BARD

mikel evins

Constants 3

Special forms and standard macros 4

Built-in types and literals 17

- Anything 17
- Applicable 17
- Boolean 18
- Character 18
- Eof 19
- List 19
- Map 21
- Name 21
- Null 22
- Number 22
- Stream 22
- Type 23
- Undefined 23

Quotation 24

Pattern matching 25

Bard Types 26

Conditions 27

Serialization and images 28

Remote agents and messaging 29

Protocol Reference 30
Anything 30
Applicable 30
Classes 30
Functions 30
As 30
Classes 30
Functions 30
Boolean 31
Classes 31
Functions 31
Character 31
Classes 31
Functions 31
Eof 32
List 33
Map 33
Name 33
Null 33
Number 33
Stream 34
Type 34
Undefined 34

Constants

`eof`
The unique end-of-file value.

`no`
The Boolean false value.

`nothing`
The unique null value. `nothing` is a member of all types. As a collection it represents the empty collection. As an atom it represents a null value.

`undefined`
The unique undefined value. `undefined` is Bard's 'bottom' value. It represents the result of evaluating some expression that cannot produce a meaningful value, such as an unbound variable.

`yes`
The Boolean true value.

Special forms and standard macros

begin *expression1 expression2* **...** => Anything
Evaluates the expressions in order from left to right, returning the value of the last expression evaluated.

cond (*test1 body1*) (*test2 body2*) **...** => Anything
Evaluates *test* forms in order from left to right. The first *test* form that returns a true value causes Bard to evaluate the associated *body* form and return its result. No further forms are processed after a *body* form is selected and evaluated.

define variable *var val* [:mutable *mutable?*] **...** =>*val*
Creates a new toplevel variable named *var* and bound to *val*, or changes the binding of *var* to *val* if it already exists. define evaluates *val*, so it can be any expression; it does not evaluate *var*, which must be a symbol.

If *mutable?* is true then (setter *var*) returns a function of one argument that can be used to update the value bound to *var*, and (set! *var x*) assigns the value of *x* to *var*. The default value of *mutable?* is no.

```
define function [(protocol protocol)]
  (fname arg1 arg2 ...
        [& restarg])
        [{ key1 val1 key2 val2 ... }]
   expr1 expr2 ...
=> fname
```

Creates or augments a function, defining a method that matches the parameter list given in the define function expression.

If the protocol clause appears then the function is added to the named protocol, or, if a function of the same name already exists, its definition is augmented or replaced. If the protocol clause does not appear then the protocol affected is Anything.

The new value of *fname* is a function that accepts arguments as specified by *arg1 arg2 ...* and which has a method whose execution is described by *expr1 expr2 ... .* When the

method is called, the expressions *expr1 expr2 ...* are evaluated from left to right as if in the body of a `begin` form, and the value or values of the final expression are returned.

The arguments arg1 arg2 ... are required formal parameters. A required formal parameter may take either of two forms:

```
arg
```

or:

```
(arg type)
```

In either case, *arg* is a symbol that is used as the name of the parameter. The name becomes a lexical variable in the environment of the method body when it's called.

In the second form, the formal parameter is a list whose first element is the symbol used as a parameter name, and whose second element is a type qualifier. The type qualifier is evaluated at the time the `define-function` form is evaluated, yielding a type object. The method being defined matches an input parameter in *arg1* only if its type is *type1*. This feature makes Bard functions polymorphic in the types of their inputs: defining different methods on different input types results in a function that calls a different method for inputs of different types.

The type expression can be any Bard expression that yields a type object. Specifically, it can be an expression of the form `(singleton foo)`, which yields a **singleton type**. A singleton type is an object that Bard recognizes as a type, but which represents a specific value (that is not normally treated as a type). For example, `(singleton 3)` returns a singleton object that represents the number 3 as a type.

Bard functions can match input arguments against singleton types, dispatching on specific values. This feature is equivalent to what Common Lisp programmers call "EQL specializers".

