form and it is legit to re-raise errors (or throw new ones) inside an error handler: they will be passed to the previously registered handler, if any, or just propagate to the top-level and stop the program otherwise.

2.10.3 Capturing Errors with *try/catch*

WhizzML also includes syntactic sugar for handling errors via the `try` and `catch` keywords.

```
(try <body> (catch <id> <handler-body>))
```

executes `<body>` and, in case an error is raised, binds it to the variable `<id>` which is in the scope of `<handler-body>` , to which control is transferred. In other words, the try/catch form is equivalent to

```
(handle (lambda (<id>) <handler-body>) <body>)
```

For instance:

```
(try
  (log-info "Trying primary source")
  (create-dataset {"source" "source/12367890795948224 5aa31"})
  (catch e
    (log-warn "Could not create primary source: " e)
    (create-dataset {"source" "source/123456789012345678 90abcd"})))
```

2.10.4 System Errors

Built-in and standard library procedures always raise errors in the form of a map with keys “message” and “code”, as in the following example:

```
{"code" -10
 "message" "Error computing primitive operation '/': Divide by zero"}
```

The error codes used by built-in and standard whizzml procedures are shown in Table 2.1.

Code	Cause
-10	Division by zero
-15	Key not found
-20	Empty list
-25	List of less than two elements
-30	Arguments out of range
-40	Incorrect number of arguments
-50	Error handling BigML resource
-60	BigML resource creation failed
-100	Generic exception

Table 2.1: Error codes

In addition to these WhizzML system codes, errors raised while creating, modifying or fetching API resources use the same codes as the API, as listed in the API documentation[1]. For instance, here is an error thrown while trying to create a source with a malformed request; the code below:

```
(handle (lambda (e) e)
        (create-dataset {"source" "source/123456789012345678901234"}))
```

will evaluate to a map value (thrown by the standard library function `create-datsaet`), that contains, at least, two keys:

```
{"message"
 "Error computing primitive operation 'create': Id does not exist",
 "code" -1201}
```

[1] https://bigml.com/developers/status_codes

CHAPTER 3

Program Structure

3.1 Programs

A WhizzML program consists of a sequence of expressions and definitions (cf. Chapter 2), and it evaluates to the value of the last expression in the program.

Definitions are only allowed at the top level, and they bind variable identifiers to values (including procedural values) at global scope.

Expressions and definitions in a program are evaluated sequentially. At the top level of a program, any expression of the form:

```
(prog <e1> ... <en>)
```

is totally equivalent to the sequence of expressions and definitions in the body of the `prog`, that is to the sequence of expressions

```
<e1>
...
<en>
```

3.2 Definitions

3.2.1 Variable Assignments

A *global variable* is associated with a (possibly newly created) location and value by means of the `define` keyword:

```
(define <id> <expression>)
```

E.g.:

```
(define age 46)
(define name "Biel")
(define same-name name)
```

3.2.2 Parallel Variable Assignments

It is possible to define in parallel more than one variable with a singl `define` form by using an explicit list of identifiers instead of a single one:

```
(define [<id0> ... <idn>] <expression>)
```

The above expression will be translated to the semantic equivalent of:

```
(define tmp-var <expression>)
(define <id0> (nth tmp-var 0))
...
(define <idn> (nth tmp-var n)
```

where `tmp-var` is a fresh, private name. From this equivalence it follows that `<expression>` can evaluate to a list of more than `n` elements: the trailing ones will simply be ignored.

For instance:

```
(define [a b] [1 2])
a ;; => 1
b ;; => 2
(define (alist h) (list h true false "extra"))
(define [c d e] (alist "a"))
c ;; => "a"
d ;; => true
e ;; => false
```

As in the case of destructuringlet bindings, dot notation is also available (cf. Subsection 2.7.1):

```
(define [<id0> ... <idn> . <y>] <expression>)
```

translates to:

```
(define tmp-var <expression>)
(define <id0> (nth tmp-var 0))
...
(define <idn> (nth tmp-var n)
(define <y> (drop n tmp-var))
```

For instance:

```
(define [a b . c] (range 10))
a ;; => 0
b ;; => 1
c ;; => [2 3 4 5 6 7 8 9]
```

3.2.3 Procedure Definitions

Since naming procedure values by assigning them to a variable via `define` is a very common need, WhizzML provides special syntax for doing it in a shorter form. Formally, the two following definitions are equivalent:

```
(define <p-id> (lambda (<id1> ... <idn>) <body>))
(define (<p-id> <id1> ... <idn>) <body>)
```

as are the two following ones for variadic procedures:

```
(define <p-id> (lambda (<id1> ... <idn> . <idn+1>) <body>))
(define (<p-id> <id1> ... <idn> . <idn+1>) <body>)
```

Thus, instead of defining the `inc` procedure as

```
(define inc (lambda (x) (+ x 1)))
```

one can write, equivalently and more idiomatically,

```
(define (inc x) (+ x 1))
```

The same pattern applies for any number of arguments:

```
(define rand-less-than-42 (lambda () (rand-int 42)))
(define (rand-less-than42) (rand-int 42))

(define safe-div (lambda (x y) (if (= y 0) 0 (/ x y))))
(define (safe-div x y) (if (= y 0) 0 (/ x y)))
```

and, in the same way, to functions that take a variable number of arguments. Instead of

```
(define add (lambda (x . xs) (apply + x xs)))
(define to-list (. xs) xs)
```

one can define `add` and `to-list` more directly with:

```
(define (add x . xs) (apply + x xs))
(define (to-list . xs) xs)
```

CHAPTER 4

Standard Procedures

This chapter provides an exhaustive description of WhizzML's standard library procedures. For each one we give first its signature and return type, as in this template:

```
(proc-name obj1 obj2)                                                    return-type
```

Optional arguments are enclosed in square brackets:

```
(proc-name obj1 [obj2])                                                  return-type
```

When a procedure can take an arbitrary number of arguments (possibly after some required ones), we denote it by means of an ellipsis. For instance, a procedure taking one or more arguments is denoted as:

```
(proc-name obj1 ...)                                                     return-type
```

The name of the arguments in the template reflects their accepted types, using one of the following prefixes:

- `obj` for any type.
- `str` for a string value.
- `num` for a numeric value.
- `int` for an integer.
- `bool` for a boolean value.
- `list` for an arbitrary list.
- `map` for an arbitrary map.
- `proc` for a procedure.
- `res` for a resource identifier.

4.1 Utilities

4.1.1 Identity

```
(identity obj)                                                           any
```

The `identity` procedure just returns its passed argument, whatever its value.

```
(= (identity x) x) ;; => true (identically)
```

4.1.2 Versioning

It is possible to access at runtime the current WhizzML version, both as a string and as three separated integers giving its major, minor and micro components:

```
(version )                                                          string
(version-major )                                                   integer
(version-minor )                                                   integer
(version-micro )                                                   integer
```

```
(version) ;; => "0.16.1"
(version-major) ;; => 0
(version-minor) ;; => 16
(version-micro) ;; => 1
```

4.2 Equality

```
(= obj1 ...)                                                       boolean
(!= obj1 ...)                                                      boolean
```

Values of any type can be compared for (structural) equality using the `=` procedure which takes one or more arguments and evaluates to `true` only if all the values are equal. For example:

```
(= 1)  ;; => true
(= 1 "one") ;; => false
(let (x "one") (= x "one"))  ;; => true
(= {"a" 3 "b" "hello"}
   {"b" "hello" "a" (+ 1 2)}
   {"a" 3 "b" (str "he" "llo")})  ;; => true
```

The operator `!=` is the logical complement to `=`.

```
(!= "hi") ;; => false
(!= "hi" "hi")   ;; => false
(!= "Hi" "hi")    ;; => true
(!= 1 (- 2 1) 3)  ;; => true
```

The resources and objects created in WhizzML can be compared using the `compare-objects` procedure.

```
(compare-objects res res)                                           list
```

The result of the call is a list of the attributes that differ. For identical objects, the result is an empty list, e.g.:

```
(compare-objects {"first-key" "string"
                  "second-key" 2
                  "third-key" [1 2]}
                 {"first-key" "string"
                  "second-key" 2
                  "third-key" [1 2]})
;; => []
```

If the objects are different, the list has an element for each different attribute. It describes the path to the attribute in the map structure, the value of the attribute in each object, the type of difference and a message describing it. The type of differences range from different value types, different lengths for lists, different values or missing keys.

```
(compare-objects {"first-key" "string"
                  "second-key" 2
                  "third-key" [1 2]}
                 {"first-key" "a different string"
                  "second-key" 3
                  "third-key" [1 4]
                  "fourth-key" "new attribute"})
;; => [{"path1" ["first-key"]
;;       "path2" ["first-key"]
;;       "obj1" "string"
;;       "obj2" "a different string"
;;       "type" "eq"
;;       "msg" "The strings in [first-key] differ."}
;;      {"path1" ["second-key"]
;;       "path2" ["second-key"]
;;       "obj1" 2
;;       "obj2" 3
;;       "type" "eq"
;;       "msg" "The numbers in [second-key] differ."}
;;      {"path1" ["third-key" 1]
;;       "path2" ["third-key" 1]
;;       "obj1" 2
;;       "obj2" 4
;;       "type" "eq"
;;       "msg" "The numbers in [third-key 1] differ."}
;;      {"path1" []
;;       "path2" ["fourth-key"]
;;       "obj1" ""
;;       "obj2" "new attribute"
;;       "type" "missing"
;;       "msg" "The key fourth-key is missing in the left hand side."}]
```

4.3 Logical Functions

Besides the special forms `and` and `or` (see Section 2.6), the standard library provides the unary `not` function, which complements its boolean argument.

```
(not obj)                                                    boolean
```

The negation of any non-boolean value is `false`, as is the negation of `true`, i.e., any value different from `false` represents truth in the language:

```
(not 3) ;; => false
(not {}) ;; => false
(not []) ;; => false
(not [1 2]) ;; => false
(not true) ;; => false
(not false) ;; => true
```

4.4 Procedures

It is possible to *apply* a procedure to a list of arguments, by means of the procedure `apply`. In its simplest form, `apply` takes two arguments: the function to call, and a list of the arguments to use to call it:

```
(apply <proc> <args-list>)
```

For example:

```
(apply + [1 2 3]) ;; => (+ 1 2 3) => 6
(apply list ["a" "b"]) ;; => (list "a" "b") => ["a" "b"]
```

You can easily see that, in general, `(apply list x)` evaluates to `x` for any list value of `x`.

In general, `apply` can take multiple arguments besides the procedure to call. The full signature of `apply` is the following:

```
(apply proc objarg1 ...  list-args)                                    any
```

`proc` must be a procedure and `list-args` must be a list. `apply` then calls `proc` with the elements of the list `(concat (list arg1 ...)  list-of-args)` as the actual arguments.

```
(define compose
   (lambda (f g)
     (lambda (. args)
       (f (apply g args)))))

 ((compose sqrt *) 12 75)   ;; =>  30
```

Given a procedure `f`, one can partially apply it to a list of values and obtain a new procedure of lower arity using the standard procedure `partial`:

```
(partial proc objarg1 ...)                                          procedure
```

`partial` takes a procedure `proc` and fewer than the normal number of arguments to `proc`, and returns a new procedure that takes a variable number of additional args. When called, the returned function calls `proc` with `objargs1`... plus the additional args.

```
(map (partial + 2) [1 2 3]) ;; => [3 4 5]
(map (partial + 2 4) [1 2 3]) ;; => [7 8 9]

(define prep-ab (partial concat ["a" "b"]))
(prep-ab) ;; => ["a" "b"]
(prep-ab [3 1]) ;; => ["a" "b" 3 1]
(prep-ab ["x" "y"] [false true 2]) ;; => ["a" "b" "x" "y" false true 2]
```

4.5 Numbers

4.5.1 Numerical Type Predicates

Predicates checking whether any value is a number, integer or real.

```
(number?  obj)                                                      boolean
(integer?  obj)                                                     boolean
(real?  obj)                                                        boolean
```

```
(integer? 0.0) ;; => false
(real? "hello!") ;; => false
(real? 1) ;; => true
(integer? 1) ;; => true
```

4.5.2 Arithmetic Operators

The basic arithmetic operators take one or more arguments and are represented by their usual symbols:

```
(+ num1 ...)                                                    number
(- num1 ...)                                                    number
(* num1 ...)                                                    number
(/ num1 ...)                                                    number
```

Division raises error code -10 if division by zero. If passed only one argument, / computes its inverse.

