Ballerina Language Specification, v0.970, Working Draft, 2018-05-01

Primary Contributors:

- Sanjiva Weerawarana, Lead, sanjiva@wso2.com
- James Clark, Design Co-Lead, jjc@jclark.com
- Sameera Jayasoma, sameera@wso2.com
- Hasitha Aravinda, hasitha@wso2.com
- Srinath Perera, srinath@wso2.com
- Frank Leymann, frank.leymann@iaas.uni-stuttgart.de

Other Contributors (in alphabetical order):

- Shafreen Anfar, shafreen@wso2.com
- Afkham Azeez, azeez@wso2.com
- Anjana Fernando, anjana@wso2.com
- Chanaka Fernando, chanakaf@wso2.com
- Joseph Fonseka, joseph@wso2.com
- Paul Fremantle, paul@wso2.com
- Antony Hosking, antony.hosking@anu.edu.au
- Kasun Indrasiri, kasun@wso2.com
- Tyler Jewell, tyler@wso2.com
- Anupama Pathirage, anupama@wso2.com
- Manuranga Perera, manu@wso2.com
- Supun Thilina Sethunga, supuns@wso2.com
- Sriskandarajah Suhothayan, suho@wso2.com
- Isuru Udana, isuruu@wso2.com
- Rajith Lanka Vitharana, rajithv@wso2.com
- Mohanadarshan Vivekanandalingam, mohan@wso2.com
- Lakmal Warusawithana, lakmal@wso2.com
- Ayoma Wijethunga, ayoma@wso2.com

Language and Document Status

The design of the Ballerina language is still in progress and its specification in this document is far from the complete. Furthermore, the implementation of the language is not yet completely aligned with this specification. We are releasing this document now in the hope that it will, even in its current incomplete state, be useful to those who wish to understand, evaluate or start using the language.

Sections 1 to 5 are significantly closer to completion than subsequent sections, which are in many places little more than an outline. For future versions of this document, we expect to have in place a proper process for handling comments; in the meantime, comments may be sent to the ballerina-dev@googlegroups.com mailing list.

Table of Contents

1. Introduction

Ballerina is a concurrent, transactional, statically typed programming language. It provides all the functionality expected of a modern, general purpose programming language, but it is designed specifically for integration: it brings fundamental concepts, ideas and tools of distributed system integration into the language with direct support for providing and consuming network services, distributed transactions, reliable messaging, stream processing, security and workflows.

Ballerina provides dual textual and graphical syntaxes. The graphical syntax is not described in this document, but is a fundamental aspect of the design of Ballerina; the syntax and semantics of Ballerina have been designed from the outset to support both syntaxes and both syntaxes are equally expressive.

Ballerina’s concurrency model and graphical syntax are both based on sequence diagrams. In Ballerina, concurrency is an inherent part of the concept of a function. A Ballerina function can be thought of as a sequence diagram where the actors of the sequence diagram are either concurrent blocks of code, called workers, or represent network endpoints that those workers interact with. An entire Ballerina program can thus be thought of as a collection of sequence diagrams that interact with each other.

Ballerina’s type system is much more flexible than traditional statically typed languages. First, it is structural: instead of requiring the program to explicitly say which types are compatible with each other, compatibility of types and values is determined automatically based on their structure; this is particularly useful in integration scenarios that combine data from multiple, independently-designed systems. Second, it provides union types: a choice of two or more types. Third, it provides open records: records that can contain fields in addition to those explicitly named in its type definition. This flexibility allows it also to be used as a schema for the data that is exchanged in distributed applications. Ballerina’s data types are designed to work particularly well with JSON; any JSON value has a direct, natural representation as a Ballerina value. Ballerina also provides support for XML and relational data.

As Ballerina programs are expected to produce and consume network services, it includes a distributed security architecture to make it easier to write applications that are secure by design. This includes taint checking and propagation and an integrated authentication & authorization architecture.

Ballerina is designed for modern development practices with a component based development model with namespace management via component repositories, including a global shared central repository. Component version management, dependency management, testing, documentation, building and sharing are part of the language platform design architecture and not left for later add-on tools.

The Ballerina standard library is in two parts: the usual standard library level functionality (akin to libc) and a standard library of network protocols, interface standards, data formats, authentication/authorization standards that make writing secure, resilient distributed applications significantly easier than with other languages. This document only covers the language syntax and not the standard library.

Ballerina has been inspired by C, C++, Java, Go, Rust, Haskell, Kotlin, Dart, JavaScript, TypeScript, Flow, Swift, Relax NG and other languages.

2. Notation

Productions are written in the form:

```
symbol := rhs
```

where symbol is the name of a nonterminal, and `rhs` is as follows:

- `0xX` means the single character whose Unicode code point is denoted by the hexadecimal numeral X
- `^x` means any single Unicode code point that does not match x and is not a disallowed character;
- `x..y` means any single Unicode character whose code point is greater than or equal to that of x and less than or equal to that of y
- `str` means the characters `str` literally
- `symbol` means a reference to production for the nonterminal `symbol`
- `x|y` means x or y
- x&y means x and y interleaved in any order
- `[x]` means zero or one times
- `x?` means x zero or one times
- `x*` means x zero or more times
- `x+` means x one or more times
- `(x)` means x (grouping)

The rhs of a symbol that starts with a lower-case letter implicitly allows white space and comments, as defined by the production `TokenWhiteSpace`, between the terminals and nonterminals that it references.

3. Program Structure

A Ballerina program is divided into modules. A module has a source form and a binary form. The module is the unit of compilation; a Ballerina compiler translates the source form of a module into its binary form. A module may reference other modules. When a compiler translates a source module into a binary module, it needs access only to the binary form of other modules referenced from the source module.

A binary module can only be referenced if it is placed in a module store. There are two kinds of module store: a repository and a project. A module stored in a repository can be referenced from any other module. A module stored in a project can only be referenced from other modules stored in the same project.

A repository organizes binary modules into a 3-level hierarchy:

1. organization;
2. module name;
3. version.

Organizations are identified by Unicode strings, and are unique within a repository. A module name is a Unicode string and is unique within a repository organization. A particular module name can have one or more versions each associated with a separate binary module. Versions are semantic, as described in the SemVer specification.

A project stores modules using a simpler single level hierarchy, in which the module is associated directly with the module name.

A binary module is a sequence of octets. Its format is specified in the Ballerina Platform Specification.

An abstract source module consists of:

- an unordered collection of one or more source parts; each source part is a sequence of octets that is the UTF-8 encoding of part of the source code for the module
- metadata containing the following
    - always required: module name
    - required only if the source module is to be compiled into a binary module stored in a repository:
        - organization name
        - version

An abstract source module can be stored in a variety of concrete forms. For example, the Ballerina Platform Specification describes a method for storing an abstract source module in a filesystem, where the source parts are files with a `.bal` extension stored in a directory, the module name comes from the name of that directory, and the version and organization name comes from a configuration file `Ballerina.toml` in that directory.

4. Lexical Structure

The grammar in this document specifies how a sequence of Unicode code points is interpreted as part of the source of a Ballerina module. A Ballerina module part is a sequence of octets (8-bit bytes); this sequence of octets is interpreted as the UTF-8 encoding of a sequence of code points and must comply with the requirements of RFC 3629.

After the sequence of octets is decoded from UTF-8, the following two transformations must be performed before it is parsed using the grammar in this document:

- if the sequence starts with a byte order mark (code point 0xFEFF), it must be removed
- newlines are normalized as follows:
    - the two character sequence 0xD 0xA is replaced by 0xA
    - a single 0xD character that is not followed by 0xD is replaced by 0xA

The sequence of code points must not contain any of the following disallowed code points:

- surrogates (0xD800 to 0xDFFF)
- non-characters (the 66 code points that Unicode designates as non-characters)
- C0 control characters (0x0 to 0x1F and 0x1F) other than whitespace (0x9, 0xA, 0xC, 0xD)
- C1 control characters (0x80 to 0x9F)

Note that the grammar notation ^X does not allow the above disallowed code points.

```
identifier := UndelimitedIdentifier | DelimitedIdentifier
UndelimitedIdentifier :=
   IdentifierInitialChar IdentifierFollowingChar*
DelimitedIdentifier := ^" StringChar+ "
IdentifierInitialChar := A .. Z | a .. z | _ | UnicodeIdentifierChar
IdentifierFollowingChar := IdentifierInitialChar | Digit
UnicodeIdentifierChar := ^ ( AsciiChar | UnicodeNonIdentifierChar )
AsciiChar := 0x0 .. 0x7F
UnicodeNonIdentifierChar :=
   UnicodePrivateUseChar
   | UnicodePatternWhiteSpaceChar
   | UnicodePatternSyntaxChar
UnicodePrivateUseChar :=
   0xE000 .. 0xF8FF
   | 0xF0000 .. 0xFFFFD
   | 0x100000 .. 0x10FFFD
```

```
UnicodePatternWhiteSpaceChar := 0x200E | 0x200F | 0x2028 | 0x2029
UnicodePatternSyntaxChar :=
   character with Unicode property Pattern_Syntax=True
Digit := 0 .. 9
```

Note that the set of characters allowed in identifiers follows the requirements of Unicode TR31 for immutable identifiers; the set of characters is immutable in the sense that it does not change between Unicode versions.

```
TokenWhiteSpace := (Comment | WhiteSpaceChar)*
Comment ::= // AnyCharButNewline*
AnyCharButNewline ::= ^ 0xA
WhiteSpaceChar := 0x9 | 0xA | 0xD | 0x20
```

`TokenWhiteSpace` is implicitly allowed on the right hand side of productions for non-terminals whose names start with a lower-case letter.