If the "`{}`" clause appears, it defines a set of **keyword parameters**. *key1*, *key2*, and so on are symbols that become the names of lexical variables in the method body. Passing a keyword with the same name as *key1* when the function is called causes the variable *key1* to be bound to the value of the expression after the keyword.

The "`{}`" clause is syntactically a `Map`, and for each *key* a *val* must appear. The val becomes the default value of the corresponding key, used as the value of the corresponding lexical variable when the function is called without supplying the keyword argument.

If the "&" clause appears, it defines a **rest** argument. All parameters that appear after the position of the rest argument are collected into a list, which is then bound to a lexical variable whose name is the same as the symbol *restarg*.

If both rest and keyword arguments appear in a function definition then all keyword arguments must appear after all rest arguments. When Bard evaluates a function call it first binds all re-

quired arguments to the corresponding required parameters, then binds all remaining arguments up to the first keyword argument to the *restarg* parameter, then collects all keyword arguments and binds them to variables named for the corresponding keyword parameters.

define macro (*macro-name parameter-list*) [:as *form-name*]
   body
=> *macro-name*

Creates a new toplevel variable, or redefines an existing one, so that the variable is bound to a Bard macro whose behavior is given by *body*. After successful evaluation of the define-macro form, an expression of the form (*macro-name*...) is a macro call that first applies the macroexpansion given by body to the form, then compiles and executes the resulting Bard expression.

macro-name is a symbol that becomes the toplevel name of the new macro.

parameter-list is a Bard parameter list of the same form used by define-function.

body is zero or more Bard expressions that are executed at macroexpansion time; the value of the last expression becomes the new Bard expression that is to be executed in place of the macro call.

If the as: *form-name* clause appears then the variable form-name can be used in the body of the macro to refer to the entire macro form; in other words, if we define a macro like so:

```
(define-macro (foo bar) :as mexpr ...)
```

and then call it like so:

```
(foo grault)
```

then in the body of the macroexpander, the variable mexpr has the value (foo grault). The :as feature makes it easy to obtain a reference to the macro expression that is being expanded, for cases where macroexpansion needs the whole toplevel expression.

Macroexpansion is purely syntactic; a macroexpander operates on the s-expression of the macro call. It does not execute the expression or any subexpressions; it simply transforms the expression to a new expression and returns the result. The resulting output expression is then executed in the normal course of a macro call.

This form of macro is the traditional Common Lisp-style defmacro facility; it is not syntactically hygienic, and care must be taken not to accidentally capture and alias variables in the enclosing environment.

define protocol *protocol-name*
 (*fname1 param-list1*)
 (*fname2 param-list2*)
 ...
=> *protocol-name*

Creates a new toplevel variable, or redefines it if it exists already, binding the variable to a new protocol object. The protocol object represents a list of generic functions whose names are given by *fname1*, *fname2*, and so on.

The function names are defined as variables in a namespace defined by the protocol; for example, if we define a protocol named `Foo` that has a function named `bar`, that function becomes the value of a variable `Foo:bar`.

The parameter list of a protocol function is a sequence of zero or more **classes**. A class is a type variable; that is, it's a name for a (possibly undefined) type. Protocols do not define the types represented by classes; that job is done by `define-function`.

As an example, consider the following simple protocol definition:

```
(define-protocol Rational
  (numerator Ratio)
  (denominator Ratio))
```

This definition defines the protocol `Rational`, with functions `Rational:numerator` and `Rational:denominator`. It also defines the class `Rational:Ratio`, but it doesn't specify what types belong to that class.

We can define a concrete type for the classes `Rational:Ratio` by using `define-function` to add a method to each of the functions in the Rational protocol:

```
(define-function (Rational:numerator (x <fixnum>)) x)
(define-function (Rational:denominator (x <fixnum>)) 1)
```

One effect of these definitions is that the functions `Rational:numerator` and `Rational:denominator` now have methods that match and operate on arguments of type `<fixnum>`. A second effect of the definitions is that the schema `<fixnum>` is now a member of the class `Rational:Ratio`.