In addition, the standard library includes the following binary and unary operators on numbers:

```
(rem int1 int2)                                                 integer
```

Computes the remainder of dividing `int1` by `int2`. Raises error code -10 if the latter is zero.

```
(div int1 int2)                                                 integer
```

Computes the integer division `num1` by `int2`. Raises error code -10 if the latter is zero.

```
(sqrt num1)                                                     number
```

Computes the square root of its argument. Raises error code -30 if passed a negative number.

```
(+ 1) ;; => 1
(+ 1 -1) ;; => 0
(- 23.8 18.2) ;; => 42.0
(/ 2) ;; => 0.5
(/ 1 2 3) ;; => 0.16666667
(rem 23 3) ;; => 2
(div 45 7) ;; => 6
(sqrt 24) ;; => 4.898979485566356
(sqrt 0.144) ;; => 0.3794733192202055
```

4.5.3 Numeric Coercion and Parsing

The following operators act on any number, returning an integer in all cases:

```
(abs num)                                                       integer
```

The absolute value of its argument.

```
(ceil num)                                                      integer
```

The ceiling, that is, the lowest integer greater than or equal to the given value.

```
(floor num)                                                     integer
```

The floor of a number is the biggest integer less than or equal to the given number.

```
(round num)                                                     integer
```

Rounding finds the integer value that is closest to the given number.

```
(abs -21) ;; => 21
(abs 3.0) ;; => 3.0
(ceil 23.3) ;; => 24
```

```
(ceil -128.2) ;; => -128
(floor 12.8) ;; => 12
(floor -12.3) ;; => -13
(round 1.2) ;; => 1
(round 1.5) ;; => 2
```

```
(read-number str)                                                    number
```

The standard library provides `read-number` to parse a string representing a number, using any of the accepted number literal expressions in WhizzML. This procedure raises error code -30 (domain error) if passed a string that cannot be parsed.

```
(read-number "0.1232") ;; => 0.1232
(read-number "0.1232a") ;; => Error
(read-number "-32") ;; => -32
(read-number "0x32") ;; => 50 (hexadecimal notation)
(read-number "032") ;; => 26 (octal notation)
(read-number "2r111") ;; => 7 (explicit base)
(read-number "3/2") ;; => 3/2 (rational number)
```

4.5.4 Comparisons

```
(< num1 ...)                                                         boolean
(<= num1 ...)                                                        boolean
(> num1 ...)                                                         boolean
(>= num1 ...)                                                        boolean
```

Comparison operators (like, e.g. `<`) are multivariadic and can take more than two arguments, so that, say, `(< x y z)` is equivalent to `(and (< x y) (< y z))`, or the mathematical expression `x < y < z`.

```
(< 1 2) ;; => true
(<= 0 0 3) ;; => true
(> -19 -18) ;; => false
(> -19 -20 -21) ;; => true
(<= 1 1 2 2 2 3 4 44) ;; => true
(< 0) ;; => true
(< -1) ;; => true
(< 12.343) ;; => true
```

Comparison operators can also take a single argument, in which case they always evaluate to `true`

Convenience predicates for direct comparison to zero and parity are also provided:

```
(zero?  num)                                                         boolean
(positive?  num)                                                     boolean
(negative?  num)                                                     boolean
(even?  int)                                                         boolean
(odd?  int)                                                          boolean
```

```
(zero? 0) ;; => true
(zero? 0.0) ;; => true
(zero? -2) ;; => false
(positive? 0) ;; => false
(positive? 0.1) ;; => true
(negative? 0) ;; => false
(negative? 0.0) ;; => false
```

```
(even? 0) ;; => true
(odd? 3) ;; => true
```

Furthermore, WhizzML provides procedures for computing the maximum and minimum of any number of arguments:

```
(min num1 ...)                                                    number
(max num1 ...)                                                    number
```

```
(max 10) ;; => 10
(max -1 -20.0 30 10.2) ;; => 10
(min -1.1) ;; => -1.1
(min (sqrt 2) (sqrt 3)) ;; =>  1.4142135623730951
```

4.5.5 Transcedental Functions

Common non-algebraic functions on numbers are also available:

```
(log num)                                                         number
(log2 num)                                                        number
(log10 num)                                                       number
```

Logarithms in base e, 2 and 10. These procedures take a positive number as argument, and raise error code -30 otherwise.

```
(pow num1 num2)                                                   number
(exp num1)                                                        number
```

The general power function returns $num1^{num2}$, and `exp` uses e as its base, computing e^{num1}.

```
(sin num1)                                                        number
(cos num1)                                                        number
(tan num1)                                                        number
```

Standard trigonometric functions: sine, cosine and tanget. For all of them, `num1` is an angle in radians. We also provide their inverses:

```
(asin num1)                                                       number
(acos num1)                                                       number
(atan num1)                                                       number
```

Both `asin` and `acos` take a number in the interval $[1, -1]$, raising error code -30 otherwise.

```
(to-degress num1)                                                 number
(to-radians num1)                                                 number
```

Conversion functions from radians to degrees and vice versa.

```
(sinh num1)                                                       number
(cosh num1)                                                       number
(tanh num1)                                                       number
```

These are the standard hyperbolic functions.

```
(gamma num1)                                                      number
```

The gamma function for real-valued arguments is also available. Note that it is undefined for non-positive integer arguments (including 0).

```
(log 2) ;; => 0.6931471805599453
(log (exp 1)) ;; => 1.0
(pow 4 (/ 1 2)) ;; => 2.0
(+ (pow (sin 1.2) 2) (pow (cos 1.2) 2)) ;; => 1.0
(tan (atan -1.2)) ;; => -1.2
(sinh 23.32) ;; => 6.709919691913042E9
(gamma 2) ;; => 1.0000000000000002
```

4.5.6 Random Number Generators

Each instance of a WhizzML runtime has an implicit random number generator (RNG). The random number series generated by this RNG are accesible via the following standard procedures omitting the optional argument `rnd-id` in all cases.

```
(rand [rng-id])                                                    number
(rand-range num1 num2 [rng-id])                                    number
(rand-int num1 [rng-id])                                          integer
(set-rng-seed obj [rng-id])                                       boolean
```

The `rand` procedure generates a random number in the interval $[0, 1]$, whereas `rand-range` uses the range $[\text{num1}, \text{num2}]$ (with both ends arbitrary numbers). Finally, `rand-int` returns an integer between 0 and its argument.

The implicit RNG is seeded at random every time a virtual machine is started, so that two runs of the same program using the functions above will in general obtain different number sequences from the generator. To produce deterministic results, you can seed the RNG using `set-rng-seed` and providing any WhizzML value as seed.

```
(rand) ;; => 0.5778650511056185
(rand-int 2313) ;; => 501
(rand-range -10 3) ;; => 1.4949024706147611
(set-rng-seed "a seed")  ;; => true
```

Instead of using the implicit RNG, you can create your own RNG instances via `create-rng`, and use the returned identifier as the `rng` argument in the functions above.

```
(create-rng [obj])                                                 rng-id
```

The value returned by `create-rng` is a unique identifier for the RNG just created.

```
(define rng-id (create-rng 42))  ;; any value can be used as seed
(rand rng-id)  ;; => 0.13599370513111353
(rand-int 100 rng-id) ;; => 59
(rand-int 100 rng-id) ;; => 10
(rand-range -10 10 rng-id) ;; => -2.855083905160427
(set-rng-seed rng-id "whizzml")  ;; => true
```

You should treat the random indentifier value as an opaque token. Everytime the seed of an RNG is set, the generator starts over, even if the seed is the same that was used before.

Typically, if you are writing a library that uses random numbers and want to make it deterministic, you will be using in it your own RNG, created via `create-rng` with an appropriate key. In that way, you won't interfere with other libraries or scripts using the library at hand, nor get affected by other code setting the implicit RNG's seed.

4.5.7 Basic Statistics

```
(mean list)                                                    number
(variance list)                                                number
(stdev list)                                                   number
(chi-squared-test num int)                                     number
```

`mean` computes the mean value of a list of numbers, `variance` their variance and `stdev` their standard deviation. `chi-squared-test` returns the p-value for a given value and degrees of freedom.

`mean` will raise error code -20 if passed an empty list, and both `variance` and `stdev` signal code -25 if passed a list with less than two numbers. `chi-squared-test` will raise error code -20 if the first argument is negative, or the second argument is either non-positive or non-integer.

4.6 Strings

```
(string?  obj)                                                 boolean
```

String values can be recognized via the predicate `string?`.

4.6.1 Coercion to String

```
(str obj1 ...)                                                 string
```

Any value can be coerced to a string using `str`. The procedure takes an arbitrary number of arguments, and the result is the concatenation of all the coercions. When acting on string values, `str` is thus the string concatenation function.

```
(str 3) ;; => "3"
(str 3 5) ;; => "35"
(str "3" (+ 3 2)) ;; => "35"
(str "Hello" " " "world") ;; => "Hello world"
```

As shown in the examples above, `str` behaves as string concatenation for arguments of type string. To preserve quotations associated to strings in the result (for instance, because you are generating WhizzML source code), use the standard procedure `pr-str`.

```
(pr-str obj1 ...)                                              string
```

```
(pr-str 3) ;; => "3"
(pr-str 3 5) ;; => "35"
(pr-str "3" (+ 3 2)) ;; => "\"3\"5"
(pr-str "Hello" " " "world") ;; => "\"Hello\"\" \"\"world\""
```

We also provide a standard procedure that generates a JSON representation a given WhizzML value:

```
(json-str obj1)                                                string
```

```
(json-str 3) ;; => "3"
(json-str [2 true]) ;; => "[2,true]"
(json-str {"a" 2.2 "b" [1 [false "c"]]}) ;; => "{\"a\":2.2,\"b\":[1,[false,\"c\"]]}"
```

and it is also possible to parse a JSON string into its corresponding WhizzML value:

```
(read-json-str strjson)                                        object
```

```
(read-json-str "3") ;; => 3
(read-json-str "[2, true]") ;; => [2 true]
(read-json-str (json-str {"A" 2})) ;; => {"A" 2}
```

4.6.2 Digests

There are three hashing procedures available in the standard library:

```
(md5 str)                                                    string
(sha1 str)                                                   string
(sha256 str)                                                 string
```

These primitives act on the stream of bytes of their input string, `str`, and return a string representing the bytes that the cryptographic digest they name produces, in their hexadecimal representation:

```
(md5 "a text")  ;; => "b229386ec4627869d2c71b7df3c9600a"
(sha1 "a text") ;;  => "7081f2babbafff16b4bae16282859c844baa14ef"
(sha256 "") ;; => "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
```

As shown, the returned strings use charaters in `[0-9a-f]` to represent the values of the output bytes: `md5` produces 16 bytes (for a 128 bits digest), `sha1` produces 20 bytes (160 bits) and `sha256` produces 32 bytes (256 bits).

4.6.3 Pretty Printing WhizzML Code

As shown in Subsection 4.6.1, values can be coerced to strings using `str` and `pr-str`. In addition, one can use

```
(ppr-str obj1 [width])                                       string
```

to coerce an arbitrary value to a string preserving quotations (like `pr-str`) and formatting and indenting the result as if it were WhizzML code. Unlike `pr-str`, `ppr-str` accepts only one value to print and, optionally, the line width used during formatting.

```
(ppr-str {"a" 22343 "bbbbb" 3333}) ;; => "{\"a\" 22343 \"bbbbb\" 3333}"
(ppr-str {"a" 22343 "bbbbb" 3333} 10) ;; => "{\"a\" 22343\n \"bbbbb\"\n 3333}"
(ppr-str {"a" 22343 "bbbbb" 3333} 10) ;; => "{\"a\" 22343\n \"bbbbb\" 3333}"
```

If instead of a value, what you have is a string representing WhizzML code and want to reformat it as a pretty-printed one, use `pretty-whizzml`.

```
(pretty-whizzml str-of-code [width])                         string
```

`str-of-code` must be a syntactically correct WhizzML code string, and `width` is the maximum number of characters per line in the resulting code string.

Pretty printing procedures are useful mainly for code generators (such as *reify*) and not used often when programming Machine Learning workflows.