5. Values, Types and Variables

Ballerina programs operate on a rich universe of values. This universe of values is partitioned into a number of *basic types*; every value belongs to exactly one basic type. Values are of three kinds, each corresponding to a kind of basic type:

- simple values, like booleans and integers, which are not constructed from other values; simple values are always immutable
- structured values, like maps and arrays, which create structures from other values; most structured values are mutable
- behavioral values, like functions, which allow parts of Ballerina programs to be handled in a uniform way with other values

In Ballerina, types go beyond basic types. A type in Ballerina represents a set of values. Types are described in Ballerina using type descriptors. As well as type descriptors for each basic type, there are also type descriptors that create types from single values and from the union of two types. Thus a single value belongs to arbitrarily many different types, although it belongs to only one *basic* type.

Ballerina programs use types to declares the possible values that a variable may contain. The Ballerina compiler together with the runtime system ensures that variables only ever contain values belonging to the declared type.

Most types, including all simple basic types, have an implicit initial value, which is used to initialize a variable that does not have an explicit initializer. The declaration of a variable of a type that does not have an implicit initial value must have an explicit initializer.

The following table summarizes the type descriptors provided by Ballerina.

Kind	**Name**	**Set of values denoted by type descriptor**	**Implicit initial value**
basic, simple	nil	()	()
	boolean	true, false	false
	int	64-bit signed integers	0
	float	double precision IEEE 754 floating point numbers	+0.0
	string	sequences of Unicode code points	empty string
	blob	sequences of 8-bit bytes	empty blob
basic, structured	tuple	fixed length, immutable, list of values, where each member of the list has its own type	tuple where each member

			has the implicit initial value for its type
	array	dynamic length, mutable list of values, where each member of the list is specified with the same type	empty array
	map	a mutable mapping from keys, which are strings, to values; specifies maps in terms of a single type to which all keys are mapped	empty map
	record	a mutable mapping from keys, which are strings, to values; specifies maps in terms of names of fields (required keys) and value for each field	record where each field has the implicit initial value for its type
	table		a table with no rows
	XML	a sequence of zero or more characters, XML elements, processing instructions or comments	empty sequence
basic, behavioral	function	a function with 0 or more specified parameter types and a single return type	
	future		
	object		
	stream		
	typedesc	a type descriptor	a typedesc for ()
other	singleton	a single value described by a literal	the single value
	union	the union of the component types	() if that is part of the union
	optional	the underlying type and ()	()
	any	all values	()
	json	the union of (), int, float, string, and maps and arrays whose values are, recursively, json	()

Simple Values

A simple value belongs to exactly one of the following basic types:

- nil
- boolean
- int
- float
- string
- blob

The type descriptor for each simple basic type contains all the values of the basic type.

```
simple-type-descriptor :=
   nil-type-descriptor
   | boolean-type-descriptor
   | int-type-descriptor
   | float-type-descriptor
   | string-type-descriptor
   | blob-type-descriptor

literal :=
   nil-literal
   | boolean-literal
   | int-literal
   | float-literal
   | string-literal
   | blob-literal
```

Nil

```
nil-type-descriptor :=  ( )
nil-literal :=  ( ) | null
```

The nil type contains a single value, called nil, which is used to represent the absence of any other value. The nil value is written `()`. The nil value can also be written `null`, for compatibility with JSON; the use of null should be restricted to JSON-related contexts.

The nil type is special, in that it is the only basic type that consists of a single value. The type descriptor for the nil type is not written using a keyword, but is instead written `()` like the value.

The implicit initial value for the nil type is `()`.

Boolean

```
boolean-type-descriptor := boolean
boolean-literal := true | false
```

The boolean type consists of the values true and false.

The implicit initial value for the boolean type is false.

Int

```
int-type-descriptor := int
int-literal := DecimalNumber | HexIntLiteral
DecimalNumber := 0 | NonZeroDigit Digit*
HexIntLiteral := HexIndicator HexNumber
HexNumber := HexDigit+
HexIndicator := 0x | 0X
HexDigit := Digit | a .. f | A .. F
Digit := 0 .. 9
NonZeroDigit := 1 .. 9
```

The int type consists of integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (i.e. signed integers than can fit into 64 bits using a two's complement representation)

The implicit initial value for the int type is 0.

Floats

```
float-type-descriptor := float
float-literal := DecimalFloatNumber | HexFloatLiteral
DecimalFloatNumber :=
   DecimalNumber Exponent
   | DottedDecimalNumber [Exponent]
DottedDecimalNumber :=
   DecimalNumber . Digit*
   | . Digit+
Exponent := ExponentIndicator [Sign] Digit+
ExponentIndicator := e | E
HexFloatLiteral := HexIndicator HexFloatNumber
HexFloatNumber :=
   HexNumber HexExponent
   | DottedHexNumber [HexExponent]
DottedHexNumber :=
   HexDigit+ . HexDigit*
   | . HexDigit+
HexExponent := HexExponentIndicator [Sign] Digit+
HexExponentIndicator := p | P
Sign := + | -
```

The float type corresponds to IEEE 754 double precision floating point number.

As in IEEE 754, positive and negative zero are distinct values, although they are treated as numerically equal. However, the multiple bit patterns that IEEE 754 treats as NaN are considered to be the same value in Ballerina, although, following IEEE 754, two NaN values are treated as neither numerically equal nor unequal.

The implicit initial value for the float type is 0.0.

Strings

```
string-type-descriptor := string
string-literal :=
   DoubleQuotedStringLiteral | symbolic-string-literal
DoubleQuotedStringLiteral := " (StringChar | StringEscape)* "
symbolic-string-literal := ' UndelimitedIdentifier
StringChar := ^ ( 0xA | 0xD | \ | " )
StringEscape := StringSingleEscape | StringNumericEscape
StringSingleEscape := \t | \n | \r | \\ | \"
StringNumericEscape := \u[ CodePoint ]
CodePoint := HexDigit+
```

A string is an immutable sequences of zero or more Unicode code points. Any code point in the Unicode range of 0x0 to 0x10FFFF inclusive is allowed other than surrogates (0xD800 to 0xDFFF inclusive).

A `symbolic-string-literal` `'foo` is equivalent to a `double-quoted-string-literal` `"foo"`; this form of string literal is convenient when strings are used to represent an enumeration.

In a `StringNumericEscape`, `CodePoint` must valid Unicode code point; more precisely, it must be a hexadecimal numeral denoting an integer *n* where 0 <= *n* < 0xD800 or 0xDFFF < n <= 0x10FFFF.

The implicit initial value for the string type is an empty string.

Blobs

```
blob-type-descriptor := blob
blob-literal := Base16Literal | Base64Literal
Base16Literal := base16 WS ` HexGroup* WS `
HexGroup := WS HexDigit WS HexDigit
Base64Literal := base64 WS ` Base64Group* [PaddedBase64Group] WS `
Base64Group :=
   WS Base64Char WS Base64Char WS Base64Char WS Base64Char
PaddedBase64Group :=
   WS Base64Char WS Base64Char WS Base64Char WS PaddingChar
   | WS Base64Char WS Base64Char WS PaddingChar WS PaddingChar
Base64Char := A .. Z | a .. z | 0 .. 9 | + | /
```

```
PaddingChar := =
WS := WhiteSpaceChar*
```

A blob is an an immutable sequence of zero or more 8-bit bytes.

The implicit initial value for the blob type is an empty sequence.