If we define a method on `<fixnum>` for `Rational:numerator`, but not for `Rational:denominator`, Bard warns us on subsequent uses of the `Rational` protocol that the protocol is only partly defined for values of type `<fixnum>`.

We can ask whether a given schema is a member of a class:

```
(class? <fixnum> Rational:Ratio)
```

Bard answer `yes` if every function where `Rational:Ratio` appears as an argument has a method definition that qualifies the `Rational:Ratio` arguments as type `<fixnum>`. If there are no such method definitions, Bard answer `no`. If some but not all of those methods are defined, Bard answers `:partial` (which in Bard is a generalized boolean that is logically equivalent to `yes`).

```
define schema schema-name (included-schema1 included-schema2 ...)
  field1
  field2
  ...
=> Schema
```

Creates a new toplevel variable named *`schema-name`*, or redefines it if it exists already. The new value of the variable is a `Schema`, a type object that describes how to construct a value of a type defined by the Schema.

The new schema has fields given by *`included-schema1`*, *`included-schema2`*, ... and by *`field1`*, *`field2`*, ...

The fields of the included schemas become fields in the new schema, as if their definitions were written out in the body of the `define schema` form.

If duplicate field names appear in a definition then Bard issues a warning, but allows the definition. Later field definitions supersede earlier ones. These rules apply to included schemas as well as to field definitions that appear in the body of the define schema form, so it's possible to incude a schema and then selectively redefine some of its fields.

A field expression takes either of the following forms:

```
field-name

(field-name field-type
         [ :default default
           :init initname
           :mutable mutable?
           :getter gettername
           :setter settername ])
```

field-name is a symbol that becomes the name of the field in the new schema.

field-type is an expression that yields a type object.

default is a value that is used as the value of the field in the event that no initial value is supplied; if no default is supplied in the schema definition then Bard uses the value `nothing`.

initname is a keyword. When `make` is called to create an instance of the schema, passing *initname* followed by an expression supplies the value of the expression as the initial value of the field. If *initname* is not passed as an argument to `make` then the initial value of the field is *default*. The default *initname* is a keyword with the same name as *field-name*.

If *mutable?* is true then a setter function can be used to change the value of the field; if not, then the field is immutable.

If a *gettername* is supplied then a function of one argument is created and bound to *getter-name* as a toplevel variable; calling that function on an instance of the schema returns the value of the field. Thus, to get the `x` field of a `<point>` object, we use:

```
(<point>:x p)
```

By default, Bard creates a function named *schema-name:field-name* that serves this purpose; the `:getter` parameter is provided to enable programmers to customize the names of their getters.

Similarly, Bard by default creates a setter function of two arguments, obtained using the `setter` special form. Assuming that our example `Point` schema's `x` field is mutable, we set it like so:

```
((setter <point>:x) p) 101)
```

The macro `set!` enables us to write the above expression like so:

```
(set! (<point>:x p) 101)
```

If we supply the `:setter` value `set-x!`, we can then write this as:

```
(set-x! p 101)
```

The default setter function raises an error (because by default schema fields are immutable).

```
define vector schema-name
   [ :minimum-count mincount
     :maximum-count maxcount
     :element-type elt-type
     :default default
     :elements elements
     :elements-parameter elements-param
     :mutable mutable?
     :getter gettername
     :setter settername ]
=> Schema
```

Creates a new schema just as `define-schema`, except that the new schema's representation is an indexable vector of values instead of a record-like mapping of names to fields.

`mincount` is the minimum number of values allowed in an instance of the schema. The default *`mincount`* is zero.

`maxcount` is the maximum number of values allowed in an instance of the schema. The default *`maxcount`* is `nothing`, which means any number of values (greater than or equal to zero) is allowed.