4.6.4 String Manipulation

```
(subs str int1 [int2])                                       string
```

The `subs` procedure extracts the substring from `str` starting at the zero-based index `int1` and up to (but not including) the character at index `int2`. The latter is optional, and, if not provided, the substring takes until the end of `str`.

```
(subs "a string" 3) ;; => "tring"
(subs "a string" 0 3) ;; => "a s"
(subs "a" 2) ;; => ""
```

As you can easily check, the following expression will always evaluate to true when `n` is positive:

```
(= s (str (subs s 0 n) (subs s n)))  ;; true for all strings s
```

If `int2` is greater than the length of the string, we just take characters up to its end:

```
(subs "a string" 3 2500) ;; => "tring"
```

If `int1` or `int2` are negative, they will refer to positions starting at the end of string, i.e., they are subtracted from `(count str)`. For instance:

```
(subs "abcd" -1) ;; => "d"
(subs "abcd" -2) ;; => "cd"
(subs "abcd" 0 -1) ;; => "abc"
(subs "abcd" -1 -1) ;; => "d"
(subs "abcd" -3 2) ;; => "b"
```

```
(join list-of-strings)                                          string
(join str-sep list-of-strings)                                  string
```

The multivariadic `join` procedure concatenates a list of strings using an optional separator:

```
(join "/" ["a" "path" "x.whizzml"]) ;; => "a/path/x.whizzml"
(join "" ["1" "2"]) ;; => "12"
(join ["whizz" "ml" "!"]) ;; => "whizzml!"
```

The inverse operation, splitting a given string, is performed by the multivariadic standard procedures `split` and `split-regexp`:

```
(split str str-sep)                                         string list
(split str str-sep int)                                     string list
(split-regexp str rx-sep)                                   string list
(split-regexp str rx-sep int)                               string list
```

These procedures take a string to split (`str`) and either a literal separator (`str-sep`) or a regular expression that separators should match (`rx-sep`), and return a list of strings. The optional argument `int` specifies the maximum length of the returned list:

```
(split "a,b,c" ",") ;; => ["a" "b" "c"]
(split "a,b,c" "," 2) ;; => ["a" "b,c"]
(split "a,b,c" "," 0) ;; => []
(split "a,b,c" "," -2) ;; => []
(split "a,,b,c" ",") ;; => ["a" "" "b" "c"]
(split-regexp "a,,b,c" ",+") ;; => ["a" "b" "c"]
(split-regexp "a,,b,c" ",+" 2) ;; => ["a" "b,c"]
(split-regexp "a,,b,c" ",+" 0) ;; => []
```

The standard library also includes the following case conversion procedures:

```
(lower-case str)                                                string
(upper-case str)                                                string
(capitalize str)                                                string
```

which perform the expected conversions:

```
(lower-case "An Example") ;; => "an example"
(upper-case "An Example") ;; => "AN EXAMPLE"
(capitalize "an Example") ;; => "An example"
(capitalize "3 EXAMPLES") ;; => "3 examples"
```

Note that, as shown in the above example, `capitalize` treats its argument as a single unstructured token, upcasing only its first character.

4.6.5 String Length and Distance

```
(count str)                                                    integer
(empty?  str)                                                  boolean
```

The length of a string can be obtained by means of the polymorphic procedure `count`, which can also be applied to lists and maps. For convenience, you can use the also polymorphic predicate `empty?`, which is a shorthand for `(zero?  (count str))`.

The primitive `levenshtein` computes, as a non-negative integer, the distance between two given string values:

```
(levenshtein str1 str2)                                        integer
```

The Levenshtein distance[1] between two strings is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one into the other.

```
(levenshtein "a text" "a text") ;; => 0
(levenshtein "a text" "b text") ;; => 1
(levenshtein "a text" "another xxx") ;; => 7
```

4.6.6 Flatline Strings

WhizzML provides helpers to generate flatline s-expressions as strings (typically for use in resource creation parameters). The basic function for flatline generation is `flatline`, which constructs strings via interpolation of variables.

```
(flatline str ...)                                             string
```

The arguments to `flatline` are a list of templates, or format strings, to generate the final flatline expressions, via concatenation. Each string may refer to any WhizzML variable in scope, and it will be substituted by its value, by quoting it according to the following rules (let's call the variable to be substituted `x`):

- `{x}` to replace `x`'s value into the format string:

```
(let (w "world")
  (flatline "(hello {w})")) ;; => "(hello world)"

(define days 12)
(let (delta 2)
  (flatline "(= (+ {days} {delta}) (field \"000000\"))"))
 ;; => "(= (+ 12 2) (field \"000000\"))''
```

- `{{x}}` to replace `x` as a *quoted* value into the format string:

[1] https://en.wikipedia.org/wiki/Levenshtein_distance

```
(let (w "something blue")
  (flatline "this is {{w}}, " " right?"))
   ;; => "this is \"something blue"\, right?"
```

- `@{x}` when `x` is a list, to splice its elements into the format string:

```
(let (x [1 2 3])
  (flatline "(+ @{x})")) ;; => "(+ 1 2 3)"
```

- `@{{x}}` when `x` is a list, to splice its (recursively) quoted elements into the format string.:

```
(let (ids ["000000" "000001"])
  (flatline "(fields @{{ids}})"))
  ;; => "(fields \"000000\" \"000001\")"

(let (ids ["0" "1"]
      rows [[1 2] ["A" 3]]
      eqs (map (lambda (r) (flatline "(= fs (list @{{r}}))")) rows))
    (flatline "(let (fs (fields @{{ids}}))\n (not (or @{eqs})))"))
  ;; => "(let (fs (fields \"0\" \"1\"))
  ;;       (not (or (= fs (list 1 2)) (= fs (list \"A\" 3)))))"
```

As shown, braces have a special meaning in `flatline`'s format strings. If you need to introduce them literally, you should use a quoted variable, to avoid ambiguities and parsing errors. For example:

```
(let (ob "{"
      cb "}")
  (flatline "(if (even? x) {{ob}} {{cb}})"))
  ;; => "(if (even? x) \"{\" \"}\")"
```

Since braces are not part of Flatline's syntax, the need of quoting them will only arise, as in the above example, when they appear within string values in the resulting Flatline expression.

4.6.7 Regular Expressions

A regular expression in WhizzML is represented as a string following the Perl or Java standard notation. There is no "regular expression" type, just strings that comply to that format.

```
(regexp?  str)                                                  boolean
(re-quote str)                                  regular expression (string)
```

The `regexp?` predicate checks whether the string `str` represents a valid regular expression, and can therefore be directly used as such, and `re-quote` returns a string that matches the give string literally. Thus, `(regexp? (re-quote s))` is identically `true` for any string `s`.

```
(regexp? "a") ;; => true
(regexp? "[ab]x.") ;; => true
(regexp? "x[a") ;; => false

(re-quote "no special symbols") ;; => "no special symbols"
(re-quote "a dot: .") ;; => "\\Qa dot: .\\E"
```

To check whether a string matches a given regular expression, use the following standard library procedures:

```
(matches rx str)                                              list of string
(matches?  rx str)                                                   boolean
```

`matches` returns the list of matching groups of the regular expression `rx` found in the string str, or an empty list if no matches are found, while `matches?` checks whether the given string matches the given regular expression, i.e., whether its list of matches is not empty. Hence `(matches? r x)` is just syntactic sugar for `(not (empty? (matches r x)))`.

The list returned by `matches` always contains the original string as its first element, followed by other matching subgroups in the regular expression, if any. For instance:

```
(matches ".*x.*" "axz") ;; => ["axz"]
(matches "x([yzk]+)3" "xzzky3") ;; => ["xzzky3" "zzky"]
(matches "x(y)x([zj])" "xyxj") ;; => ["xyxj" "y" "j"])
```

Note that both `matches` and `matches?` perform full-string matching, not substring matching; e.g.:

```
(matches? "an x" "an x or two") ;; => false
(matches? "an x [a-z ]+" "an x or two") ;; => true
```

At the substring level, WhizzML provides the following replacement primitives:

```
(replace str-target rx str-repl)                                    string
(replace-first str-target rx str-repl)                              string
```

`replace` substitutes in `str-target` all (partial) matches of `rx` by the value of its third argument (another string); `replace-first` works like replace, but performing only one substitution (the first match).

```
(replace "replace me here and there" "e\\b" "X")
  ;; => "replacX mX herX and therX"
(replace-first "replace me here and there" "e\\B" "Y")
  ;; => "rYplace me here and there"
```

We provide a convenience predicate to check for occurrences of a term within a string, with a case-sensitivity flag:

```
(contains-string?  str-needle str-hay [bool-cs])                    boolean
```

The predicate checks whether the string `str-needle` occurs as a substring in `str-hay`. By default, the matching is case-sensitive, but a case-insenstive search can be requested by passing `false` as the third argument.

```
(contains-string? "foo" "bazquuxfoooo") ;; => true
(contains-string? "foo" "bazquuxFOooo" true) ;; => false
(contains-string? "foo" "bazquuxFOooo" false) ;; => true
```

Likewise, these variants of `replace` and `replace-first` take as second argument a literal string to be replaced, rather than a regular expression:

```
(replace-string str-target str-needle str-repl)                     string
(replace-first-string str-target str-needle str-repl)               string
```

For instance:

```
(replace-string "[ab] in a regexp is not '[ab]' in a string" "[ab]" ".")
  ;; => ". in a regexp is not '.' in a string"
(replace-first-string "[ab] in a regexp is not '[ab]' in a string" "[ab]" ".")
  ;; => ". in a regexp is not '[ab]' in a string"
```

4.7 Lists

List values can be expressed as literals using square brackets (as in `[1 2 3]`) or constructed via procedures such as `list` and `cons`, among others (see below). The basic accessors are `head` and `tail`, but, as we will see, it is also possible to access list elements by position, count their number, and so on and so forth.

```
(list?  obj)                                                    boolean
```

Predicate to check whether a given object is a list.

4.7.1 Constructors

```
(list obj1 ...)                                                    list
(list* obj1 ...list-or-set-args)                                   list
(cons obj list)                                                    list
(append list obj)                                                  list
(concat list1 ...)                    list    (flatten list)       list
```

The first constructor, `list`, simply constructs a list whose elements are the arguments passed in the call. To prepend `obj` to `list` and return the resulting list, use `cons`, and to append an element to the end of a list, use `append`. When you have a list `listargs` and want to prepend to it several values `obj1`..., you can use `list*`. `concat` returns the result of concatenating all its arguments, which must be lists, and `flatten`. Finally, `flatten` takes any nested combination of lists and returns their contents as a single list.

```
(list 1 "hello" true) ;; => [1 "hello" true]
(list* "hello" "hi" ["goodbye"]) ;; => ["hello" "hi" "goodbye"]
(cons 3 [2 1]) ;; => [3 2 1]
(append [1 2] 3) ;; => [1 2 3]
(concat [1 1] [] ["a"] []) ;; => [1 1 "a"]
(flatten [1 [2 3] [[[4]] [5 6]]]) ;; => [1 2 3 4 5 6]
```

Thus, when `xs` is a list, we have the equivalences

```
(list* x0 ... xn xs) => (concat (list x0) ... (list xn) xs)
                     => (cons x0 (cons x1 ... (cons xn xs)))
```

Note that `list*` also accepts as its last argument a set, in which case the set is first converted to a sequence, with unspecified order, and then the concatenation of the previous elements to the new sequence performed:

```
(list* 1 2 #[3 "a"]) ;; => [1 2 "a" 3]
(list* #[1 false 3]) ;; => [1 3 false]
(list* #[]) ;; => []
```

As shown in the examples above, when called with a single set argument, `list*` becomes WhizzML's set to list coercion procedure.[2]

It is also possible to create new lists by repeating a single element or function call.

```
(repeat int obj)                                                   list
(repeatedly int proc)                                              list
```

[2] Due to the fact that sets and lists are heterogenous, and comparisons between values of different types are not well–defined, it is not possible to provide a natural ordering when a set is transformed to a list. Thus, to avoid subtle bugs in real–world programs, most procedures taking lists as arguments will not accept a set: the transformation of sets to lists is expected to be explicit, either via `list*` or more sophisticated, user–provided, translation functions.

The `repeat` procedure constructs a new list by copying `obj` the number of times given by int, while `repeatedly` calls the procedure `proc` that number of times and constructs a list with the results of those calls.

Values for `n` that are less than zero are treated as zero, and will cause these functions to return an empty list.

```
(repeat 5 1) ;; => [1 1 1 1 1]
(repeat 3 [1 2]) ;; => [[1 2] [1 2] [1 2]]
(repeat "0" x) ;; => []
(repeatedly 5 (lambda () 1)) ;; => [1 1 1 1 1]
(repeatedly 2 rand) ;; => [0.257718843873590 23 0.2169657724443823]
```

Lists of integers can be constructed with `range`:

```
(range int [int] [int])                                                    list
```

If a single integer argument is given the result is all integers greater than or equal to 0, and less than the argument. If two arguments are given they specify the start (inclusive) and end (exclusive) of the sequence. A third argument may be passed to specify a step size.

```
(range 10) ;; => [0 1 2 3 4 5 6 7 8 9]
(range -2 4) ;; => [-2 -1 0 1 2 3]
(range 6 20 3) ;; => [6 9 12 15 18]
(range 5 -5 -2) ;; [5 3 1 -1 -3]
```

4.7.2 Accessors

```
(head list [obj])                                                        object
(tail list)                                                                list
(last list [obj])                                                        object
(butlast list)                                                             list
```

To access the first element of a list, use `head`. To get all the elements of a list but its head, ask for its `tail`. The dual of those operations are `last`, which returns the last element of a given list, and `butlast`, which returns a list containing all elements of a given list but the last one. All these standard procedures expect as argument a non-empty list, and will signal error code -20 if passed an empty list and, in the case of `head` and `last`, no default value. Both `head` and `last` accept an optional argument `obj` that is returned when list is empty instead of throwing an error.

```
(head ["a" "b" "c"]) ;; => "a"
(head [] 42) ;; => 42
(tail ["a" "b" "c"]) ;; => ["b" "c"]
(cons (head ["a" "b" "c"]) (tail ["a" "b" "c"])) ;; => ["a" "b" "c"]
(last [1 2]) ;; => 2
(last [] "a") ;; => "a"
(butlast [1 2]) ;; => [1]
(butlast [true]) ;; => []
```

We have the identities:

```
(= lst (cons (head lst) (tail lst))) ;; => true for all lists lst
(= lst (append (butlast lst) (last lst))) ;; => true for all lists lst
```

```
(nth list int [obj])                                                     object
```

Indexed access to the elements of a list is provided by `nth`, with an optional value to return for out-of-bounds indexes. The index is zero-based: the head of a list is accessed via `(nth lst 0)`. `nth` raises error code -30 if `int` is equal or greater than the length of `list` and no default value (`obj`) is provided.