Structured Values

A structured value belongs to exactly one of the following basic types:

- tuple
- array
- mapping
- table
- xml

A type descriptor for a basic structured type typically describes only a subset of the possible values of the type.

```
structured-type-descriptor :=
   tuple-type-descriptor
   | array-type-descriptor
   | mapping-type-descriptor
   | table-type-descriptor
   | xml-type-descriptor
```

Tuples

A tuple is a fixed length, immutable, list of two or more values. A tuple type is described by specifying the type for each member of the list.

```
tuple-type-descriptor :=
   ( type-descriptor (, type-descriptor)+ )
```

If the member types of a tuple type all have an implicit initial value, then the implicit initial value for the tuple type is the tuple of those implicit initial values; otherwise, the tuple type does not have an implicit initial value.

Tuples are the only structured values that are immutable.

Arrays

An array of type T is a mutable, ordered list of zero or more items of type T.

```
array-type-descriptor := type-descriptor [ ]
```

The number of items in the array is called the *length* of the array. A member of an array can be referenced by an integer index representing its position in the array. For an array of length *n*, the indices of the members of the array, from first to last, are 0,1,...,*n* - 1.

Attempting to read a member of an array using an out of bounds index will result in a runtime exception. Attempting to write a member of an array at an index *i* that is greater than or equal to the length of the array will first increase the length of the array to *i* + 1, with the newly added members of the array having the implicit initial value of T.

An array can be of type T for any T (including arrays), provided only that T has an implicit initial value. (When T does not have an implicit initial value, an array of T? may be used instead; see Optional Types.)

The implicit initial value of an array type is an array of length 0.

Mutability of arrays requires some additional complexity related to typing. The meaning of declaring a variable to be of type T[] is that the array referenced by the variable will always only contain values of type T. Arrays are thus covariant: if type T' is a superset of type T, then type T'[] is a superset of type T[]. For example, if it is true that an array x always contains only values of type int, it is necessarily also true that x will always contain only values of type int or type nil. (If you are a cat, then you are also a cat or a dog.)

In order to ensure that the typing cannot be subverted by mutation, an array value includes a type descriptor that constrains the values that can be stored into the array at runtime; this is called the *storage type* of the array. An attempt to store a value v into an array whose storage type is T will generate a runtime exception if v does not belong to T. The storage type of an array is immutable: it is fixed when the array is constructed. An array with storage type T belongs to type T[]; but it is also belongs to type T'[], for any type T' that is a superset of T. It is therefore possible that storing a value v of type T into an array that has type T[] may generate a runtime exception, because v may belong to T but not belong to the storage type of the array.

Mappings

A mapping value is a mutable mapping from keys, which are strings, to values. The type of mapping values can be described by two kinds of type descriptors.

```
mapping-type-descriptor :=
   map-type-descriptor | record-type-descriptor
```

Typing and mutability for mapping values is managed in a similar way to arrays. Types for mapping values say what can be read from the mapping. The types of keys and values that can be stored in a mapping value are constrained by an immutable storage descriptor that is part of the mapping value.

Map types

A map type-descriptor describes a type of mapping value by specifying the type that the value for all keys must belong to.

```
map-type-descriptor := map [< type-descriptor >]
```

The bare type descriptor `map` means a map that can contain any value. (It is thus equivalent to `map<any>`; see section XX.)

When a mapping constructor is evaluated and the type context is a type map<T>, then the constructed mapping value will have a storage context that allows only values of type T to be stored in the map.

The implicit initial value for a map type is an empty map.

Record types

A record type descriptor provides a more detailed way of describing a type of mapping value, which make mapping values convenient for use as records. When referring to mapping values described by record types, we use the term *field* to mean a key together with the value to which it is mapped. A record type specifies the names and types of the fields of a mapping value. Fields may be required or optional: a required field is always present in a record, whereas for an optional field the name and value may be omitted. Note that a field that is not present is not the same as a field that is present with a nil value. Records may may be closed, allowing only the fields names in the type-descriptor to occur, or open, allowing other fields as well.

```
record-type-descriptor :=
   record { field-descriptor+ [record-rest-descriptor] }
field-descriptor :=
   individual-field-descriptor | record-type-reference
individual-field-descriptor := type-descriptor field-name [?];
record-type-reference := * type-descriptor-reference ;
record-rest-descriptor := [type-descriptor] ... ;
field-name := identifier
```

Each `individual-field-descriptor` specifies a required or optional field; if `?` is present following the `field-name` then the field is optional; otherwise it is required; the `type-descriptor` specifies the type to which the value of the field must belong if it is present. The order of the `individual-field-descriptors` within a `record-type-descriptor` is not significant. Note that the delimited identifier syntax allows the field name to be any non-empty string.

A `record-type-reference` pulls in fields from a named record type. The `type-descriptor-reference` must reference a type defined by a `record-type-descriptor`. The field-descriptors from the referenced type are copied into the type being defined; the meaning is the same as if they had been specified explicitly.

If the `record-rest-descriptor` is present, then the record is open; otherwise it is closed. If it is open, then the record may contain a field with any name; if the field name is not specified in a field-descriptor that the value of the field must match the type in the record-rest-descriptor. If the type is omitted from the `record-rest-descriptor`, then a type of any is implied.

When a mapping constructor is evaluated and the type context is a record type, then the constructed mapping value will have a storage context that prevents any mutations of the keys and values of the mapping that would cause the mapping value to no longer belong to the record type.

Note that records are covariant with respect to the field types. Thus a closed record type `record { T1' f1; ... ; Tn' fn; }` is a subtype of type `record { T1 f1; ... ; Tn fn; }` if and only `Ti'` is a subtype of `Ti` for 0 <= `i` <= n.

If the types of the fields of a record type all have an implicit initial value, then the implicit initial value for the record type is a record where each required field is initialized to its implicit initial value; the implicit initial value does not include optional fields.

Tables

A table is intended to be similar to the table of relational database table. A table value contains an immutable set of column names and a mutable bag of rows. Each column name is a string; each row is a mapping that associates a value with every column name; a bag of rows is a collection of rows that is unordered and allows duplicates.

A table value also contains a boolean flag for each column name saying whether that column is a primary key for the table; this flag is immutable. If no columns have this flag, then the table does not have a primary key. Otherwise the value for all primary keys together must uniquely identify each row in the table; in other words, a table cannot have two rows where for every column marked as a primary key, that value of that column in both rows is the same.

```
table-type-descriptor := table { column-type-descriptor+ }
column-type-descriptor :=
   individual-column-type-descriptor
   | column-record-type-reference
individual-column-type-descriptor :=
   [key] type-descriptor column-name ;
column-record-type-reference :=
   * type-reference [key-specifier (, key-specifier)*] ;
key-specifier := key column-name
column-name := identifier
```

A table type descriptor has a descriptor for each column, which specifies the name of the column, whether that column is part of a primary key and the type that values in that column must belong to. If a column is part of a primary key, then the type descriptor for the column must not allow anything that is not allowed by the following rules:

- any simple value other than nil is allowed
- a tuple is allowed if all of its member types are allowed

Typing and mutability for table types is managed in a similar way to record types. A table value contains a type descriptor for its columns, giving the required storage type for each column. A table literal can explicitly specify its storage type by including a cast to an appropriate table type. Note that a table type T' will be a subtype of a table type T if and only if:

- T and T' have the same set of column names;
- T and T' have the same set of primary keys; and
- for each column, the type for that column in T' is a subtype of the type of that column in T.

The implicit initial value for a table is a table with no rows. The columns in the implicit initial value are determined by the type context in which the implicit initial value is used.

[Issues

- Need to say that unique constraint is part of the value.
- Do we want to allow values that are distinct but numerically equal e.g. -0.0 and +0.0 or 1.0 and 1 in primary key columns?
- Are there any other constraints on the values allowed in a table?

]

XML

```
xml-type-descriptor := xml
```

An XML value represents a sequence of zero or more of the kinds of things that can occur inside an XML element, specifically:

- elements
- characters
- processing instructions
- comments

XML values and their associated operations are described in section 11.

Behavioral Values

```
behavioral-type-descriptor :=
   function-type-descriptor
   | object-type-descriptor
   | future-type-descriptor
```

```
    | stream-type-descriptor
    | type-desc-type-descriptor
```

Functions

```
function-type-descriptor :=
   function ( param-list ) return-type-descriptor
return-type-descriptor := [ returns annos type-descriptor ]
param-list :=
   [ individual-param-list [, rest-param]]
   | rest-param
individual-param-list := individual-param (, individual-param)*
individual-param :=
   annos type-descriptor [param-name] [= default-value]
default-value := dflt-value-expr
dflt-value-expr :=
   nil-literal
   | boolean-literal
   | [Sign] int-literal
   | [Sign] float-literal
   | string-literal
   | blob-literal
   | ( dflt-value-expr (, dflt-value-expr)+ )
rest-param := type-descriptor ... [param-name]
```

A function is a part of a program that can be explicitly executed. In Ballerina, a function is also a value, implying that it can be stored in variables, and passed to or returned from functions.