`elt-type` is a type object that specifies the allowed types of elements. For certain element types (e.g. `<bit>` and `<unsigned-word-8>`), Bard may choose an optimized representation of the vector schema. The default value of *`elt-type`* is `Anything`.

`default` is the default value of each element of the vector if no initial element is supplied, and if the minimum count is greater than zero. *`default`* must be of type *`elt-type`*.

`elements` is a list of values to be used as the initial elements of a new instance, in the absence of supplied initial values. All objects in *`elements`* must be of type *`elt-type`*. The count of

members of elements must be greater than or equal to *mincount*, and less than or equal to *maxcount*.

elements-param is a name to be used for passing initial elements to the make function when creating an instance of the schema. For example, supposing we defined a vector schema named <choice>, and passed the name :options as the value for *elements-param*, we could then create an instance of <choice> like so:

```
(make <choice> :options [:a :b :c])
```

If *mutable?* is true then elements of an instance of the schema can be changed after an instance is created. The default value of *mutable?* is no.

gettername and *settername* are used as in define-schema, with the following differences:

- the default getter name is schema-name-getter

- the default setter name is schema-name-setter

To get an element of a vector schema named <my-vector>, we call its getter like this:

```
(<my-vector>-getter my-vector n)
```

where n is some integer index into the vector.

Similarly, to update my-vector's nth element with a value val, we use:

```
((<my-vector>-setter my-vector n) val)
```

ensure *init-expr protected-expr cleanup-expr* **...** => Anything

ensure evaluates *init-expr*, then evaluates *protected-expr*, then evaluates *cleanup-expr*. *cleanup-expr* is evaluated when control leaves *protected-expr*, even if control leaves it abnormally, as in the case of a *with-exit* or an exception handler.

```
generate (var1 var2 ...)
    expr1 expr2 ...
    [(yield val) ... ]
    [(recur new1 new2 ...)]
=> Generator
```

Returns a Generator instance whose parameters are given in *var1*, *var2*, ... and whose behavior is defined in the body following the *vars* expression.

A `Generator` is a `Stream` that produces values one-by-one on-demand. Applying the function `next` to a `Generator` instance causes it to produce its next value.

`generate` creates a lexical environment like that created by a `let` expression, binding *`var1`*, *`var2`*, and so on as in `let`.

When `next` is applied to the generator, it evaluates *`expr1`*, *`expr2`*, and so on, halting and returning after the last expression is evaluated.

A `yield` expression causes the generator to return the value of *`val`* after it halts execution.

A `recur` expression causes the generator to replace the values of its *`vars`* with the values passed as arguments to `recur` after it halts execution.

Either a `yield` expression or a `recur` expression or both may appear at the end of the body of a generator. If both appear, they may appear in either order; the effects of both will be respected. No other expression will be evaluated after either a `yield` or `recur` expression.

If a generator's body has no `recur` expressions then the values of the vars never change and the expressions in its body are evaluated only once.

If a generator's body has no `yield` expressions then the generator never returns a value; applying `next` always causes it to return without producing a value.

A generator is an `InputStream`, and the functions of the `InputStream` protocol can be used to access its values. Stream functions produce values from generators by calling `next`.

`if` *`test then-form else-form`* `=> Anything`

Evaluates *`test`*. If the result is a logically true value then evaluates *`then-form`*. Otherwise, evaluates *`else-form`* .

`initialize` *`val initarg1 initarg2 ...`* `=> Anything`

Returns a value computed from *val*. `initialize` is a standard method that is called by `make` when creating an instance of a schema. In the body of an `initialize` method, all fields of `val` are effectively mutable, even if declared immutable. The body of the `initialize` method is free to modify the fields of *`val`* as-needed. This feature enables programmers to control the details of how newly-created objects are constructed.