As discussed in Section 2.5, explicit calls to `nth` are rarely needed, because one can directly apply list values to an index to obtain their elements:

```
(["a" "b" "c"] 0) ;; => "a"
(["a" "b" "c"] 3 "d") ;; => "d"
(let (l [1 2 3])
  [(l 2) (l 0) (l 1) (l 12 42)]) ;; => [3 1 2 42]
```

See also Subsection 2.7.1 for a way to access list elements via sequential destructuring in `let` forms.

```
(insert list int obj)                                                    list
```

The `insert` procedure inserts a given element at its `int`-th position. Note that lists, as all WhizzML values, are immutable: `insert` constructs and returns a new list, leaving its arguments untouched.

The insertion index `int` is zero-based, so that `(insert lst 0 x)` is the same as `(cons x lst)` and `(insert lst (count lst) x)` is equivalent to `(append lst x)`.

If the insertion position is greater than the list length, the element is just appended to the end of the list.

```
(insert [1 2 3] 1 "a") ;; => [1 "a" 2 3]
(insert [1 2 3] 5 "t") ;; => [1 2 3 5 "t"]
```

```
(take int list)                                                          list
(drop int list)                                                          list
```

These procedures return new lists obtained by either taking or droping the first (if `int` is positive) or last (if `int` is negative) `int` elements of `list`. Both `take` and `drop` accept `int` larger (in absolute value) than the size of the collection, in which case they return either the entire collection or an empty list.

```
(take 4 [1 2 3 4 5]) ;; => [1 2 3 4]
(take -2 [1 2 3 4 5]) ;; => [4 5]
(drop 3 [1 2 3 4 5]) ;; => [4 5]
(drop -3 [1 2 3 4 5]) ;; => [1 2]
(take 4 [1 2]) ;; => [1 2]
(drop 20 [1 2]) ;; => []
(drop -3 [1 2]) ;; => []
(take -10 [1 2 3]) ;; => [1 2 3]
```

We have the trivial equivalences:

```
n < 0 => (take n l) == (drop (+ (count l) n) l)
n < 0 => (drop n l) == (take (+ (count l) n) l)
```

4.7.3 Membership

```
(member?  obj list-or-set)                                               boolean
```

The standard procedure `member?` performs a linear search of `obj`, using structural equality (`=`), over `list-or-set`, which can be either a list or a set.

```
(member? 3 [1 2 8 4 3 2]) ;; => true
(member? {"a" 2} ["foo" {"a" 2}]) ;; => true
(member? "foo" []) ;; => false
```

```
(remove-duplicates list)                                                    list
```

The `remove-duplicates` standard procedure ensures all elements in the returned list are distinct, preserving order. The first occurence of each unique element is kept.

```
(remove-duplicates [1 2 1 1 3 2]) ;; => [1 2 3]
(remove-duplicates [1 2 false]) ;; => [1 2 false]
```

```
(some proc list)                                                            object
```

The `some` procedure takes a procedure of one argument and a list as its arguments. It applies the input function to the list elements in order, returning the result of the application if it is anything other than `false`. If no element in the list causes the input function to evaluate to something other than false, `some` returns false.

```
(some odd? [2 4 6 7 8]) ;; => true
(some (lambda (n) (if (odd? n) n false)) [2 4 6 7 8]) ;; => 7
(some (lambda (n) (if (odd? n) n false)) [2 4 6 8]) ;; => false
```

4.7.4 Length

```
(count list)                                                              integer
(empty?  list)                                                            boolean
```

```
(count {"a" 2 "b" 3}) ;; => 2
(count [18 -23]) ;; => 2
(count "whizzml") ;; => 6
(empty? {}) ;; => true
(empty? "") ;; => true
(empty? []) ;; => true
(empty? [[]]) ;; => false
```

4.7.5 Extrema Finding

```
(min-key proc list)                                                        object
(max-key proc list)                                                        object
```

The `min-key` (`max-key`) method returns the element `x` in the input list for which the value of `(proc x)` is less than (greater than) or equal to the value of `(proc y)` for any element `y` in the input list. This can be useful for, e.g., getting the largest value for a key in a list of maps.

```
(define lst [{"a" 2 "b" 9} {"a" 9 "b" 3}])
(min-key (lambda (x) (get x "a")) lst) ;; => {"a" 2 "b" 9}
(max-key (lambda (x) (get x "b")) lst) ;; => {"a" 2 "b" 9}
```

4.7.6 Sorting and Reordering

```
(reverse list)                                                              list
```

This standard procedure returns a new list with the same elements as `list`, but in reverse order.

```
(sort list)                                                                 list
```

Lists of numbers, strings, or lists can be sorted by `sort`, but in general the lists (and the lists of lists) must be homogenous or an exception will be thrown.

```
(sort [2 3 1]) ;; => [1 2 3]
(sort [[1] [0 0] [0]]) ;; => [[0] [1] [0 0]]
```

`(sort-by-key str-key list-of-maps)` `list`

Using `sort-by-key`, one can sort a list of maps by a specfic key in the given maps. Note that primative values and maps not containing the specified key are equivalent under the specified ordering and come *before* all other values in the returned list.

```
(sort-by-key "a" [{"a" 2} {"a" 3} {"a" 1}]
  ;; => [{"a" 1} {"a" 2} {"a" 3}])
(sort-by-key "a" [1 {"a" 3} {"b" 1} {"a" 1}]
  ;; => [1 {"b" 1} {"a" 1} {"a" 3}])
```

4.7.7 Folding with reduce and iterate

`(reduce proc obj list)` `object`

The `reduce` procedure is the familiar left fold[3] for lists, which can be defined in pure WhizzML as:

```
(define (reduce fn init lst)
  (loop (l lst r init)
    (if (empty? l) r (recur (tail l) (fn r (head l))))))
```

So `reduce` applies its first argument, a function of two arguments, to its second argument and the first element of the list given as third argument. Then it applies the function to the result of that call and the second element of the list, and so on repeteadly until the list is exhausted; i.e., `reduce` follows the following reduction pattern:

```
(reduce fn init lst) => (reduce (fn init (head lst)) (tail lst))
```

Iteration over a list with an accumulator is so frequent that WhizzML provides also a syntactic form, `iterate` , that makes writing folding expressions over one or more lists easier. Its syntax is as follows:

```
(iterate (<acc> <init-value> <var0> <list0> <var1> <list1>...)
  <body>)
```

Here, `<acc>` is a variable name, the accumulator, which takes the initial value `<init-value>` , and `<var1>` . . . are variable names that take, in order, the values of each of the elements of `<list0>` , `<list1>` . . . , which must be all expressions evaluating to lists. These variable names can be used in `<body>` , and the result of evaluating it for each set of their values is assigned again to `<acc>` for the next iteration. In other words, the above `iterate` skeleton is approximately equivalent to the following loop:

```
(loop (<acc> <init-value>
       vs0 <list0>
       vs1 <list1>
       ...)
  (if (some empty? vs0 vs1 ...)
      <acc>
```

[3] https://en.wikipedia.org/wiki/Fold_(higher-order_function)

```
      (let (<var0> (head vs0)
            <var1> (head vs1)
            ...)
        (recur (prog <body>) (tail vs0) (tail vs1) ...))))
```

where `vs0`, `vs1`... are local variable names that are not free in `<body>`. So, for instance:

```
(iterate (a 0 x (range 4) y [1 8 9 3]) (+ a x y))
```

is equivalent to the loop:

```
(loop (a 0
       xs (range 4)
       ys [1 8 9 3])
  (if (empty? xs)
      a
      (let (x (head xs)
            y (head ys))
        (recur (+ a x y) (tail xs) (tail ys)))))
```

We can also rewrite a single-list `iterate` as a `reduce` call:

```
(reduce (lambda (<acc> <var0>)
          <body>)
        <init-value>
        <list0>)
```

So, for single lists, `iterate` lets you write your reductions in a more compact form, specially when `<<body>` is not trivial. But `iterate` is more powerful than `reduce` in that it can traverse (in parallel) more than one list, and also allows early exits of the iteration with `break`.

Indeed, the reason why the above loop or reduce forms are not exactly equivalent to the corresponding `iterate` form is that in the body of `iterate` you can use the reserved form `break` to exit early from the iteration; `break` takes a single argument which will be the value of the full `iterate` expression. For instance:

```
(iterate (a 0 x [1 20 -1]) (+ a x)) ;; => 20

(iterate (a 0 x [1 20 -1])
  (if (negative? x)
      (break 0)
      (+ a x)))  ;; => 0
```

4.7.8 Filtering

```
(filter proc list)                                                    list
```

The `filter` procedure returns a filtered version of the input list, where the filtered version includes only items `x` for which `(proc x)` does *not* evaluate to false.

```
(filter (lambda (x) (> x 5)) [10 5 4 3 10 7]) ;; => [10 10 7]
(filter (lambda (x) (> x 10)) [10 5 4 3 10 7]) ;; => []
```

4.7.9 Other List Traversal Procedures

```
(every?  proc list)                                                    bool
```

The `every?` standard predicate traverses the given `list`, applying to every element the predicate `proc`, and returning `true` is all applications evaluate to a non-false value:

```
(every? odd? [1 3 5 7]) ;; => true
(every? odd? []) ;; => true
(every? odd? [1 2 3 4 5]) ;; => false
```

4.8 Sets

WhizzML has a *set* datatype, representing a collection of unique, unordered values. WhizzML's set is implemented as a hash table, and has the associated performance characteristics, including extremely fast insertion and retrieval. The corresponding type predicate is `set?`:

```
(set?  obj)                                                          boolean
```

4.8.1 Construction

```
(set obj1 ...)                                                         set
(set* obj1 ...list-or-set-args)                                        set
```

The first constructor, `set`, simply constructs a set whose elements are the arguments passed in the call, eliminating any duplicates. When you have a list or a set `list-or-set-args` and want to add to it several values `obj1`..., you can use `set*`.

```
(set 2 1 3) ;; => #[1 2 3]
(set [1 2] 3) ;; => #[3 [1 2]]
(set* [2 1 3]) ;; => #[1 2 3]
(set* 3 ["a" 2]) ;; => #["a" 3 2]
(set* 3 ["a" 2 3]) ;; => #["a" 3 2]
(set* "a" [1 2] #[true "b" "a"]) ;; => #[true "a" "b" [1 2]]
(list* (set* 1 2 #[3 4])) ;; => [1 4 3 2]
```

As shown in the examples above, when called with a single list argument, `set*` becomes WhizzML's set to list coercion procedure, the dual of `list*`, but not its inverse because sets preserve neither order nor duplicates:

```
(list* (set* [2 3 "a"])) ;; => ["a" 3 2]
(list* (set* [3 2 3 "a"])) ;; => ["a" 3 2]
```

4.8.2 Membership

```
(add obj set)                                                          set
(remove obj set)                                                       set
```

Adding and removing single elements to and from sets is accomplished via the standard procedures `add` and `remove`, while to check whether a given value belongs to a set one can use the overloaded standard procedure `member?`:

```
(add 2 #[]) ;; => #[2]
(add 2 #[1 2]) ;; => #[1 2]
(remove 1 #[1 3]) ;; => #[3]
```

```
(remove 1 #["1" 3]) ;; => #["1" 3]
(member? "a" #["a" 2]) ;; => true
(member? 3 #["a" 2]) ;; => false
```

It is also possible to check for subset relationships using the following predicates:

```
(subset?  set-0 set-1 [strict?])                                    boolean
(superset?  set-0 set-1 [strict?])                                  boolean
```