When a function is executed, it is passed argument values as input and returns a value as output.

A function always returns exactly one value. A function that would in other programming languages not return a value is represented in Ballerina by a function returning `()`. (Note that the function definition does not have to explicitly return (); a return statement or falling of the end of the function body will implicitly return ().)

A function can be passed any number of arguments. Each argument is passed in one of three ways: as a positional argument, as a named argument or as a rest argument. A positional argument is identified by its position relative to other positional arguments. A named arguments is identified by name. Only one rest argument can be passed and its value must be an array.

A function definition defines a number of named parameters, which can be referenced like variables within the body of the function definition. The parameters of a function definition are of three kinds: required, defaultable and rest. The relative order of required parameters is significant, but not for defaultable parameters. There can be at most one rest parameter.

When a function is executed, the arguments are used to assign a value to each of the parameters. First, the required parameters are assigned values from the positional arguments in order. There must be at least as many positional arguments as required parameters. If there are more positional arguments than required parameters, then there must be a rest parameter and not be a rest argument, and an array of the remaining positional arguments is assigned to the rest parameter. Next, the defaultable parameters are assigned values from the named arguments. For each named argument, there must be a defaultable parameter with the same name. If there is no named argument for a defaultable parameter then the default value is assigned to that parameter. Finally, if there is a rest argument, there must be a rest parameter and the rest argument is assigned to the rest parameter.

The type system classifies function based on the arguments they are declared to accept and the values they are declared to return.

For return values, typing is straightforward: `returns T` means that the value returned by the function is always of type `T`. A function value declared as returning type T will belong to a function type declared as returning type T’ only if T is a subset of T’.

Objects

Objects are a combination of public and private data (records) along with a set of associated functions that can be used to manipulate them.

```
object-type-descriptor :=
   object {
     object-field-descriptor*
     [object-ctor-function]
     (member-function-decl | member-function-defn)*
   }

object-field-descriptor :=
   [visibility-qual] type-descriptor field-name ;
   | [visibility-qual] * type-descriptor-reference ;
```

The names of all the fields of an object must be distinct.

The `type-descriptor-reference` in an `object-field-descriptor` must reference a type defined by a `record-type-descriptor`. The field-descriptors from the referenced type are copied into the type being defined; the meaning is the same as if they had been specified explicitly; the visibility is determined for all copied field descriptors is determined by the `visibility-qual`.

If the type of every field of an object type has an implicit initial value, then the object type need not include an object initializer; otherwise, the object type must include an object constructor function. If the object type does not include an object constructor function, then the implicit initial value is a object with each field initialized to its implicit initial value. If the object type has an object constructor function, then the implicit initial value for the object type is the result of calling the constructor function with no arguments, if that would be a valid call; if it would not be a valid call, then the object type does not have an implicit initial value.

Constructor function

An object type descriptor may have a constructor function that will be invoked to create objects of this type.

```
object-ctor-function :=
   metadata?
   [public] new ( ctor-param-list ) { function-body }
   | [public] native new ( ctor-param-list ) ;
 ctor-param-list :=
   [ individual-ctor-param-list [, rest-param]]
   | rest-param
individual-ctor-param-list :=
   individual-param (, individual-ctor-param)*
individual-ctor-param :=
   [type-descriptor] [param-name] [= default-value]
```

Constructor function parameters are the same as function parameters except that the type descriptor may be omitted from a constructor function parameter, in which case the parameter name must be the same as a name of a field of the object, and the type of the parameter will be implied to be the same as that of the field. Each field with the same name as a parameter is initialized to the argument (or default value) for that parameter before the function body executes.

Within the function-body, the keyword `self` can be used to refer to the object.

Member Functions

Member functions are functions that are associated to the object and are accessed via a variable of that type using “`.function-name(..)`”. All the member functions of an object must be either declared or defined in the object.

A member-function-decl declares a member function without defining it; a member-function-defn also supplies a definition.

```
member-function-decl :=
   metadata?
   [visibility-qual]
   function function-name ( param-list ) return-type-descriptor ;
```

```
member-function-defn :=
   metadata?
   [visibility-qual]
   function function-name ( param-list ) return-type-descriptor
      { function-body }
function-name := identifier
```

A member function that is declared but not defined within an object must be defined outside the object at the top-level of the module:

```
external-member-function-defn :=
   function qual-function-name ( param-list ) return-type-descriptor
      { function-body }
qual-function-name := object-type-name.identifier
```

Visibility

```
visibility-qual := public | private;
```

Visibility of fields, object initializer and member functions is specified uniformly: `public` means that access is unrestricted; `private` means that access is restricted to the same object; if no visibility is specified explicitly, then access is restricted to the same module.

Visibility of an object-ctor-function cannot be specified to be `private`.

Visibility member functions is determined by the declaration in the object type descriptor; external member functions definitions therefore do not specify visibility.

Futures

```
future-type-descriptor := future [ < type-descriptor > ]
```

A future value represents a value to be returned by an asynchronous function invocation. The type-descriptor for future values specifies the type that will be returned; the type of an asynchronous invocation of a function of return type T will thus be `future<T>`.

A bare type `future` is equivalent to `future<any>`.

The future value supports a variety of operations relating to the asynchronously invoked function, including getting its status, waiting for it to complete, and getting its return value.

Streams

```
stream-type-descriptor := stream [  < type-descriptor > ]
```

A value of type `stream<T>` is a distributor for values of type `T`: when a value v of type T is put into the stream, a function will be called for each subscriber to the stream with v as an argument.

The implicit initial value of a stream type is a newly created stream, ready to distribute values.

A bare type `stream` is equivalent to `stream<any>`.

Type descriptor values

```
type-desc-type-descriptor := typedesc
```

A type descriptor value is a value representing a type descriptor. Referencing an identifier defined by a type definition in an expression context will result in a type descriptor value.

Type Descriptors

```
type-descriptor :=
   simple-type-descriptor
   | structured-type-descriptor
   | behavioral-type-system
   | singleton-type-descriptor
   | union-type-descriptor
   | optional-type-descriptor
   | any-type-descriptor
   | json-type-descriptor
   | error-type-descriptor
   | type-descriptor-reference
   | ( type-descriptor )
```

It is important to understand that the type descriptors specified in this section do not add to the universe of values. They are just adding new ways to describe subsets of this universe.

Singleton Types

```
singleton-type-descriptor := singleton-value-expr
singleton-value-expr :=
   nil-literal
   | boolean-literal
   | [Sign] int-literal
   | string-literal
   | blob-literal
   | singleton-tuple-expr
singleton-tuple-expr :=
   ( singleton-value-expr (, singleton-value-expr)+ )
```

A singleton type is a type whose value space consists of a single value. A singleton type is described using an expression for its single value. The expression is constrained to be a compile-time constant.

A single value from any simple type other than float is allowed as a singleton type. In addition, if all the member types of a tuple are allowed as a singleton type, then the tuple itself is also allowed as a singleton type.

The implicit initial value for a singleton type is its single value.

Union types

```
union-type-descriptor := type-descriptor | type-descriptor
```

The value space of a union type `T1|T2` is the union of `T1` and `T2`.

If () is in the value space of a union type, then the implicit initial value for the union type is (); otherwise the union type does not have an implicit initial value.

Optional types

```
optional-type-descriptor := type-descriptor ?
```

A type `T?` means T optionally, and is exactly equivalent to `T|()`.

Following the rules for the implicit initial value of union types implies that the implicit initial value for an optional type will always be ().

Any Type

```
any-type-descriptor := any
```

The type descriptor `any` describes a type that contains all values.

JSON types

```
json-type-descriptor := json [ < type-descriptor > ]
```

`json` means `() | int | float | string | json[] | map<json>`
`json<T>` means `record { *T; json ...; }`

Error type

```
error-type-descriptor := error
```

The type `error` is a built-in record type with the following type descriptor:

```
record { true isError; string message; error? cause; }
```

Variables

When a variable may be declared with an explicit type, or with an implicit type using `var`. When it is declared with an implicit type, the type of the variable is inferred from the static type of the expression on the right hand side. In this case, the broad type is used. See the “Static Typing of Expressions” section for more details.