The Bard runtime does not guarantee that the return value of initialize is a reference to the same memory as *`val`*; it is free to allocate new objects in order to support the modifications expressed in the body of initialize, so user code should never count on the identity of *`val`*.

initarg1, *initarg2*, ... are **initialization arguments**; they take the following form:

```
:init-name init-expr
```

The *:init-name* is the keyword passed for a field's `:init` parameter in the schema's definition. *init-expr* is an expression that gives the initial value of the field in the newly-created instance.

Supposing we defined a point schema like this:

```
(define schema <point> ()
  (x :init :x)
  (y :init :y))
```

Then we could create a Point instance with an x coordinate of 101 and a y coordinate of 202 like this:

```
(make <point> :x 101 :y 202)
```

let (*var1 var2* **...**) *expr1 expr2* **...**
=> Anything

Creates a lexical environment in which *var1*, *var2*, ... establish lexical bindings for variables, then evaluates *expr1*, *expr2*, ... in the resultng environment. References to the variables defined by *var1*, *var2*, ... in *expr1*, *expr2*, ... see the values established by the *var1*, *var2*, ... expressions.

A method returned from the body of a `let` expression is a closure that contains the lexical environment established by the `let` form.

A *var* expression can take the following forms:

```
(varname val-expression)
(varname1 varname2 ... values-expression)
```

In the first case, *varname* is bound to the result of evaluating *val-expression*.

In the second case, the variables *varname1*, *varname2*, ... are bound in order to the multiple values returned by *values-expression*.

Each *var* is lexically visible to all the *vars* that are defined after it.

```
loop (var1 var2 ...) expr1 expr2 ... (recur new1 new2)
=> Anything
```

`loop` creates a lexical environment like that created by a `let` expression, and evaluates *`expr1`*, *`expr2`*,... in that environment, just as `let` does. The difference is that in the body of the `loop` expression the program can call `recur`, which causes the body of the `loop` to be executed again.

`recur` takes a number of arguments equal to the number of `var`s. When the `loop` body is executed again, the `var`s are bound to the values passed as arguments to `recur`.

This facility is similar to the `loop`...`recur` pattern of Clojure, or to the named `let` pattern of Scheme, and provides a convenient form of iteration without requiring destructive assignment to lexical variables.

```
make schema initarg1 initarg2 ... => schema instance
```

Creates a new instance of schema and then calls `initialize`, passing the new instance and *`initarg1`*, *`initarg2`*, ... to the `initialize` method.

```
match ((vars-pattern-1 vals-pattern-1)
       (vars-pattern-2 vals-pattern-2) ...)
   expr1 expr2 ...
=> Anything
```

Creates a lexical environment in which variables given by *`vars-pattern-1`*, *`vars-pattern-2`*, are bound to values given by *`vals-pattern-1`*, *`vals-pattern-2`*. The *`vals`* patterns are any valid Bard expressions. The *`vars`* patterns are expressions that conform to a set of **pattern-matching** rules explained in the chapter, “Pattern Matching”.

The expressions *`expr1`*, *`expr2`*, ... are evaluated in the lexical environment created by matching the *`vars`* patterns against the *`vals`* patterns.

```
method parameter-list expr1 expr2 ...
=> Anything
```

Creates and returns a new method. *parameter-list* is a list of formal parameters as in `define function`, except that no type qualifiers are accepted. A method is a callable object. When ap-

plied to arguments, it binds the argument values to its formal parameters and evaluates *expr1*, *expr2*, ... in the resulting environment, returning the result of the last expression.

next *gen* **...**
=> Anything

gen must be a `Generator` instance created by `generate`. `next` causes the generator to execute its body, optionally producing a value when it returns.

next-method *params* **...**
=> Anything

`next-method` passes params to the next-most-specific method of the current function.