The standard predicate `subset?` (resp. `superset?`) checks whether its first argument is a subset (resp. superset) of its second one, i.e., whether the second (resp. first) set contains all the elements in the first (resp. second) one, e.g.:

```
(subset? #[1 true] #[true 2 false 1]) ;; => true
(subset? #[1 true] #[true 2 false]) ;; => false
(subset? a-set a-set) ;; => true
(subset? #[] a-set) ;; => true
(superset? #[true 2 false 1] #[1 true]) ;; => true
(superset? #[true 2 false] #[1 true]) ;; => false
(superset? a-set a-set) ;; => true
(superset? #[] a-set) ;; => false
```

As shown in the above examples, inclusion checks are non-strict by default, i.e., a set is always a superset and a subset of itself. To check whether the relationship is strict (that is, whether the two subsets are strictly different), set the optional argument `strict?` to `true`:

```
(subset? a-set a-set) ;; => true
(superset? a-set a-set) ;; => true
(subset? a-set a-set false) ;; => true
(superset? a-set a-set false) ;; => true
(subset? a-set a-set true) ;; => false
(superset? a-set a-set true) ;; => false
```

4.8.3 Set Operations

The standard set–theoretical combiners are available for WhizzML sets via the following standard procedures:

```
(union set-0 set-1)                                                 set
(intersection set-0 set-1)                                          set
(difference set-0 set-1)                                            set
```

`union` adds all elements of the first set to the second one, `intersection` returns all elements in both sets and `difference` removes from list-or-set-0 all elements in list-or-set-1. For instance:

```
(union #[1 2] #[false 1 4]) ;; => #[false 1 4 2]
(union #[] a-set) ;; => a-set
(intersection #[1 2 "c"] #["c" 2 "b"]) ;; => #[2 "c"]
(intersection a-set #[]) ;; => #[]
(difference #[1 2 3] #[3 1]) ;; => #[2]
(difference #[1 2 3] #["a" false true]) ;; => #[1 2 3]
(difference a-set a-set) ;; => #[]
(difference a-set #[]) ;; => a-set
```

4.9 Maps

4.9.1 Construction

```
(make-map list-of-keys list-of-values)                                    map
```

`make-map` is the basic map constructor in WhizzML. This procedure takes two lists of equal length, and keys and values match by position.

```
(let (v0 3 v1 2) (make-map ["k0" "k1"] [v0 v1])) ;; => {"k0" 3 "k1" 2}
(let (ks ["k1" "k0"]) (make-map ks [v0 v1])) ;; => {"k1" 3 "k0" 2}
```

The list of keys and values of a map can be recovered via the procedures:

```
(keys map)                                                                list
(values map)                                                              list
```

So we have the following identities, for any lists `ks` and `vs`:

```
(= (keys (make-map ks vs)) ks)   ;; => true
(= (values (make-map ks vs)) vs) ;; => true
```

It is also possible to construct a new map using a subset of the keys and values in another one, via `select-keys`:

```
(select-keys map list-of-strigs)                                          map
```

Non-existing keys are ignored. Thus, for instance:

```
(select-keys {"a" 2 "b" 12 "c" [1 2]} ["a" "b"]) ;; => {"a" 2 "b" 12}
(select-keys {"a" 2 "b" 12 "c" [1 2]} ["c" "x"]) ;; => {"c" [1 2]}
(select-keys {"a" 2 "b" 12 "c" [1 2]} ["d" "x"]) ;; => {}
(select-keys {"a" 2 "b" 12 "c" [1 2]} []) ;; => {}
```

4.9.2 Accessors

Access to values in a map is provided by the map itself used as a function (see Section 2.5), `get` and `get-in`, and `contains?` provides a membership test:

```
(contains?  map str-key)                                                boolean
(get map str-key [obj])                                                  object
(get-in map list-of-keys-or-ints [obj])                                  object
```

To access the value associated with a key `str-key` in a map, we use `get`, which takes an optional third argument with the value to return in case the key is not found in `map`. `get-in` performs a lookup following nested maps, given a list of keys and list positions (for cases where the corresponding value in the path is a list) and, optionally, a default value to return if the element is not found.

```
(get {"a" 42 "b" 3} "a") ;; => 42
(get {"a" 42} "c" 21) ;; => 21
(get-in {"a" {"b" 34}} ["a" "b"]) ;; => 34
(get-in {"a" [1 2 {"b" [3 4]}]} ["a" 2 "b" 0]) ;; => 3
(contains? {"a" 42} "a") ;; => true
(contains? {"c" {"a" 3}}) ;; => false
```

However, it is more common to simply apply the map value as a procedure to the key or list of keys we are looking up:

```
({"a" 42 "b" 3} "a") ;; => 42
({"a" 42} "c" 21) ;; => 21
({"a" {"b" 34}} ["a" "b"]) ;; => 34
({"a" [1 2 {"b" [3 4]}]} ["a" 2 "b" 0]) ;; => 3
```

Applicability of maps thus makes explicit use of the `get` and `get-in` accessors unnecessary in the vast majority of cases.[4]

In order to perform lookups using a path that starts with a list value instead of a map, `get-in` is actually overloaded and can take a list as its first argument, making the following lookups valid:

```
([{"b" 2 "a" 22}] [0 "a"]) ;; => 22
([1 [2 [3 [4]]]] [1 1 1 0]) ;; => 4
```

To avoid undefined values, asking for a key (or key list) that is not contained in a map without providing a default value raises an error with code -15 (key not found).

4.9.3 Element Insertion

As all WhizzML values, maps are immutable, but you can create new ones by adding values to an existing one, using `assoc`, `assoc-in` and `merge`:

```
(assoc map str-key1 obj1 ...)                                    map
(assoc-in map list-of-keys obj)                                  map
(merge map1 map2)                                                map
```

To add key/value pairs to a given `map` you can use `assoc` or `assoc-in` for nested keys. Besides the map, `assoc` takes an arbitrary number of alternating keys and values, and will raise a bad arity error (code -40) if passed an even number of arguments. `merge` adds to `map1` all keys in `map2`, overriding any one already in the former.

```
(assoc {} "key" 42) ;; => {"key" 42}
(assoc {"age" 23} "name" "Johnny") ;; => {"age" 23 "name" "Johnny"}
(assoc {} "key0" {} "key1" [] "key3" 42)
  ;; => {"key0" {} "key1" [] "key3" 42}
(assoc-in {"foo" 42 "submap" {"k" 3}} ["submap" "m"] 23)
  ;; => {"foo" 42 "submap" {"k" 3 "m" 23}}
(merge {"a" 2 "b" 3} {"a" 8 "c" 5}) ;; => {"a" 8 "b" 3 "c" 5}
```

4.9.4 Element Removal

One can remove elements from an existing map with `dissoc` and `dissoc-in`, which, again, won't mutate their map arguments, but, rather, return a new map:

```
(dissoc map str1 ...)                                            map
(dissoc-in map list-of-keys)                                     map
```

```
(dissoc {"a" 1 "b" 2 "c" 3} "a") ;; => {"b" 2 "c" 3}
(dissoc {"a" 1 "b" 2 "c" 3} "a" "c") ;; => {"b" 2}
(dissoc-in {"a" {"b" 3 "c" {"d" 5}}} ["a" "c" "d"])
  ;; => {"a" {"b" 3 "c" {}}}
```

[4] One can concoct scenarios in which the accessor functions are needed because, for instance, they are used (or stored) indirectly as first class values, but those situations arise very rarely in practice.

4.10 Time

Time manipulation procedures are currently limited to the primitive `current-time`, which returns the current time as the number of milliseconds since Jan 1st, 1970 (that is, the Unix epoch).

```
(current-time )                                                    integer
```

```
(current-time) ;; => 1494285448247
```

The standard library also includes the procedure `sleep`:

```
(current-time intmsecs)                                            integer
```

which takes an integer number of milliseconds and waits (up to 16 seconds), returning the actual number of milliseconds spent waiting: zero if `intmsecs` is negative, or its argument up to the maximum sleep period (16000 milliseconds).

```
(sleep -3) ;; => 0 (no wait)
(sleep 12345) ;; => 12345
(sleep 10000000) ;; => 16000
```

4.11 Logging

There's no interactive input in WhizzML, but you can output logs using the procedures enumerated below. All of them take any number of arguments, coerce them to strings and ouput the result of concatenating those strings, prepended by a the log level label, when the current log level is equal or greater to the requested one.

```
(log-debug obj1 ...)                                               false
(log-info obj1 ...)                                                false
(log-warn obj1 ...)                                                false
(log-error obj1 ...)                                               false
```

For instance, the statement

```
(let (x 3) (log-info "The x value is " x))
```

will log the message

```
INFO: The x value is 3
```

when you run your script locally. When WhizzML scripts are run on the server side, logs are collected in a map of the execution metadata, as lists of strings keyed by log level.

As mentioned, logs are only output if the requested log level is not below the current minimum log level. By default, the minimum log level is set to "info". You can query and alter it using these two standard procedures:

```
(log-level )                                                       integer
(set-log-level int)                                                integer
```

There is also a special form that will automatically log the running time for any group of expressions:

```
(with-time-log body)                                                object
```

This form will evaluate its `body` normally, but will also compute the time the evaluation took, and write a log with that information. Since it evaluates to its body's value, the `with-time-log` expression can be used in any place its body would be used. For instance:

```
(+ (with-time-log (+ 1 2 3)) 8) ;; => 14
```

Evaluating the above expression will also give you a log similar to:

```
INFO: Elapsed time: 1
```

Finally, there's a special log function to report progress in script executions, and an associated accessor:

```
(log-progress num)                                                    number
(logged-progress )                                                    number
```

The `log-progress` primitive will update the progress reported by your WhizzML execution (when run in BigML's servers) to the given value, after coercing it to the unit interval `[0, 1]`. It evaluates to the actual value stored as the current progress, which can also be retrieved at any later moment using `logged-progress`:

```
(log-progress 0.12) ;; => 0.12
(logged-progress) ;; => 0.12
(log-progress 0.9) ;; => 0.9
(log-progress 0.7) ;; => 0.7
(logged-progress) ;; => 0.7
(log-progress 23) ;; => 1.0
(logged-progress) ;; => 1.0
```

4.12 BigML Resources

The WhizzML standard library provides procedures to **list**, **create**, **fetch**, **update** and **delete** BigML resources. There are thus five generic functions that work for any resource type, as well as specialized versions of the listing and creation calls, where the resource type is implicit.

4.12.1 Resource Types

In `resources` and the `list` and `create` families of calls (see below), the resource type can be any of the following supported types:

- source
- dataset
- model
- composite
- fusion
- optiml
- ensemble
- prediction
- batchprediction
- evaluation
- anomaly
- anomalyscore
- batchanomalyscore
- cluster
- centroid

- batchcentroid
- association
- associationset
- linearregression
- logisticregression
- correlation
- statisticaltest
- topicmodel
- topicdistribution
- batchtopicdistribution
- deepnet
- timeseries
- forecast
- pca
- projection
- batchprojection
- project
- configuration
- sample
- library
- script
- execution

4.12.2 Resource Identifiers

We provide a few methods for getting common information from resources and their identifiers:

```
(resource-id?  obj)                                                    boolean
(parse-resource-id res)                                                  list
(resource-types )                                                        list
(resource-type res)                                                    string
(resource-id res)                                                  resource-id
```

To check whether obj is a well-formed resource identifier, use the `resource-id?` predicate. Resouce identifiers are of the form `type/bare-id`, and `parse-resource-id` returns a list of the two components of the identifier, if it is well-formed (or the empty list otherwise). One can ask for a list of all available resource types using the standard procedure `resource-types`, which takes no arguments and returns a sorted list of strings. Finally, given either a resource map or its identifier, the primitives `resource-type` and `resource-id` extract from it the corresponding resource type or resource identifier. In both cases, passing as argument a value that is not either a resource map or a resource identifier produces the empty string as the result of the call.

```
(resource-id? 3) ;; => false
(resource-id? "source/121212321231231231231231") ;; => true
(parse-resource-id "source/121212321231231231231231")
  ;; => ["source" "121212321231231231231231"]
(parse-resource-id "not-a-resource-id") ;; => []
(resource-id "source/121212321231231231231231")
```

```
  ;; => "source/121212321231231231231231"
  (resource-id (fetch "source/121212321231231231231231"))
  ;; => "source/121212321231231231231231"
(resource-id "nosource/121212321231231231231231") ;; => ""
(resource-id 3) ;; => ""
(resource-id {"foo" 3}) ;; => ""
(resource-type "source/121212321231231231231231") ;; => "source"
(resource-type (fetch "source/121212321231231231231231")) ;; => "source"
(resource-type "nosource/121212321231231231231231") ;; => ""
(resource-type 3) ;; => ""
(resource-types) ;; => ["anomaly" "anomalyscore" ... "topicmodel"]
```

4.12.3 Resource Properties

In order to obtain a resource property given its identifier, one needs to `fetch` it first and then access the resulting map. This is such a common operation that the standard library provides a helper, optimizing to fetching minimal information:

```
(resource-property res-or-id path [default])                    any
```

This procedure takes either a resource identifier or a full resource map (such as the ones returned by `fetch`) and extracts from it it the procedure indicated by the string or list of strings `path`. If the property is not found, `resource-property` will throw an error unless a default value (the last, optional argument) has been provided. So, when `id` is the identifier of a resource, the call

```
(resource-property id path default)
```

is loosely equivalent to

```
((fetch id) path default)
```

with the difference that the fetch performed by `resource-property` sets query string parameters minimizing bandwidth. The name of a resource is requested so often that we provided a trivial specialization:

```
(resource-name res-or-id [default])                             string
```

Similarly, syntax sugar is provided to inform about whether a source is open for editing.

```
(source-open?  res-or-id)                                       boolean
```

See also Subsection 4.12.13 for retrieval of the the "fields" property.