6. Expressions

```
expression :=
   literal
   | constructor-expr
   | new-expr
   | string-template-expr
   | xml-expr
   | variable-reference-expr
   | field-access-expr
   | index-expr
   | xml-attributes-expr
   | function-call-expr
   | lambda-expr
   | table-query-expr
   | type-cast-expr
   | unary-expr
   | multiplicative-expr
   | additive-expr
   | relational-expr
   | equality-expr
   | logical-expr
   | conditional-expr
   | but-expr
   | await-expr
   | ( expression )
```

Static Typing of Expressions

Expressions have two static types, a precise type and a broad type. Usually, the precise type is used. However, in a few situations, using the precise type would be inconvenient, and so Ballerina uses the broad type. In particular, the broad type is used for inferring the type of an implicitly typed non-final variable. Similarly, the broad type is used when it is necessary to infer the storage type of a member of a mutable structure.

In most cases, the precise type and the broad type of an expression are the same. For a compound expression, the broad type of an expression is computed from the broad type of the sub-expressions in the same way as the precise type of the expression is computed from the precise type of sub-expressions. Therefore in most cases, there is no need to mention the distinction between precise and broad types.

The most important case where the precise type and the broad type are different is literals. For a literal other than a float literal, the precise type is a singleton type for the literal value, but the broad type is the basic type containing the literal value. (There are no singleton float types, so the precise and broad types for a float literal are both the basic type `float`.)

For a type cast expression, the precise type and the broad type are the type specified in the cast.

The detailed rules for the static typing of expressions are quite elaborate and are not specified completely in this document. This document only mentions some key points that programmers might need to be aware of.

Casting and Conversion

Ballerina makes a sharp distinction between type conversion and type casting.
Casting a value does not change the value. Any value always belongs to multiple types. Casting means taking a value that is statically known to be of one type, and using it in a context that requires another type; casting checks that the value is of that other type, but does not change the value.

Conversion is a process that takes as input a value of one type and produces as output a possibly distinct value of another type. Note that conversion does not mutate the input value.

Ballerina usually requires programmers to make conversions explicit. There is one exception: int values will be automatically converted to float values as necessary.

Constructors

```
constructor-expr  :=
   [ explicit-type-indicator ]
   (tuple-constructor-expr
    | array-constructor-expr
    | mapping-constructor-expr
    | table-constructor-expr)
explicit-type-indicator = < type-descriptor >
```

A `constructor-expr` may occur in a context that expects a particular type. If so, this may affect the value constructed. For example, the right hand side of an initializer of an variable expects the declared type of that variable if there is one. The expected type for a `constructor-expr` may be specified explicitly by using an `explicit-type-indicator`.

Tuple constructor

```
tuple-constructor-expr := ( expr-list )
expr-list = expression (, expression)+
```

Array constructor

```
array-constructor-expr := [ expr-list? ]
```

Mapping constructor

```
mapping-constructor-expr := { key-value-list? }
key-value-list := key-value-pair (, key-value-pair)
key-value-pair := key : value-expr
key := key-name | key-expr
key-name := identifier
key-expr := expression
value-expr := expression
```

In a key-value-pair, an identifier is treated as specifying the name of the key not as a variable reference (a variable reference can be enclosed in parentheses to prevent it from being interpreted as a key name).
The type expected by the context of the constructor determines the storage type of the constructed value. Otherwise, the storage type is inferred from the types of the expressions occurring in the constructor.

Table constructor

```
table-constructor-expr :=
   table { [column-descriptors [, table-rows]] }
column-descriptors := { column-descriptor (, column-descriptor)* }
column-descriptor := column-constraint* column-name
column-constraint :=
   key
   | unique
   | auto auto-kind
auto-kind := auto-kind-increment
auto-kind-increment := increment [(seed, increment)]
seed := integer
increment := integer
table-rows :=  [ table-row (, table-row)* ]
table-row := { expression (, expression)* }
```

For example,

```
table  {
  { key firstName, key lastName, position },
  [
    {"Sanjiva", "Weerawarana", "lead" },
    {"James", "Clark", "design co-lead" }
  ]
}
```

New expression

```
new-expr :=
   new [( func-args )]
   | new type-descriptor ( func-args )
```

The new expression may be used to create new instances of objects, records or streams. If the type descriptor is omitted then the type is inferred from the context, which may be a variable declaration or a function argument.

The arguments may be given only when constructing an object which has a constructor with the corresponding signature.

Note that use of new is always optional for streams as their have an initial default value. It is also optional for record and object types that have initial default values.

String template expression

```
string-template-expr := string StringTemplate
StringTemplate := ` StringTemplateFrag* `
StringTemplateFrag :=
   StringChar
   | StringEscape
   | \{
   | string-template-expr-frag
string-template-expr-frag := {{ expression }}
```

String templates allow the construction of a string by embedding literal characters and expressions. Each embedded expression must evaluate to a string value or must be automatically converted to a string. The result of a string template is the string consisting of the concatenation of all characters outside of the embedded expressions along with the string values of the expressions in the order they are written within the template.

XML expression

An XML expression creates an XML value.

```
xml-expr := xml XmlExpr
XmlExpr := `XmlContentItem*`
```

Variable reference expression

```
variable-reference-expr := variable-reference
variable-reference := [identifier :] identifier
```

Field access expression

```
field-access-expr := expression . field-name
field-name := identifier
```

Index expression

```
index-expr := expr [ expr ]
```

XML attributes expression

```
xml-attributes-expr := expr @
```

Function call expression

```
function-call-expr := func-name ( func-args? )
func-name := variable-reference-expr | field-access-expr
func-args := func-arg (, func-arg)* (, rest-arg?)
func-arg := expression | (identifier = expression)
rest-arg := ... expression
```

Lambda expression

```
lambda-expr :=
   ( param-list ) => [type-descriptor] { function-body }
```

Table query expressions

```
table-query-expr :=
   from query-source [query-join-type query-join-source]
      [query-select] [query-group-by] [query-order-by]
      [query-having] [query-limit]
query-source := identifier [as identifier] [query-where]
query-where := where expression
query-join-type := [([left | right | full] outer)| inner] join
query-join-source := query-source on expression
query-select := select (* | query-select-list)
query-select-list :=
   expression [as identifier] (, expression [as identifier])*
query-group-by := group by identifier (, identifier)*
query-order-by :=
   order by identifier [(ascending | descending)]
      (, identifier [(ascending | descending)])*
query-having := having expression
query-limit := limit int-literal
```

Query expressions being language integrated SQL-like querying to Ballerina tables. See section 10 for details.

Type cast expression

```
type-cast-expr := < type-descriptor > expression
```

Unary expression

```
unary-expr :=
   + expression
   | - expression
   | ! expression
   | lengthof expression
   | untaint expression
```

Multiplicative expression

```
multiplicative-expr :=
   expression * expression
   | expression ^ expression
   | expression / expression
   | expression % expression
```

Additive expression

```
additive-expr :=
   expression + expression
   | expression - expression
```

Relational expression

```
relational-expr :=
   expression < expression
   | expression > expression
   | expression <= expression
   | expression >= expression
```

Equality expression

```
equality-expr :=
   expression == expression
   | expression != expression
```

Logical expression

```
logical-expr :=
   expression && expression
   | expression || expression
```

Conditional expression

```
conditional-expr :=
   expression ? expression : expression
   | expression ?: expression
```

L `?:` R is evaluated as follows:

1. Evaluate L to get a value x
2. If x is not nil, then return x.
3. Otherwise, return the result of evaluating R.

But expression

```
but-expr := expression but { but-expr-clause (, but-expr-clause)* }
but-expr-clause := type-descriptor identifier? => expression
```

Await expression

```
await-expr := await expression
```

The expression must be of type future. Await blocks the calling worker until the function represented by the future completes or results in an error. Thus, the type of this expression is T | error, where T is the return type of the function.