Bard functions are polymorphic on all required arguments.

set! *place val* **...** => *val*
A standard macro that expands to the expression `((setter` *place*`)` *val*`)`. Evaluating this expression has the effect of assigning *val* to *place*, assuming *place* is mutable. If *place* is not mutable then Bard signals an error.

setter *place* **...** => Method
Returns a function that can be applied to an argument to update the value stored at *place*. If *place* is not mutable then Bard signals an error. The returned setter function accepts one argument, the new value .

singleton *val* **...** => Singleton
Returns an object whose value is *val*, but which Bard recognizes as a type. Singleton objects can be used as types in function definitions to tell Bard to match a method to a specific value.

values *val* **...** => *val* **...**
Returns *val* and any additional values as multiple values. You can bind multiple variables to multiple values using `let`:

```
(let ((a b c (values 1 2 3)))
  b)
=> 2
```

unless *test expr1 expr2* **...** => *val* **...**

If evaluating *test* returns a false value then unless evaluates *expr1*, *expr2*, ... from left to right, returning the value of the last expression evaluated. If evaluating *test* returns a true value then unless returns no value.

when *val* **...** => *val* **...**

If evaluating *test* returns a true value then when evaluates *expr1*, *expr2*, ... from left to right, returning the value of the last expression evaluated. If evaluating *test* returns a false value then when returns no value.

with-exit (*exitname*) expr1 expr2 **...** => Anything

Creates a lexical environment in which *exitname* is bound to a function. The function bound to *exitname* immediately returns from the with-exit expression as if control had reached the end, regardless of the context of execution when it's called. with-exit can be used to exit from arbitrarily complex computations. The function *exitname* accepts any number of arguments and returns them as if rom a call to values.

Built-in types and literals

Bard's base types are called **schemas**. A schema is a concrete description of how data are laid out in memory.

Schemas are organized into related abstract types according to **protocols**. A protocol is a set of generic functions that together define an abstract type.

The chapter "Bard types" describes schemas and protocols in greater detail. This chapter lists Bard's built-in schemas, organized by the most important protocols that apply to them.

Anything

The abstract type of all Bard values. The Anything protocol is also the default protocol to which new functions are added by `define function` in cases where no protocol is specified.

Literal syntax

Every Bard value is a member of Anything; there is no special Anything value, and so no literal syntax for creating it.

Applicable

```
Applicable
   Function
   Method
   Primitive
```

Literal syntax

```
(function (Anything) Integer Integer)
(method (x y) (+ x y))
(^ x y . (+ x y))
```

The third example is synonymous with the second. The terser form is convenient in some situations, such as when using anonymous functions as arguments.

No literal syntax is provided for constructing primitives; Bard primitives must be created by modifying the Bard virtual machine.

Schemas

`<function>`
The type of Bard's built-in generic functions.

`<method>`
The type of Bard's built-in methods.

`<primitive>`
The type of primitive procedures implemented by Bard's virtual machine.

Boolean

Literal syntax

```
yes
no
```

Schemas

`<bit>`
The type of the values `0` and `1` when treated as Booleans, or as members of bitvectors.

`<boolean>`
The type of the values `yes` and `no`.

Character

Literal syntax

```
#\a
#\space
#\U+0065
```

Schemas

`<simple-character>`

The type of character values, represented as ASCII characters. Additional character types may be present, depending on the implementation, but **`<simple-character>`** should always be present.

Eof

Literal syntax

`eof`

Schemas

`<eof>`

The type of Bard's unique end-of-file value, `eof`.

List

Literal syntax

```
()
(a b c)
(as <pair> (a b c))
#<pair>(a b c)
[]
[a b c]
(as <object-vector> [a b c])
#<object-vector>[a b c]
“Foo bar baz”
```

Bard reads expressions delimited by "`(...)`" or "`[...]`" as lists. It differs from other Lisps is some details of how it treats list data.

Since Bard lists are abstract types, it doesn't make any promises about the concrete representation of a list constructed by reading an expression, unless you use one of the forms that specifies a particular schema.