4.12.4 Resource Children

In order to model, predict or evaluate, resources are created based on other previously existing resources. For instance, to build a model first you need a source to contain your uploaded data and then a dataset to summarize the information in the source. The dataset becomes the origin for the final model. Therefore, many resources can stem from a given one in a tree-like generation chain. To obtain the structure of those children, we provide the `resource-children` procedure:

```
(resource-children res-id)                                      list
```

which will produce a list of lists storing the *parent to children* relations. As an example, if you build an optiml from a dataset built from an existing source and ask for the source children, the result will be:

```
(resource-children "source/121212321231231231231231") ;; =>
```

```
["source/12121232123123123123123"
 ["dataset/5f921b062275c111e903394d" ["optiml/5f921b08ff0f14111001083a"]]]
```

4.12.5 Error Reporting

All resource-related procedures can raise exceptions when the BigML API services report an error in fulfilling the request. These errors are reported by WhizzML by raising, as usual, a map that always has as keys "message" and "code", with the latter being always the error code -50. In addtion, full details of the error reported by the API, including its code as listed in the BigML API documentation[5] and associated HTTP status and extra information, are reported as a map under the key "cause". Here's an example of the error map for a malformed source creation request:

```
{"code" -50
 "message" "Error computing primitive operation 'create': Bad request: "
 "instruction" {"source" {"lines" [1 1]
                "columns" [0 34]}
                "instruction" "apply"}
 "cause" {"code" -1204
          "http_status" 400
          "extra" ["Data or remote arguments are missing"]}}
```

Errors during resource handling are treated uniformly using exceptions. That means that whenever you try to use a resource whose status is not either in-progress or finished, the primitives will raise an error and, unless you are using an error handler (see Section 2.10), execution of your program will stop.

4.12.6 Listing Resources

`(resources str-type [map-of-options])` `list of map`

`resources` asks the BigML service for a list of available resources belonging to the caller, returned in the form of a list of maps representing resource metadata.

The first argument of `resources` is a string naming the type of the resources to be listed and the optional `map-of-options` is a map containing the key/values that you would use in the BigML's API query string to filter the returned list the corresponding resource type (that is, essentially, the list that you obtain in JSON via the API as the "objects" key value).

For instance, you can paginate over all of your sources with a snippet of the form:

```
(define (process-source src) ....)

(loop (offset 0)
 (let (srcs (resources "sources" {"offset" offset "limit" 10}))
   (map process-source srcs)
   (when (not (empty? srcs))
     (recur (+ offset (count srcs))))))
```

Although it is rather trivial to extract a list of identifiers from a list of resource maps, we define it as part of the standard library:

`(resource-ids list-of-maps)` `list of string`

`resource-ids` is implemented in pure WhizzML as a variation of

[5] https://bigml.com/developers/status_codes

```
(define (resource-ids resources)
  (map resource-id resources))
```

that incorporates a bit more error checking.

For convenience, we define a list function for each resource type:

```
(list-sources [maparg-of-options])                                    list of map
(list-datasets [maparg-of-options])                                   list of map
(list-models [maparg-of-options])                                     list of map
(list-composites [maparg-of-options])                                 list of map
(list-fusions [maparg-of-options])                                    list of map
(list-optimls [maparg-of-options])                                    list of map
(list-ensembles [maparg-of-options])                                  list of map
(list-predictions [maparg-of-options])                                list of map
(list-batchpredictions [maparg-of-options])                           list of map
(list-evaluations [maparg-of-options])                                list of map
(list-anomalies [maparg-of-options])                                  list of map
(list-anomalyscores [maparg-of-options])                              list of map
(list-batchanomalyscores [maparg-of-options])                         list of map
(list-clusters [maparg-of-options])                                   list of map
(list-centroids [maparg-of-options])                                  list of map
(list-batchcentroids [maparg-of-options])                             list of map
(list-associations [maparg-of-options])                               list of map
(list-associationsets [maparg-of-options])                            list of map
(list-linearregressions [maparg-of-options])                          list of map
(list-logisticregressions [maparg-of-options])                        list of map
(list-correlations [maparg-of-options])                               list of map
(list-statisticaltests [maparg-of-options])                           list of map
(list-topicmodels [maparg-of-options])                                list of map
(list-topicdistributions [maparg-of-options])                         list of map
(list-batchtopicdistributions [maparg-of-options])                    list of map
(list-deepnets [maparg-of-options])                                   list of map
(list-timeseriess [maparg-of-options])                                list of map
(list-forecasts [maparg-of-options])                                  list of map
(list-pcas [maparg-of-options])                                       list of map
(list-projections [maparg-of-options])                                list of map
(list-batchprojections [maparg-of-options])                           list of map
(list-projects [maparg-of-options])                                   list of map
(list-configurations [maparg-of-options])                             list of map
(list-samples [maparg-of-options])                                    list of map
(list-libraries [maparg-of-options])                                  list of map
(list-scripts [maparg-of-options])                                    list of map
(list-executions [maparg-of-options])                                list of map
```

4.12.7 Creating Resources

```
(create str-type map-of-options)                                      resource-id
(create str-type res-parent [map-of-options])                         resource-id
(create str-type res-parent res-parent-2 [map-of-options])            resource-id
```

In `create` calls, `str-type` can be any of the supported resource types listed in Subsection 4.12.1 and the options map accepts the same keys and values as the JSON body of an API call to create the respective resource. For instance, a call to create a remote source could be as simple as:

```
(create "source" {"remote" "https://static.bigml.com/csv/iris.csv"})
```

while for a source named "test source" with a brief description and explicit source parser we would write:

```
(create "source" {"remote" "https://static.bigml.com/csv/iris.csv"
                  "name" "test source"
                  "description" "powered by whizzml"
                  "source_parser" {"separator" ";"
                                   "header" false}})
```

`create` just launches resource creation, and doesn't wait for its completion. It returns the new resource identifier, as a string. Typically, you will associate that identifier to a variable for later use:

```
(define src-id (create "source" {"remote" "s3://bucket.com/data.csv"}))
(define ds-id (create "dataset" {"source" src-id}))
```

There are two other forms of `create` taking, in addition to an optional options map, either one or two resource identifiers which will be the *parent* or origin of the newly created resource. For instance, the parent of a model is a dataset, the parent of a dataset can be either a source or another dataset (when creating subsamplings or filtering), and the parents of an evaluation or batchprediction are a dataset and a model. Table 4.1 shows the full list of possible parent resources (resources not appearing in the table don't have any valid parent). Some examples:

```
(create "dataset" "source/121212321231231231231233")
(create "dataset" "dataset/a212b23212312312312312 33" {"sample_rate" 0.7})
(create "evaluation"
        "dataset/a212b2321231231231231233"
        "model/562fe2d8636e1c5ec500688c")
(create "batchcentroid"
        "dataset/a212b2321231231231231233"
        "cluster/562fe2d8636e1c5ec500688c"
        {"name" "test"})
```

Table 4.1: Possible parent resources in calls to `create`

Created resource	Parent resource	Second parent resource
dataset	dataset, source	
sample	dataset	
model	dataset	
ensemble	dataset	
optiml	dataset	
linearregression	dataset	
logisticregression	dataset	
timeseries	dataset	
forecast	timeseries	
deepnet	dataset	
pca	dataset	
batchprojection	pca	dataset
projection	pca	
prediction	fusion, model, deepnet, timeseries, ensemble, logisticregression, linearregression	

Table 4.1: Possible parent resources in calls to `create`

Created resource	Parent resource	Second parent resource
evaluation	fusion, model, deepnet, timeseries, ensemble, logisticregression, linearregression	dataset
batchprediction	fusion, model, deepnet, ensemble, logisticregression, linearregression	dataset
cluster	dataset	
centroid	cluster	
batchcentroid	cluster	dataset
anomaly	dataset	
anomalyscore	anomaly	
batchanomalyscore	anomaly	dataset
association	dataset	
associationset	association	
statisticaltest	dataset	
correlation	dataset	
topicmodel	dataset	
topicdistribution	topicmodel	
batchtopicdistribution	topicmodel	dataset
execution	script	

For convenience, the standard library offers a method, `create*`, which will create a list of resources in parallel, without waiting for completion:

```
(create* list-of-types list-of-options)                    list of resource-id
```

For example:

```
(create* ["source" "source"]
         [{"remote" "http://url/1"} {"remote" "http://url/2"}])
```

If all resources to be created are of the same type, you may pass a single string for the first parameter, which will be duplicated implicitly. Thus, the following is equivalent to the above call:

```
(create* "source" [{"remote" "http://url/1"} {"remote" "http://url/2"}])
```

Instead of providing a map of options, you can also use a parent resource when it's unique, and mix them as needed with options, as in the following examples:

```
(create* ["source" "dataset" "model"]
         [{"remote" "http://static.bigml.com/csv/iris.csv"}
          "source/121212321231231231231233"
          "dataset/a212b23212312312312312333"])
```

The standard library also provides convenience procedures for creation of specific resource types, for each of the ways the basic `create` primitive can be invoked. Thus, there is a collection of procedures for creating resources given either a single options map, a parent resource identifier and an optional options

map, and (for those resources created from two parents, see Table 4.1) two parent resource identifiers plus an optional options map:

```
(create-source [res-id res-id-2 map-of-options])                          source-id
(create-dataset [res-id res-id-2 map-of-options])                        dataset-id
(create-model [res-id res-id-2 map-of-options])                            model-id
(create-composite [res-id res-id-2 map-of-options])                    composite-id
(create-fusion [res-id res-id-2 map-of-options])                          fusion-id
(create-optiml [res-id res-id-2 map-of-options])                          optiml-id
(create-ensemble [res-id res-id-2 map-of-options])                      ensemble-id
(create-prediction [res-id res-id-2 map-of-options])                  prediction-id
(create-batchprediction [res-id res-id-2 map-of-options])          batchprediction-id
(create-evaluation [res-id res-id-2 map-of-options])                    evaluation-id
(create-anomaly [res-id res-id-2 map-of-options])                        anomaly-id
(create-anomalyscore [res-id res-id-2 map-of-options])                anomalyscore-id
(create-batchanomalyscore [res-id res-id-2 map-of-options])      batchanomalyscore-id
(create-cluster [res-id res-id-2 map-of-options])                        cluster-id
(create-centroid [res-id res-id-2 map-of-options])                      centroid-id
(create-batchcentroid [res-id res-id-2 map-of-options])              batchcentroid-id
(create-association [res-id res-id-2 map-of-options])                  association-id
(create-associationset [res-id res-id-2 map-of-options])            associationset-id
(create-linearregression [res-id res-id-2 map-of-options])        linearregression-id
(create-logisticregression [res-id res-id-2 map-of-options])    logisticregression-id
(create-correlation [res-id res-id-2 map-of-options])                  correlation-id
(create-statisticaltest [res-id res-id-2 map-of-options])          statisticaltest-id
(create-topicmodel [res-id res-id-2 map-of-options])                    topicmodel-id
(create-topicdistribution [res-id res-id-2 map-of-options])      topicdistribution-id
(create-batchtopicdistribution [res-id res-id-2 map-of-options])
batchtopicdistribution-id
(create-deepnet [res-id res-id-2 map-of-options])                          deepnet-id
(create-timeseries [res-id res-id-2 map-of-options])                    timeseries-id
(create-forecast [res-id res-id-2 map-of-options])                        forecast-id
(create-pca [res-id res-id-2 map-of-options])                                  pca-id
(create-projection [res-id res-id-2 map-of-options])                    projection-id
(create-batchprojection [res-id res-id-2 map-of-options])          batchprojection-id
(create-project [res-id res-id-2 map-of-options])                          project-id
(create-configuration [res-id res-id-2 map-of-options])              configuration-id
(create-sample [res-id res-id-2 map-of-options])                            sample-id
(create-library [res-id res-id-2 map-of-options])                          library-id
(create-script [res-id res-id-2 map-of-options])                            script-id
(create-execution [res-id res-id-2 map-of-options])                       execution-id
```