7. Statements

```
statement :=
   var-decl-stmt
   | xmlns-decl-stmt
   | assignment-stmt
   | compound-assignment-stmt
   | destructuring-assignment-stmt
```

```
    | post-arithmetic-stmt
    | check-stmt
    | function-call-stmt
    | action-invocation-stmt
    | if-else-stmt
    | match-stmt
    | iteration-stmt
    | fork-stmt
    | try-stmt
    | throw-stmt
    | transaction-stmt
    | lock-stmt
    | worker-interaction-stmt
    | forever-stmt
    | return-stmt
    | done-stmt

block-stmt := { statement* }
```

Declaration statements

```
var-decl-stmt :=
   explicit-var-decl-stmt
   | implicit-var-decl-stmt
   | implicit-tuple-destruct-decl-stmt

explicit-var-decl-stmt :=
   annos
   type-descriptor identifier [= expression ;]
```

The explicit-type-decl-stmt declares a variable of the indicated type. The variable will have the implicit initial value for the type descriptor. If the type does not have have an implicit initial value then an initial value must be provided.

```
implicit-var-decl-stmt :=
   annos
   var identifier = expression ;
```

The implicit-type-decl-stmt declares a variable whose type is derived from the type of the expression and assigns it the value of the expression.

```
implicit-tuple-destruct-decl-stmt :=
   annos
   var ( id-or-ign, id-or-ign (, id-or-ign)* ) = expression ;
id-or-ign := identifier | _
```

The implicit-tuple-destruct-decl-stmt declares a group of variables to capture the de-structured parts of a tuple typed expression. The right hand side expression must be of a tuple type with the same number of parts as there are identifiers in the left hand side. The type of each declared variable is the type of the corresponding part of the tuple. If the identifier is given as “_” then that part is ignored.

XML namespace declaration statement

```
xmlns-decl-stmt := xmlns string-literal [ as identifier ] ;
```

The xml-decl-stmt is used to declare a XML namespace. If the identifier is omitted then the default namespace is defined. Once a default namespace is defined, it is applicable for all XML values in the current scope. If an identifier is provided then that identifier is the namespace prefix used to qualify elements and/or attribute names.

Assignment statement

```
assignment-stmt := lhs = expression ;
lhs :=
   variable-reference
   | lhs . field-name
   | lhs [ expression ]
   | lhs @
```

An lhs evaluates to a reference. A reference is one of

- a variable
- a mapping value plus a string key
- an object plus a string key
- an array plus an integer index i with 0 <= i < length

Operations on references:

- store a value
- get a value

An lhs is evaluated for a particular usage type, which is one of

- assignment
- keyed access
- indexed access

L = R is evaluated as follows:

1. L is evaluated for assignment to give a reference r
2. R is evaluated to give a value v
3. Value v is stored in the reference r

L.x is evaluated as follows:

1. L is evaluated for keyed access to give a reference r

2. r is dereferenced to give a value v, which must be a mapping
3. if L.x is being evaluated for keyed access or indexed access, then if v does not have a field x or the value of field x is nil, then a new value is stored in field x, where the new value is the implicit initial value of the intersection of T1 and T2, where T1 is the storage type of field x of v, and T2 is map<any> for keyed access and any[] for indexed access
4. result is reference to field x of v

L[E] is evaluated as follows:

1. E is evaluated to give a value x
2. if x is a string, then evaluation proceeds as for L.x; otherwise x must be a non-negative int, and evaluation proceeds as follows
3. L is evaluated for indexed access to give a reference r
4. r is dereferenced to give a value v, which must be an array
5. the length of v is increased if necessary so that the length of v is greater than x
6. if L[E] is being evaluated for keyed access or indexed access, then if value stored at index x of v is nil, then a new value is stored at index x of v, where the new value is the implicit initial value of the intersection of T1 and T2, where T1 is the storage type of v, and T2 is map<any> for keyed access and any[] for indexed access
7. result is reference to index x of v

Compound assignment statement

```
compound-assignment-stmt :=
   lhs += expression ;
   | lhs -= expression ;
   | lhs *= expression ;
   | lhr /= expression ;
```

These statements update the value of the LHS variable to the value that results from applying the corresponding binary operator to the value of the variable and the value of the expression.

Destructuring assignment statement

```
destructure-assignment-stmt := structure-pattern = expr
structure-pattern :=
   tuple-structure-pattern
   | record-structure-pattern
tuple-structure-pattern :=
  ( inner-structure-pattern (, inner-structure-pattern)+ )
record-structure-pattern :=
  { key-value-pattern (, key-value-pattern)* }
key-value-pattern := field-name : inner-structure-pattern
inner-structure-pattern := variable-reference | structure-pattern
```

Postfix arithmetic statement

```
post-arithmetic-stmt := single-lhs post-op ;
post-op := ++ | --
```

The post increment operator (++) adds one to the value of the variable and the post decrement operator (--) subtracts one from the value.

Check statement

```
check-stmt := checked-assignment-stmt | checked-function-call-stmt
checked-assignment-stmt := lhs = check expression ;
checked-function-call-stmt := check function-call-exp ;
```

The check construct is used to to remove the error type from an expression: if type of expression is T|error, then the result is of type T. If an error occurs, at runtime check forces an immediate return from the containing function. If the containing function is not declared to be returning an error then the error is thrown as an exception. It is a compile-time error if the type of expression does not include error.

Function call statement

```
function-call-stmt := function-call-exp ;
```

Action invocation statement

```
action-invocation-stmt :=
   [lhs =] variable-reference -> identifier ( func-args? ) ;
```

If-else statement

```
if-else-stmt :=
   if ( logical-expr ) block-stmt
   [ else if ( logical-expr ) block-stmt ]*
   [ else block-stmt ]
```

The if-else statement is used for conditional execution.

When a logical expression is true then the corresponding block statement is executed and the if statement completes. If no statement is true then, if the else block is present, the corresponding block statement is executed.

Match statement

```
match-stmt := match expression { match-pattern-clause + }
```

```
match-pattern-clause :=
   type-descriptor [ identifier ] => statement | block-stmt
```

A match statement is a type switching construct that allows selective code execution based on the type of the expression that is being tested.

The expression is first evaluated. Then the resulting value is tested sequentially against the type descriptors to find the first match and the corresponding statement is executed. The match will be successful if the value is a member of the set of values that the type descriptor identifies.

If an identifier is given in the match-pattern-clause then that is of the corresponding type descriptor and its value will bound to the value of the expression for the scope of the statement or block statement that will be executed.

The compiler will validate that the clauses are exhaustive for the possible types of the expression.

Iteration statements

```
iteration-stmt := foreach-stmt | while-stmt
loop-body-stmt := { (next-stmt | break-stmt | statement)* }
```

Foreach statement

```
foreach-stmt := range-foreach-stmt | collection-foreach-stmt
range-foreach-stmt :=
   foreach identifier in int-range-expr loop-body-stmt
collection-foreach-stmt :=
   foreach identifier [, identifier] in expression loop-body-stmt
int-range-expr := ([|() expression .. expression ()|])
```

A foreach statement is used to iteratively execute a block of statements.

In the first version, the block statement is executed with the identified bound to each integer value starting from the first expression’s value to the second expression’s value, inclusively or exclusively depending on whether square brackets or parentheses are used, respectively.

In the second version, the block statement is executed with the identifiers bound iteratively to the values from the collection. If two identifiers are provided the first one is the 0-based index of the items in the collection and the second is the i-th value in the collection when the first variable’s value is i. If only one variable is provided then it is bound to each value in the collection iteratively prior to executing the block statement.

If a break statement is executed within the loop then the loop immediately completes skipping any remaining iterations. If a next statement is executed then any subsequent statements are skipped, the loop counters updated and the loop starts again at the beginning of the loop body.

The table below indicates the types and values of the identifiers for different types of build-in iterable structured values:

Kind of structured value	When one identifier is provided	When two identifiers are provided	
		1st identifier	2nd identifier
array	Type: array's element type Value: current element	Type: int Value: index of the current element	Type: array's element type Value: current element
map	Type: map's element type Value: current map value	Type: string Value: key of the current map value	Type: map's element type Value: current map value
table	Type: record type of a table row	Type: int Value: index of the current row of the table	Type: record type of a table row Value: current row
XML	Type : xml Value : current XML value	Type : int Value : index of the current XML value	Type : xml Value : current XML value

While statement

```
while-stmt := while logical-expr loop-body-stmt
```

A while statement is used to execute a block of statements in a loop as long as a logical condition is true.

Next statement

```
next-stmt := next ;
```

A next statement is used to jump to the top of a looping construct (a foreach or while statement) skipping any statements that come after this. If this statement is placed unconditionally then the compiler will refuse to compile if there are any statements following this as they will not be executed.

Break statement

```
break-stmt := break ;
```

A break statement is used to immediately jump out of a looping construct (a foreach or while statement) and execute the statement immediately following the looping construct statement.

Fork statement

```
fork-stmt := fork { worker-decl+ } join-clause? join-timeout?
join-clause := join [ join-condition ] join-action
join-condition := join-all | join-some
join-all := all [ identifier (, identifier)* ]
join-some := some int-literal [ identifier (, identifier)* ]
join-action := ( map<any?> identifier ) block-stmt
join-timeout := timeout expression join-action
```

A fork statement is used to “fork” the current worker into multiple parallel workers and then wait for them to complete their work, subject to certain conditions. The fork statement completes when the join condition has been satisfied and all the workers have completed their work. Workers communicate their results back to the fork by sending data to the fork.