If you use the "`(...)`" form to write a list, Bard infers that the resulting expression is to be treated as a function call. If you use the "`[...]`" form then it infers that you mean to construct a list as literal data, as if you had written "`(list...)`".

The empty list, when treated as a function call, returns `nothing`.

Schemas

```
List
   Pair
      <pair>
   Text
      <ascii-string>
   Vector
      Bitvector
      Bytevector
```

The types of Bard's List values. A list is a container value whose contents are a finite number of other Bard values stored in a stable order. Various schemas provide lists with various representations, suitable for different uses, and with different performance profiles.

`<pair>` provides an implementation of List in terms of the traditional Lisp cons cells. A cons cell is a structure consisting of two pointers, in which one pointer, the **head** is the first element of the list, and the other, the **tail**, is a reference to the remainder of the list (another cons cell).

The Pair protocol, an extension of List, provides pair-specific functions that allow us to treat `<pair>` instances as 2-tuples.

The Text protocol provides implementations of text strings, including at least `<ascii-string>`.

The Vector protocols implement lists as arrays of values. `<object-vector>` is an array of references to general Bard values; `<bitvector>` is an array of bits that may be represented as one or more integers; the Bytevector schemas are homogeneous arrays of machine words of various sizes.

Note: List is an extension of Map; that is, all schemas that are members of List are also members of Map. A List's keys are the indexes of its elements.

Map

Literal syntax

```
{}
{ :a 1 :b 2 :c (+ 1 2)}
```

Schemas

`<table>`

The types of Bard's Map values. A map is a container value whose contents are a finite number of associations between keys and values. Each key appears in a map only once.

Various schemas provide maps with various representations. Maps whose keys are stored in a stable order are also members of List; the List protocol treats such maps as lists of key/value pairs.

Name

Literal syntax

```
:Foo
‘bar
#<class>Integer
```

Schemas

`<class>`

The type of Bard **classes**. Bard's classes are simple symbolic names that serve as type variables. They appear in protocol definitions, where they name the abstract types of function arguments. The chapter "Bard Types" describes classes in greater detail.

`<keyword>`

The type of Bard **keywords**. Keywords are symbolic constants that always evaluate to themselves.

`<symbol>`

The type of Bard **symbols**. Symbols are symbolic constants used as the names of Bard variables.

Null

Literal syntax

```
()
[]
nothing
```

Schemas

`<null>`
The type of Bard's unique null value, `nothing`. The forms "`()`" and "`[]`" are synonyms for `nothing`.

Number

```
Number
   Integer
      <bignum>
      <fixnum>
   Fraction
      Float
         <flonum>
      Ratio
         <ratio>
...
```

The types of various numeric values implemented by the Bard implementation. The exact set of schemas may be extended, but the above list should always be present.

Stream

```
Generator
   <generator>
   InputStream
   OutputStream
      IOStream
```

The types of various stream structures that represent data sources and sinks.

The Generator protocol implements procedural objects that can be repeatedly involved to produce sequences of values.

IOStream, and its supertypes InputStream and OutputStream, implement objects that either produce or consume values, or both. There are numerous schemas and extensions of all three of these protocols, implementing input or output, or both, for binary octets, character data, or higher-level data structures such as Bard values.

Type

`<protocol>`
`<schema>`
`<singleton>`

The types of values that represent Bard types.

Undefined

`<undefined>`
The type of Bard's unique undefined value, `undefined`.

Quotation

Pattern matching

Bard Types

Conditions

Serialization and images

Remote agents and messaging

Protocol Reference

Anything

Applicable

Describes operations on values that can be applied to arguments to produce a result.

Classes

`Applicable`

Functions

`applicable?` `Anything => Boolean`
Returns `yes` if its argument is applicable, and `no` otherwise.

`apply` `Applicable List => Anything`
Applies its Applicable argument to the elements of its List argument, returning the value or values produced.