Using the resource-specific create procedures is thus just a matter of directly translating the corresponding calls to the basic `create` primitive:

```
(create-source {"remote" "s3://bucket.com/data.csv"}))
(create-dataset "source/121212321231231231231233")
(create-ensemble "dataset/abababab32ab3ab312312f1233"
                 {"number_of_models" 13})
(create-prediction "model/562fe2d8636e1c5ec500688c"
                   {"input_data" {"age" 23}})
(create-batchanomalyscore "anomaly/fffe2d8636e1c5ec50069ac"
                          "dataset/a212b23212312312312312233")
```

All *create* procedures will implicitly wait for their parent resources to finish, without the need for explicit calls to `wait` (see Subsection 4.12.8 below) in your code. Thus, despite the fact that when you call, say

```
(create-source {"remote" "s3://bucket.com/data.csv"}))
```

the given source is just queued for creation when the procedure create-source returns its identifier, that identifier can be immediately used in other create calls, and the WhizzML runtime will make sure that all parent resources are finished before starting creating their children. Thus, the following "one-click" ensemble from a source identifier is safe:

```
(let (src-id (create-source {"remote" "s3://bucket.com/data.csv"})
      ds-id (create-dataset src-id))
  (create-ensemble ds-id {"number_of_models" 20}))
```

and could even be rewritten without intermediate variables as:

```
(create-ensemble (create-dataset (create-source {"remote"
                                                 "s3://bucket.com/data.csv"}))
                 {"number_of_models" 20})
```

It's also possible to list the ids of created (and not deleted) resources, at any point during the execution of a whizzml program:

```
(created-resources )                                          list of resource-id
```

So, for instance, you could delete all the resources created during a script execution with the following expression in your source:

```
(for (id (created-resources)) (delete id))
```

Some batch resources can create an additional dataset, whose identifier is always found in the `output_dataset_resource` property. To obtain a list of created resources that also includes those datasets you can use:

```
(created-resources* )                                         list of resource-id
```

To identify resources containing an associated dataset, use:

```
(batch-resource-types )                                     list of resource-type
```

This standard procedure returns a list of resource types, all of which have an optional dataset associated to them. Typical examples are "batchprediction" or "batchprojection".

4.12.8 Waiting for Resource Completion

Resources created by the `create` family of functions will evolve from state 1 (queued) to state 5 (finished) or -1 (faulty). The `wait` and `wait*` procedures will block waiting for the resources status to be 5 before returning its identifier, or signal an error if it reaches -1.

```
(wait res [int-timeout])                                             resource-id
(wait* list-of-res-id)                                          list of resource-id
```

`wait` returns `res` as soon as the resource reaches its finished status or the (optional) timeout expires. If you want to wait forever, don't pass any timeout to `wait`.[6]

The standard procedure `wait*` just waits in turn for each of the resources in `list-of-res-id`, and returns the list of resources upon completion. It can be defined in pure WhizzML simply as:

```
(define (wait* ids) (map wait ids))
```

[6] In practical terms, the server won't let you block indefinitely in these calls, because your script will have a maximum running time.

If any resource enters a failed state while waiting (or is failed right away), the waiting functions signal error code -50.

Note that, frequently, you will not need to explicitly call `wait` on resources that are going to be use to create other resources, since, as explained in Subsection 4.12.7, the creation primitives implicitly wait for parent resource completion. The most common use cases for `wait` or `wait*` explicit calls are just before calling `fetch` to access the metadata of a resource (for instance, you want to use the histogram of a dataset's field, and therefore need to make sure the dataset creation is finished) and when assigning a script's outputs (to ensure they are usable immediately after the script execution finishes).

4.12.9 Creating and Waiting for Resource Completion in one Call

We offer convenience procedures that will create a resource and use `wait` until it's either finished or in error. The generic procedure is called `create-and-wait`, and takes the resource type and a map of creation parameters as arguments:

```
(create-and-wait str-type map-of-options)                    resource-id
```

where the arguments have the same meaning as for `create` (see Subsection 4.12.7).

As with `create*` above, there is an equivalent method, `create-and-wait*` to create a list of resources in parallel and wait for them all to complete.

```
(create-and-wait* listof-types list-of-options)          list of resource-id
```

If the creation of all resources completes successfully, the procedure returns a list of resource ids. **If not, the procedure attempts to delete all resources in the list, completed or not**, and raises an error with code -60 and the id of the first failed resource.

There are also specific versions of create and wait for each resource type, each taking as their single argument a map specifiying the creation parameters:

```
(create-and-wait-source map-of-options)                                  source-id
(create-and-wait-dataset map-of-options)                                dataset-id
(create-and-wait-model map-of-options)                                   model-id
(create-and-wait-composite map-of-options)                             composite-id
(create-and-wait-fusion map-of-options)                                  fusion-id
(create-and-wait-optiml map-of-options)                                  optiml-id
(create-and-wait-ensemble map-of-options)                              ensemble-id
(create-and-wait-prediction map-of-options)                          prediction-id
(create-and-wait-batchprediction map-of-options)                  batchprediction-id
(create-and-wait-evaluation map-of-options)                           evaluation-id
(create-and-wait-anomaly map-of-options)                               anomaly-id
(create-and-wait-anomalyscore map-of-options)                       anomalyscore-id
(create-and-wait-batchanomalyscore map-of-options)               batchanomalyscore-id
(create-and-wait-cluster map-of-options)                               cluster-id
(create-and-wait-centroid map-of-options)                             centroid-id
(create-and-wait-batchcentroid map-of-options)                     batchcentroid-id
(create-and-wait-association map-of-options)                         association-id
(create-and-wait-associationset map-of-options)                    associationset-id
(create-and-wait-linearregression map-of-options)                 linearregression-id
(create-and-wait-logisticregression map-of-options)             logisticregression-id
(create-and-wait-correlation map-of-options)                           correlation-id
(create-and-wait-statisticaltest map-of-options)                   statisticaltest-id
(create-and-wait-topicmodel map-of-options)                            topicmodel-id
(create-and-wait-topicdistribution map-of-options)               topicdistribution-id
(create-and-wait-batchtopicdistribution map-of-options)     batchtopicdistribution-id
(create-and-wait-deepnet map-of-options)                               deepnet-id
(create-and-wait-timeseries map-of-options)                          timeseries-id
(create-and-wait-forecast map-of-options)                            forecast-id
(create-and-wait-pca map-of-options)                                       pca-id
(create-and-wait-projection map-of-options)                          projection-id
(create-and-wait-batchprojection map-of-options)                  batchprojection-id
(create-and-wait-project map-of-options)                               project-id
(create-and-wait-configuration map-of-options)                    configuration-id
(create-and-wait-sample map-of-options)                                sample-id
(create-and-wait-library map-of-options)                               library-id
(create-and-wait-script map-of-options)                                script-id
(create-and-wait-execution map-of-options)                           execution-id
```

4.12.10 Fetching Resources

The `fetch` call, which takes a resource identifier, retrieves the full resource metadata *in its current status*.

```
(fetch res [map-of-options])                                            resource map
```

The optional `map-of-options` argument is a map with any desired key/values to use in the HTTP GET requests used to fetch the resource. Typical parameters are fields filters, as in the following example:

```
(fetch "source/1212222343556aa343433"
       {"fields" "000000,00000a" "offset" 10})
```

4.12.11 Updating Resources

To update an existing resource given its id and a map describing the changes to apply (again, with the key/values that you would use in a regular API call), use:

```
(update res map)                                                            boolean
(update-and-wait res map)                                                 resource-id
```

The `update` primitive makes sure the requested resource is finished (waiting for it to finish if necessary) and requests from the server the given update, specified by means of `map`. The procedure returns immediately the resource identifier if the server has accepted the update request, signaling an error code -50 if the server cannot be contacted or refuses the request.

Resource updates are generally an asynchronous operation in BigML, so you will sometimes want to `wait` on an updated resource (see Subsection 4.12.8) in order to see the change you just requested in a fetch call: the built-in `update-and-wait` will do that in a single step, and it could be implemented in pure WhizzML as:

```
(define (update-and-wait id params)
  (wait (update id params)))
```

Note however that you do not need explicit calls to `wait` or `update-and-wait` in order to use an updated resource as the parent of another one (see also Subsection 4.12.7 and Subsection 4.12.8), since the corresponding `create` call will implicitly wait for you. Thus, for instance, in the following call:

```
(create-model (update ds-id {"objective_field" {"id" "000001"}}))
```

it is guaranteed that the model will be created using "000001" as its objective field (in other words, the `update` operation is started and completed before `create-model` starts, despite the fact that it is asynchronous).

4.12.12 Deleting Resources

```
(delete res [map])                                                    boolean
(delete* list-of-res-id)                                    list of boolean
```

The `delete` function deletes any resource type from your account. On success, `delete` returns `true`. There are a few cases where a delete request may be accompanied by options (which in the API appear in the request's query string). For instance, when deleting executions, one can request the deletion of their child resources by setting `delete_all` to true. For those cases, `delete` accepts an optional map argument, `map`. `delete*` iterates over the given list of resource identifiers, deleting all of them and returning a list of success flags.

Examples:

```
(delete "sample/57a3c4da58a27e5803005880")
(delete "execution/57abf210eb3273117e000000" {"delete_all" true})
```

4.12.13 Field Procedures

The standard library includes some helper procedures to aid in the manipulation of field maps and individual fields, as described below.

Field descriptors are present in many BigML resources, usually as a map under the key "fields", and play an important role in most workflows. Each field is identified by a unique identifier (usually, a key in a fields map) and is described as a map with keys such as "name", "optype" or "summary".

Fields map retrieval

To extract the fields map from a resource we can use:

```
(resource-fields res-or-id)                                               map
```

This procedure takes either a resource identifier or a full resource map (such as the ones returned by `fetch`) and extracts from it its map of fields. If the given argument is not of the correct type, an empty map is returned. For convenience, all the values in the returned map contain the key "id" with the corresponding field identifier.

Once we have at our disposal a fields map, a very common operation is to fetch from it the descriptor of a single field. If we know the field identifier, that operation is trivial (just a map lookup), but it's often the case that we want to perform a lookup by field name.

```
(find-field map-of-fields str)                                    map
```

The standard procedure `find-field` takes a fields map (as returned by, e.g., `resource-fields`) and looks up an individual field by either its identifier or its name (passed as `str`). The procedure returns `false` if the lookup fails.

Field properties

An individual field descriptor is a map with the field's properties. To make sure a map value is actually a field descriptor you can use the `field?` predicate:

```
(field?  map)                                                  boolean
```

This procedure will make sure that the passed `map` has a "name" and an "optype" keys, with valid a value for them, so that they contain the bare minimum information related to a field.

There is a collection of predicates to check the optype of a given field:

```
(categorical-field?  map)                                      boolean
(numeric-field?  map)                                          boolean
(text-field?  map)                                             boolean
(items-field?  map)                                            boolean
(regions-field?  map)                                          boolean
(image-field?  map)                                            boolean
(path-field?  map)                                             boolean
(datetime-field?  map)                                         boolean
```

An important bit of information contained in field summaries is the field's distribution, i.e., how the field's values are distributed across categories, bins, items or terms, depending on their specific optype. In all cases, the distribution is represented as a list of pairs. In each pair, the first component is the value that is being counted (category, bin center, item name, term), and the second component is its count (number of instances associated to the first value). The standard procedure `field-distribution` gives access to that information, regardless of the field's optype:

```
(field-distribution map)                                  list-of-pair
```

As mentioned, for categorical fields this procedure will return the "categories" in the field's "summary", for numeric field it will retrive either "bins" or "counts", for text field the key inside the summary will be "tag_cloud" and, for item fields, "items".

In the case of categorical, items and text fields, it is often useful to get a list of all the first elements in the distribution, which correspond, respectively, to the list of categories, items and terms for the field. For convenience, there are predefined procedures returning directly those lists:

```
(field-categories map)                                    list-of-string
(field-items map)                                         list-of-string
(field-terms map)                                         list-of-string
```

Field maps

Same resources taking as inputs collections of other resources that contain fields need a mapping from a set of fields to another one. For instance, when creating multi–datasets one may need to specify a mapping between fields of different input datasets; or, when making a batch prediction, sometimes we need to specify in the request what fields of the dataset to be scored correspond to the model fields. In those and other cases, a *fields map* is specified as a WhizzML map that maps identifiers between two sets of fields, with the default being an identity map.