The join-clause is used to describe under what conditions the forked workers are to be considered as having completed their work. The “all” condition is used to require that either all the workers, or all the named workers, must complete. The “some” condition is used to require either k, where k is an integer value between 1 and the number of workers in the fork. If any workers are named then either they must all complete (in the case of all) or some number of those named workers must complete (in the case of some). When the join condition is satisfied all remaining workers are terminated upon completion of the current instruction.

Workers can send their results to the fork by sending the data by treating “fork” as the name of a worker (see Worker interaction statements). Any single value can be sent.

Within the join action, the results from the workers are delivered in a map whose type is optional and sufficiently wide to capture all the values that the workers return. The name of the worker serves as the key to this map - if the worker completed and returned a value then it there would be a value present corresponding to the key.

The timeout clause is used to provide an upper bound on how long the fork has to complete its action before its aborted. The expression must provide an integer value which is interpreted as a time in milliseconds. If the time runs out then all workers are dismissed and the timeout clause executed with the results of any workers that may have completed offering their results similar to the join action.

Try statement

```
try-stmt := try block-stmt try-stmt-catch* try-stmt-finally?
try-stmt-catch := catch ( type-descriptor identifier ) block-stmt
try-stmt-finally := finally block-stmt
```

The try statement is used to execute a block of code and catch and process any exceptions that occur within that block.

If an exception does not occur during the execution of the block then, if there is a finally block that executes prior to the try completing, else the try completes.

The type of the catch parameter must be assignable to the error type.

If an exception occurs, then the resulting exception is matched against the catch clause error types one by one until a matching one is found. If it is found the corresponding block statement is executed. If none is found then the exception gets thrown out of the current worker after processing any finally block.

If a finally block is present then that block is executed whether an exceptions occurs or not, or even if it occurs but does not match any of the catch statements and the exception propagates further.

Throw statement

```
throw-stmt := throw expression ;
```

A throw statement is use to throw any value which is assignable to error as an exception. If the exception is not handled by an enclosing try statement the enclosing worker dies with this exception.

As this statement completes the current worker, no other statement can follow this statement.

Transaction statement

```
transaction-stmt := transaction trans-conf? trans-body trans-retry?
trans-conf := trans-conf-item (, trans-conf-item)*
trans-conf-item := trans-retries | trans-oncommit | trans-onabort
trans-retries := retries = expression
trans-oncommit := oncommit = identifier
trans-onabort := onabort = identifier
trans-retry := onretry block-stmt
trans-body := { (retry-stmt | abort-stmt | statement)* }
retry-stmt := retry ;
abort-stmt := abort ;
```

A transaction statement is used to execute a block of code within a 2PC transaction. A transaction can be established by this statement or it may inherit one from the current worker.

Initiated transactions

If no transaction context is present in the worker then the transaction statement starts a new transaction (i.e., becomes the initiator) and executes the statements within the transaction statement.

Upon completion of the block the transaction is immediately tried to be committed. If the commit succeeds, then if there’s an on-commit handler registered that function gets invoked to signal that the commit succeeded. If the commit fails, and if the transaction has not been retried more times than the value of the retries configuration, then the on-retry block is executed and the transaction block statement will execute again in its entirety. If there are no more retries available then the commit is aborted the on-abort function is called.

The transaction can also be explicitly aborted using an abort statement, which will call the on-abort function and give up the transaction (without retrying).

If a retry statement is executed if the transaction has not been retried more times than the value of the retries configuration, then the on-retry block is executed and the transaction block statement will execute again in its entirety.

Participated transactions

If a transaction context is present in the executing worker context, then the transaction statement joins that transaction and becomes a participant of that existing transaction. In this case, retries will not occur as the transaction is under the control of the initiator. Further, if the transaction is locally aborted (by using the abort statement), the transaction gets marked for abort and the participant will fail the transaction when it is asked to prepare for commit by the coordinator of the initiator. When the initiating coordinator decides to abort the transaction it will notify all the participants globally and their on-abort functions will be invoked. If the initiating coordinator decides to retry the transaction then a new transaction is created and the process starts with the entire containing executable entity (i.e. resource or function) being re-invoked with the new transaction context.

When the transaction statement reaches the end of the block the transaction is marked as ready to commit. The actual commit will happen when the coordinator sends a commit message to the participant and after the commit occurs the on-commit function will be invoked. Thus, reaching the end of the transaction statement and going past does not have the semantic of the transaction being committed nor of it being aborted. Thus, if statements that follow the transaction statement they are unaware whether the transaction has committed or aborted.

When in a participating transaction, a retry statement is a no-op.

Transaction propagation

The transaction context in a worker is always visible to invoked functions. Thus any function invoked within a transaction, which has a transaction statement within it, will behave according to the “participated transactions” semantics above.

The transaction context is also propagated over the network via the Ballerina Microtransaction Protocol [XXX].

Lock statement

```
lock-stmt := lock block-stmt
```

A lock statement is used to execute a series of assignment statements in a serialized manner. For each variable that is used as an L-value within the block statement, this statement attempts to first acquire a lock and the entire statement executes only after acquiring all the locks. If a lock acquisition fails after some have already been acquired then all acquired locks are released and the process starts again.

Worker interaction statements

```
worker-interaction-stmt := worker-send | worker-receive
worker-send := expression -> (identifier | fork) ;
worker-receive := lhs <- identifier ;
```

Forever statement

```
forever-stmt :=
   forever {
      streaming-query-pattern+
   }

streaming-query-pattern :=
   streaming-query-expr => ( array-type-descriptor identifier )
      block-stmt
streaming-query-expr :=
   from (sq-source [query-join-type sq-join-source]) | sq-pattern
      [query-select] [query-group-by] [query-order-by]
      [query-having] [query-limit]
      [sq-output-rate-limiting]
sq-source :=
   identifier [query-where] [sq-window [query-where]]
      [as identifier]*
sq-window := window function-call-exp
sq-join-source := sq-source on expression
```

```
sq-output-rate-limiting :=
   sq-time-or-event-output | sq-snapshot-output
sq-time-or-event-output :=
   (all | last | first) every int-literal (time-scale | events)
sq-snapshot-output :=
   snapshot every int-literal time-scale
time-scale := seconds | minutes | hours | days | months | years
sq-pattern := [every] sp-input [sp-within-clause]
sp-within-clause := within expression
sp-input :=
   sp-edge-input (followed by) | , streaming-pattern-input
   | not sp-edge-input (and sp-edge-input) | (for simple-literal)
   | [sp-edge-input ( and | or ) ] sp-edge-input
   | ( sp-input )
sp-edge-input :=
   identifier [query-where] [int-range-expr] [as identifier]
```

The forever statement is used to execute a set of streaming queries against some number of streams concurrently and to execute a block of code when a pattern matches. The statement will never complete and therefore the worker containing it will never complete. See section 10 for details.

Return statement

```
return-stmt := return [ expression ] ;
```

A return statement is used to mark the completion of the worker that contains it and further that the function should considered as completed. See section 9 for more on the function invocation protocol.

Done statement

```
done-stmt := done ;
```

A done statement is used to mark the completion of the worker that contains it. Using done is equivalent to the worker completing its work at the end (“falling off the end”) and does not imply that the containing function has completed.

8. Module-level declarations

Each source part in a Ballerina module must match the production `module-part`.

```
module-part := import-decl* other-decl*
other-decl := metadata? other-decl-item
other-decl-item :=
   type-def-stmt
   | module-var-decl
   | module-endpoint-decl
   | function-def
   | service-def
   | xmlns-decl-stmt
   | annotation-decl
```

Import declarations

```
import-declaration :=
   import [org-name /] pkg-name [version sem-ver] [as identifier] ;
org-name := identifier
pkg-name := identifier (. identifier)*
sem-ver := major-num [. minor-num [. patch-num]]
major-num := DecimalNumber
minor-num := DecimalNumber
patch-num := DecimalNumber
```

Type definitions

```
type-definition :=
   metadata?
   public? type identifier type-descriptor ;
```

Module variables

```
module-var-decl :=
   metadata?
   public? var-decl-stmt
```

Module endpoints

```
module-endpoint-decl :=
   metadata?
   public? endpoint-decl
endpoint-decl := endpoint type-descriptor identifier
      mapping-constructor-expr? ;
```

Functions

```
function-def :=
   metadata?
   public? function identifier ( param-list ) return-type-descriptor
      { function-body }
function-body := endpoint-decl* (statement* | worker-decl*)
```

```
worker-decl := worker identifier block-stmt
```

A function is always a collection of workers, where each worker is a single actor of the sequence diagram of the function. The syntax of providing a set of statements instead of a worker is purely a short-circuit syntax for writing those statements in a single worker.