As

Describes operations that can be applied to values to construct equivalent values of different types.

Examples

```
(as <fixnum> 2.0) => 2
(as <flonum> 2) => 2.0
(as <symbol> “Foo!”) => Foo!
(as <pair> “Foo!”) => (#\F #\o #\o #\!)
```

Classes

Functions

`as` `(singleton` *`Type`*`) Anything =>` *`Type`*
Converts its `Anything` argument to a value of type *`Type`*, assuming that’s possible. The new *`Type`* instance is equivalent to the `Anything` value given as input, but is an instance of a

schema conforming to *Type*. If the *Type* argument is itself a schema then the result is an instance of that schema. If the *Type* value is a protocol, then the result is an instance of a schema conforming to that protocol, chosen arbitrarily by the Bard runtime.

If the conversion cannot be performed then Bard signals an error.

Boolean

Describes logical operations on values that are logically equivalent to `yes` and `no`.

Classes

`Boolean`

Functions

`boolean?` `Anything => Boolean`
Returns `yes` if the `Anything` value is `yes` or `no`, and return `no` otherwise.

`false?` `Anything => Boolean`
Returns `yes` if its argument is logically false. The Bard values `no`, `nothing`, and `undefined` are logically false.

`true?` `Anything => Boolean`
Returns `yes` if its argument is logically true. All Bard values are logically true except for `no`, `nothing`, and `undefined`. The canonical true value is `yes`.

Character

Describes operations on text characters.

Classes

Functions

`character?` `Anything => Boolean`
Returns `yes` if its argument can be treated as a text character.

Equal

Describes operations that report whether two or more values are equivalent.

Classes

`Character`

Functions

`= Anything Anything [`*`Anything`*`]* => Boolean`

Returns `yes` if its arguments are equal values, and `no` otherwise. Bard's = function is polymorphic, and compares values using methods specific to their schemas. In the case of aggregate objects, it may in general recursively compare components of the compared values.

`identical?` `Anything Anything [`*`Anything`*`]* => Boolean`

Returns `yes` if its arguments are references to the same object, and `no` otherwise. If `identical?` returns `yes` then its arguments are all references to exactly the same memory location.

Eof

Describes operations on Bard's unique end-of-file value.

Classes

Functions

`eof?` `Anything => Boolean`

Returns `yes` if the `Anything` value is the end-of-file value, and `no` otherwise.

Float

Describes operations on floating-point representations of real numbers.

Classes

Functions

`float?` `Anything => Boolean`

Returns `yes` if the `Anything` value is an instance of a schema representing a floating-point number.

Foreign

Describes operations on foreign values—that is, on values that are references to data implemented in C or some other language that is not Bard.

List

Describes operations on collections whose contents are finite numbers of values maintained in a stable order.

Map

Describes operations on collections whose contents are finite numbers of values arranged conceptually in pairs, where each pair consists of a key and a value. (A Map value may not necessarily store its contents as instances of the <pair> schema.) Each key may appear only once in each Map instance. Each Map schema implements a comparison that it uses to determine whether two keys are the same.

In general, Map keys and values may be any Bard value except `undefined`. Specific schemas may impose additional type constraints on keys and values.

Name

Describes operations on values that serve as textual names and labels. These values include symbols, keywords, and classes.

Null

Describes operations on Bard's unique null value, `nothing`.

Number

Describes operations on values that can be treated as numbers, including integers, ratios and other fractions, complex numbers, and so on.

Ordered

Describes operations on values that can be stably sorted into a deterministic order. Examples include numbers, text characters, and text strings.

Stream

Describes operations on values that can serve as sources for sinks for values. Examples include input and output streams, generators, and lazy lists.

Type

Describes operations on values that Bard can treat as datatypes, including schemas, protocols, classes, type-synonyms, and singletons.

Undefined

Describes operations on Bard's unique undefined value, `undefined`. The only such operations are `undefined?` and its inverse, `defined?`.