It is not rare to find cases where the fields match by name instead of by identifier, and we need to construct an identifiers map associating together fields of the same name. E.g., given field maps:

```
{"000000" {"name" "field a" ...}
 "000001" {"name" "field b" ...}}
```

and

```
{"000000" {"name" "field b" ...}
 "000001" {"name" "field a" ...}}
```

we would like to specify the fields map:

```
{"000000" "000001"
 "000001" "000000"}
```

This happens often enough that the standard library provides a function to compute a fields map matching by the names of fields in two input collections (maps) of fields:

```
(match-fields-by-name from-fields to-fields)                                    map
```

So for instance, in simple cases, we could construct a fields map for a batch prediction from the corresponding supervised model and dataset with code along the lines of:

```
(match-fields-by-name ((fetch model-id) ["model" "model_fields"])
                      (resource-fields dataset-id))
```

4.12.14 Dataset Procedures

This section describes standard procedures specific to the creation and manipulation of datasets.

Objective field

```
(dataset-get-objective-id dataset-id)                                           string
```

Explores the given dataset metadata map and extracts from it the preferred objective field identifier. Some datasets have it already precomputed, and the function is then rather trivial (basically, a `get-in`); otherwise, a valid objective is selected from the field information, following the same algorithm as BigML's server side.

Row distance

BigML defines a positive-definite metric between instances of a dataset (used, for instance in clustering algorithms), which depends only on the properties of the dataset's fields. The primitives `row-distance` and `row-distance-squared` provide access to that metric.

```
(row-distance map-of-fields map-point [map-point2 map-scales])                  number
(row-distance-squared map-of-fields map-point [map-point2 map-scales])          number
```

The squared version is provided for convenience, as it's computationally more efficient, and squared distances are used directly in many cases. All arguments are maps with field identifiers as keys. The parameter `map-of-fields` gives, for each field identifier, its descriptor map (as found in any BigML resource under the "fields" key); `map-point` and `map-point2` are maps from field identifier to field value, and each one therefore defines a dataset instance. Finally, `map-scales` associates to each field a numerical scale to be used by the metric during the computation (that allows weighting of the individual dimensions involved in the distance computation).

Dataset splits

Splitting a dataset in two disjoint parts is a common operation, used for instance to separate a testing subset of our input data for evaluation purposes.

The WhizzML standard library provides two procedures for creating dataset splits:

```
(create-dataset-split dataset-id rate seed [first-options second-options])
list-of-dataset-id
(create-random-dataset-split dataset-id rate [first-options second-options])
list-of-dataset-id
```

To create a split, `create-dataset-split` needs an input dataset, given by its identifier, a sampling `rate` (a number between 0.0 and 1.0), which indicates the portion of the dataset that is sampled in the first part (so it'll be composed of $N * \text{rate}$ instances, where N is the total number of instances in the input dataset, while the second part will be composed of those instances not in the first, and therefore have $N * (1 - \text{rate})$ instances) and a seed used to initialize the random number generator that is used to select instances. If you pass the same seed to two calls to `create-dataset-split` you'll obtain identical results. For convenience, the standard library includes `create-random-dataset-split`, which picks up a random seed for you, and that is simply defined as:

```
(define (create-random-dataset-split dataset-id rate)
  (create-dataset-split dataset-id rate (str (rand-int 100000))))
```

Both procedures return a list of two elements, namely the identifiers of the datasets containing, respectively, the first and second parts of the instances in the input dataset.

Both procedures also take two optional maps, `first-options` and `second-options`, which are options that will be passed to the dataset creation calls for each half of the split. For instance, if you want that the first dataset in a split is called "First half" and the second "Second half", you would use something like:

```
(create-random-dataset-split "dataset/a212b23212312312312312233"
                             0.8
                             {"name" "First half"}
                             {"name" "Second half"})
```

Dataset merges

Multiple datasets can be merged into one in BigML simply by passing an "origin_datasets" list to `create-dataset`, with the only limitation that there is a maximum number of datasets accepted.[7] To skip that limitation, and perform the merge in an as parallel way as possible, we provide the `merge-datasets` primitive:

```
(merge-datasets list-of-datasets [map-params])                    dataset-id
```

The list of datasets can contain either dataset identifiers, or maps specifying the id and additional properties such as field maps, and any additional arguments passed as `map-params` will be used in all internal dataset creation requests. Here's an example of a script concatenating a dataset a hundred times, starting from an inline source:

```
(define data "a,b,c,1\na,b,c,2\nb,c,d,3\nb,b,c,3\na,a,a,4")
(define src-id (create-source {"data" data}))
(define ds-id (create-dataset src-id))
```

[7] 32 as of this writing

```
(define name "whizzml-test")
(define mds (merge-datasets (repeat 100 {"id" ds-id}) {"name" name}))
```

If we wanted to juxtapose instead of concatenate, we could write

```
(define mds (merge-datasets (repeat 100 {"id" ds-id})
                            {"name" name "juxtapose" true}))
```

instead.

4.12.15 Execution Procedures

This section describes auxiliary procedures that can be helpful when using the results of an `execution` as inputs for other executions.

Executions can be used in scripts as any other resource. It's a common practice to use the outputs of an execution as values in a `script`. In order to help doing that, the following procedures might be handy:

```
(execution-inputs execution-id [list-of-names])                                    list
(execution-outputs execution-id [list-of-names])                                   list
(execution-output-resources execution-id [list-of-names])                          list
(execution-sources execution-id)                                                   list
(execution-logs execution-id)                                                      list
```

The mandatory argument for all the procedures will be the `execution-id` of the execution that stores the information. In addition to that, some of the procedures will accept as a second optional argument a list of names to filter out the items to be included in the returned list. `execution-inputs` will return the list of values used as input arguments in the execution (filtered by argument name, if the second argument is used). `execution-outputs` will return the list of outputs (filtered by the output name if the second argument is set). The `execution-output-resources` will return the list of BigML resources created in the execution and the variables they were assigned to, if any. If the second argument is used, the list will be filtered by variable name. `execution-sources` returns a list of the scripts being executed and their dependencies, if any. The index of this list is used as reference in the execution call-stack and location information to reference errors. Finally, `execution-logs` will return the lines logged to the console.

Using as example a script with this simple code:

```
(log-info "That's foo: " foo)
(log-info "Here's bar: " bar)
(define sources (for (i (range bar))
                  (create-source {"remote"
                                  "https://static.bigml.com/csv/iris.csv"})))
(define source1 (sources 0))
(define source2 (sources 1))
(define division (/ foo bar)
```

these would be the outputs of each of the procedures described above:

```
(execution-inputs "execution/5d5c4a0eeba31d6280001ee2")
  ;; => [["bar" 2] ["foo" 6]]
(execution-inputs "execution/5d5c4a0eeba31d6280001ee2" ["bar"])
  ;; => [["bar" 2]]
(execution-outputs "execution/5d5c4a0eeba31d6280001ee2")
  ;; => [["division" 3.0 "Number"]
  ;;     ["sources"
  ;;     ["source/5d5c4a0ec5f953036601bd0b" "source/5d5c4a0ec5f953036e0333fb"]
  ;;      "list"]]
(execution-output-resources "execution/5d5c578042129f7dfc00339d" ["source2"])
  ;; => [{"code" 5
```

```
  ;;      "id" "source/5d5c5780c5f953036d00ae11"
  ;;      "last_update" 1566332800805
  ;;      "progress" 1.0
  ;;      "state" "finished"
  ;;      "task" "Done"
  ;;      "variable" "source2"}]
(execution-sources "execution/5d5c4a0eeba31d6280001ee2")
  ;; => [["script/5d5c4a04eba31d6280001edf" ""]]
(execution-logs "execution/5d5c4a0eeba31d6280001ee2")
  ;; => [["info" "2019-08-20T19:29:18.239Z" 0 1 "That's foo: 6"]
  ;;     ["info" "2019-08-20T19:29:18.239Z" 0 2 "Here's bar: 2"]]
```

4.13 SMACdown

SMAC is a meta-algorithm for the optimization of any candidate function against a set of possible parameter values. For example, the function may be a random forest learning algorithm. In that case, the parameters would be the number of trees, number of random candidates, pruning or no pruning, etc. The criterion could be the error of the generated model on a holdout set.

WhizzML provides an implementation of the SMAC algorithm, dumbed **down** so that one can insert it somewhere in an existing optimization pipeline regardless of the context.

To use the provided function, `smacdown-optimize` one need only provide two functions as arguments, a `generator` and an `evaluator`. The generator takes no arguments and returns a random set of parameters to be evaluated as a map. For example:

```
{"number_of_trees" 57
 "random_candidates" 20
 "pruning" true}
```

The values of the parameters can be numbers, strings, or boolean values. There are no further restrictions on the form of the map, except that every generated map must have the same set of keys.

The `evaluator` takes a list of parameter maps, of the sort generated by `generator`, evaluates these candidates using the chosen criterion, and returns the resulting list of evaluations. For example, the evaluator may train a model using each set of candidate parameters provided, then evaluate those models on a holdout set of data, returning the error of each one. Note that **the criterion must be constructed so that lower values indicate higher quality parameters**. Said another way, SMACdown will search for the set of parameters that **minimizes** the objective criterion.

Given these two functions, `smacdown-optimize` learns a clever search strategy for the best set of parameters. It returns a list of all of the maps of parameters it has evaluated, sorted high to low by quality, so the **lowest** score for the objective criterion is first. The objective criterion for each of the output maps is stored under the key `actual_value` and the parameters evaluated are stored under the key `parameters`.

4.14 Resource Workflow

Every resource in BigML is totally reproducible. By inspecting the attributes defined in its JSON and those of the resources it's derived from, we can reproduce the chain of steps that led to its creation. The summary of these create|update|get operations can be obtained using the `resource-workflow` procedure.

```
(resource-workflow res-id bool bool [mapoptions])                    map
```

The first argument should be the ID of the resource whose creation workflow is rebuilt. When the second one is set to `true` the workflow will not contain the name or the range or rows in the case of datasets, so that you can reuse it to process new data files with the same structures. The third attribute set to `true` will only rebuild the last step of the workflow. The last optional argument can contain a map where different parameters can be used to tweak the process.

The available options so far are `excluded-attrs`, `stop-res-ids`, `level` and `prediction-wf`. We can use `excluded-attrs` mapped to a list of attributes so that they are not included in the procedure output. Using this option you can avoid storing some of the attributes that, even if they are mandatory for tracing purposes, need to be spared when retraining or sharing resources, like the project a resource has been created in. Also to stop the recursive process at some especific resource, we can use `stop-res-ids` mapped to a list of IDs. When any of these resources are found in the chain of parent resources, the recursive call will stop. Alternatively, the chain can be stopped after some steps using the `level` option set to the number of steps back that you want to script. Finally, the `prediction-wf` set to `true` will generate a wokflow that only takes into account the transformations needed for the test datasets to reproduce prediction resources, like a `batchprediction` or a `batchcentroid`. The models involved in the prediction chain will be stored in the `inputs` attribute of the workflow using their IDs.

The resulting map will contain the following attributes:

Workflow attribute	Description
`steps`	List of maps describing the information about the resource being operated on, and the operations applied
`inputs`	List of inputs that must be provided for the workflow to start, either IDs or remote URLs
`output`	ID of the resource being reified
`name`	Name of the resource being reified
`description`	Description of the workflow
`last-step`	Whether the workflow includes only the last step
`reuse`	Whether the workflow has been adapted for reuse
`type-counters`	Summary of the number of resources per type used in the workflow
`mapped-ids`	Map of the IDs in the workflow to the variables that represent them
`var-ids`	Map of the variable names in the workflow to the original IDs

Table 4.2: Workflow attributes

Each `step` has also a uniform structure:

Step attribute	Description
`action`	Type of action (`create`, `update` or `get`)
`origin`	The variable that contains the information used as origin when operating on the resource. It can also be a map with the origin attribute and the variable that contains its value
`order`	Number of the step in the workflow
`output`	Variable that will contain the generated output
`args`	Arguments used in the API call action. Contains both the origin information and the configuration attributes. Usually it's a map, but when the action is `get`
`ref`	Map that contains the reference information for the resource being the output of the step, like its ID, name, name_options and creator

Table 4.3: Step attributes

An example of a step would be.

```
{"ref"
```

```
{"id" "dataset/5babb5bf92fb56105d001f32"
 "name" "iris"
 "name_options" "150 instances, 5 fields (4 numeric, 1 text)"
 "creator" "demo_user"}
"output" "dataset2"
"action" "create"
"origin" {"source" "source2"}
"args"
{"source" "source2" "objective_field" {"id" "000003"} "all_fields" false}
"order" 6}
```

4.15 Scriptify

The `scriptify` procedure will receive the result of a `resource-workflow` and produce from that the `WhizzML` code needed to rebuild it. Depending on whether the `reuse` flag value, the code will use or not the original ranges and names for every resource created. The result of the procedure is a string that contains the source code describing the workflow steps.

Also, workflows created using the `prediction-wf` flag will generate predictive scripts. These scripts will have a headers section where all models involved in the workflow are defined using their IDs and only the test sources and/or datasets are recreated from the data provided as input for the script.

```
(scriptify workflow)                                                    string
```

Index

Index of Standard Procedures

List of Tables


bigml