Services

```
service-def :=
   metadata?
   service [< type-descriptor >] identifier svc-bind? {
      svc-body-decl
      resource-def*
   }
svc-bind := bind (identifier | mapping-constructor-expr)
svc-body-decl := (endpoint-decl | var-decl-stmt | xmlns-decl-stmt)*
resource-def :=
   metadata?
   identifier ( param-list ) {
      function-body
   }
```

9. Evaluation Model

A Ballerina program consists of a set of functions and services. A function named “main” and services bound to a service endpoint are the entry points of a Ballerina program.

When a Ballerina program starts, all the modules are initialized and, if present, the main function (see below) is executed. If module initialization fails then the program does not start. During module initialization all module level final variables acquire their values in the necessary order to satisfy dependency requirements, all client and service endpoints are initialized, and all services statically attached to service endpoints are started.

If a main function was executed, the program will halt if there are no services attached to any service endpoints. If there are, the main function will complete and the program will not halt until one of the services explicitly halts the program.

Functions

A function is the unit of execution in Ballerina. A function is designed as a sequence diagram where the actors of the sequence diagram are either workers or client endpoints. Invoking the function starts all the workers concurrently and the function will be considered completed when one of the workers signals the completion of the function.

Workers

A worker is a sequence of statements that is executed concurrently with all other workers in the function. Workers follow lexical scoping rules and have access to the function's parameters and any visible external symbols but do not share anything else across each other. Workers within a function may communicate with each other via worker-to-worker messaging.

Workers can also interact with client endpoints via actions, which are the callable entry points of an endpoint.

Client endpoints

Client endpoints represent external systems that one or more of the workers interacts with. Thus, the programmer does not code what the external system does within the endpoint; it merely represents such execution as actions which may be invoked by one of the workers. An action represents a single capability or functionality that is offered to its consumers by an endpoint. An action may or may not provide a response to the caller directly in response to the request.

Function invocation protocol

Any function in Ballerina can be invoked in a blocking or non-blocking fashion. Blocking invocation pauses the calling worker until the function completes execution. Non-blocking calls return immediately after initiating the function execution with a future which can be interrogated to check the status of the invocation as well as to await the completion of the called function.

Due to a function potentially containing multiple workers, function invocation in Ballerina is not as straightforward as traditional call-stack style invocation.

When a function is invoked, prior to starting all of its workers, all endpoints declared within a function are initialized, in the order of their declaration. If the initialization fails, the function invocation itself will fail with an exception thrown to the caller.

If the function was invoked in a non-blocking fashion, then the function initiation is considered complete when the endpoints are initialized.

Once the endpoints are initialized, all workers are started concurrently. All workers have read access to all the parameters of the function.

A worker can complete its execution in one of three ways:

1. by executing a "return" statement, or
2. by executing a "done" statement or by executing the last statement of the worker, or
3. by an uncaught exception terminating the worker.

The first worker to execute a return statement will cause the function to be considered as completed, resulting in the caller being unblocked. Similarly, for functions that do not return any value, the first worker to execute a done statement or complete its execution will mark the function as completed.

The remaining workers, if any, will continue to execute unhindered by the caller continuing on.

If a worker executes a return statement after the function has already been "completed" then the worker will terminate with an exception. If a worker executes the last statement of its code block or executes a done statement, it is considered to have completed the work without affecting the waiting invoker of the function. Or, in the case of function that does not return anything, all subsequent worker completions, after the first one completes, are inconsequential to the caller.

For functions that return a value, if all the workers complete their work (normally or abnormally) without any worker executing a return statement, then the function is considered to have failed and a call failed error is thrown to the invoker as an exception. Within the call failed exception will be an array of errors providing the errors that caused workers within the function to complete abnormally. Thus, exceptions within a function are propagated to the caller, but as part of the call failed error and not directly.

The "main" function

A function named "main" in any module is a possible entry point to a Ballerina program. A main function is invoked when a Ballerina process is started indicating the name of the module to run or a source file that contains the function. Whether the program terminates upon completion of the main function depends on whether there are any services running (see the top of this section).

Services

Services are attached to service endpoints and is the network entrypoint to a running Ballerina program.

Service endpoints

A service endpoint is the access point for networked interactions. Services are registered at the service endpoint with any information necessary for the endpoint to correctly dispatch messages to a particular resource within a particular service attached to it.

Service endpoints may be passive (i.e., it opens a port and waits for connections and messages) or active (i.e. it open a connection to some remote system and polls it).

Service endpoints are typically declared outside of any functions and are initialized at program startup during module initialization. Service endpoints may also be created programmatically and services may be attached dynamically, but since service endpoints are open doors for networked calls into a running Ballerina program, their lifetime is not affiliated to the scope in which they are created.

Resources

A service consists of a set of resources, which are network callable entry points accessed through a service endpoint. A resource is invoked by the service endpoint when it decides that a resource is appropriate processor for a message received at the endpoint.

A resource execution is like a function execution, except that it is not allowed to return anything. If the resource wishes to respond to the request it must do so using the endpoint that is given to it at the point of invocation. If the resource’s initiatialization fails (see function invocation) then the service endpoint must perform the appropriate action for the underlying network interaction (e.g., close the connection or send an error).

Lifetime and state

A service may be instantiated with different lifetimes - everything from a singleton to one instance per request. The lifetime of a service is asserted or requested by the service at the time it is attached to a service endpoint, but the final decision is up to the service endpoint that the service attaches to.

Services are effectively object instances; thus any variables declared with in the service behave similarly subject to their lifetime being controlled by the service endpoint.

Checkpointing / restarting and compensation

TBC.

10. Table and Stream Queries

Ballerina tables and streams are designed for processing data at rest and data in motion, respectively.

WIP.

11. XML

Details about XML support.

```
XmlContentItem :=
  XmlElement
  | XmlChar
  | XmlComment
  | XmlProcessingInstruction
  | xml-expr-item
```

```
xml-expr-item := {{ expression }}
XmlElement :=
  XmlStartTag XmlContentItem* XmlEndTag
  | XmlEmptyElement
```

12. Metadata

```
metadata := (documentation? & deprecated?) annos
```

Annotations

Ballerina uses annotations to provide additional metadata about a particular construct. Annotations can be attached at various levels. The type of the annotation must be a record type and defines the configuration data that can be given when an annotation is attached to a particular construct.

```
annotation-decl :=
   metadata?
   public? annotation [<anno-attach-point (, anno-attach-point)*>]
      identifier type-descriptor ;
anno-attach-point :=
   service
   | resource
   | function
   | object
   | type
   | endpoint
   | parameter

annos := annotation*
annotation :=
   @ variable-reference mapping-constructor-exp?
```

Documentation

```
documentation := documentation { DocTemplateFrag* }
DocTemplateFrag :=
   StringChar
   | StringEscape
   | \{
   | doc-template-identifier-frag
doc-template-identifier-frag := {{ identifier }}
```

The documentation statement is used to document various Ballerina constructs.

```
deprecated := deprecated { (DeprStringChar | DeprStringEscape)* }
DeprStringChar := ^ ( \ | } )
DeprStringEscape := \\ | \}
```

The deprecated statement is used to provide documentation about a symbol that has been deprecated since being previously published. The content is expected to be in GitHub Markdown syntax [XXX] and will be processed by the documentation generator to produce formatted output.

13. Security

- Tainting; protection against tainting (protected annotation and untaint expression)
- Authentication / authzn architecture

14. Distributed Resilience

- Distributed transactions
- Forward recovery (continue from last checkpoint)
- Microtransaction protocol
- Network interactions: circuit-breaking, retry, failover, load balancing, ..
- Security token propagation

A. References

- Unicode
- XML
- JSON
- RFC 3629 UTF-8
- IEEE 754
- Markdown

B. Future Work

This section provides a non-exhaustive list of pending changes to the language.

1. Provide syntax for reading the i-th component of a tuple without having to destructure the entire tuple and ignoring the other components.
2. Compensating transactions
3. Refined types: allow a type to be qualified by a predicate e.g. `type PositiveInt int where value > 0` (enforcing this at compile time is difficult but possible)
4. Incorporate privacy aware coding concepts