Ballerina Language Specification, v0.980, Working Draft, 2018-07-12

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 being refined. Its specification in this document is incomplete. Furthermore, the reference implementation of the language is not yet completely aligned with this specification.

Sections 1 to 6 of this document are approaching completion; some later sections are yet to be written.

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. It is intended to be a pragmatic language suitable for mass-market commercial adoption; it tries to feel familiar to programmers who are used to popular, modern C-family languages, notably Java, C# JavaScript.

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, Perl 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.

Values can be stored in variables or as members of structures. A simple value is stored directly in the variable or structure. However, for other types of value, what is stored in the variable or member is a reference to the value; the value itself has its own separate storage. Non-simple types (i.e. structured types and behavioral types) are thus collectively called reference types.

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.

Most basic types structured values (along with one simple value) are iterable, meaning that a value of the type can be accessed as a sequence of simpler values.

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	64-bit IEEE 754-2008 binary floating point numbers	+0.0
	decimal	128-bit IEEE 754-2008 decimal floating point numbers	+0.0
	string	sequences of Unicode code points	empty string
basic, structured	array	an ordered list of values, optionally with a specific length, where a single type is specified for all members of the list,	empty array, or array of the specified length where each member has the implicit initial value for its type
	tuple	an ordered list of values, where a type is specified separately for each member of the list	tuple where each member has the implicit initial value for its type
	map	a mapping from keys, which are strings, to values; specifies mappings in terms of a single type to which all keys are mapped	empty map
	record	a 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
	error	an indication that there has been an error, with a string identifying the reason for the error, and a mapping giving additional details about the error	
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	()
	byte	int in the range 0 to 255 inclusive	0
	json	the union of (), int, float, decimal, 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
- decimal
- string

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
   | floating-point-type-descriptor
   | string-type-descriptor
```

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.

Floating point types

```
floating-point-type-descriptor := float | decimal
floating-point-literal :=
   DecimalFloatingPointNumber | HexFloatingPointLiteral
DecimalFloatingPointNumber :=
   DecimalNumber Exponent
   | DottedDecimalNumber [Exponent]
DottedDecimalNumber :=
   DecimalNumber . Digit*
   | . Digit+
Exponent := ExponentIndicator [Sign] Digit+
ExponentIndicator := e | E
HexFloatingPointLiteral := HexIndicator HexFloatingPointNumber
HexFloatingPointNumber :=
   HexNumber HexExponent
   | DottedHexNumber [HexExponent]
DottedHexNumber :=
   HexDigit+ . HexDigit*
   | . HexDigit+
HexExponent := HexExponentIndicator [Sign] Digit+
HexExponentIndicator := p | P
Sign := + | -
```

There are two basic types for floating point numbers:

- the float type corresponds to IEEE 754-2008 64-bit binary (radix 2) floating point numbers
- the decimal type corresponds to IEEE 754-2008 128-bit decimal (radix 10) floating point numbers

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.

IEEE-defined operations on floating point values must be performed using a rounding-direction attribute of roundTiesToEven (which is the default IEEE rounding direction, sometimes called *round to nearest*). All floating point values, including the intermediate results of expressions, must use the value space defined for the float and decimal type; implementations must not use extended precision for intermediate results. This ensures that all implementations will produce identical results. (This is the same as what is required by strictfp in Java.)

The implicit initial value for each floating point 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.

A string is iterable as a sequence of its single code point substrings. String is the only simple type that is iterable.

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

Structured values

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

- list
- mapping
- table
- xml
- error

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

```
structured-type-descriptor :=
   list-type-descriptor
   | mapping-type-descriptor
   | table-type-descriptor
   | xml-type-descriptor
   | error-type-descriptor
```

Lists

A list value is a mutable ordered list of values, which are called the members of the list. The number of members of the list is called the *length* of the list. A member of a list can be referenced by an integer index representing its position in the list. For a list of length *n*, the indices of the members of the list, from first to last, are 0,1,...,*n* - 1.

A list is iterable as a sequence of its members.

The type of list values can be described by two kinds of type descriptors.

```
list-type-descriptor :=
   array-type-descriptor | tuple-type-descriptor
```

Array types

An array type-descriptor describes a type of list value by specifying the type that the value for all members must belong to, and optionally, a length.

```
array-type-descriptor := member-type-descriptor [ [ array-length ] ]
member-type-descriptor := type-descriptor
array-length := int-literal | implied-array-length
implied-array-length := ! ...
```

The member type of an array type can be any type 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.)

If `array-length` is present, then it means that the type-descriptor matches only lists with the length specified in `array-length`. An array length of `!...` means that the length of the array is to be implied from the context; this is allowed in the same contexts where `var` would be allowed in place of the type descriptor in which `array-length` occurs (see “Typed binding patterns”). If `array-length` is not present, then the type-descriptor can match lists of any length.

When `array-length` is present, the implicit initial value of the array type is a list of the specified length, with each member of the array having its implicit initial value; otherwise, the implicit initial value of an array type is an array of length 0.

Note that `T[N]` is a subtype of `T[]`.

An array `T[]` is iterable as a sequence of values of type `T`.

Tuple types

A tuple type descriptor describes a type of list value by specifying a separate type for each member of the list.

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

A list value belongs to a type described by a tuple type descriptor if its length is the same as the number of member-type-descriptors in the tuple-type-descriptor, and each member of the list belongs to the type described by the corresponding member-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 list of those implicit initial values; otherwise, the tuple type does not have an implicit initial value.

Note that a tuple type where all the `member-type-descriptors` are the same is equivalent to a an array-type-descriptor with a length.

Mutability

The type of a list value says what can be read from the list value. This means that both kinds of list type are covariant in their member types.

Covariance for array types means that 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 any value you read from an array is of type int, it is also true as a matter of simple propositional logic that any value you read from the array is either of type int or of type type nil. Covariance for tuple types means that if type T1’ is a superset of type T1, and type T2’ is a superset of type T2, then type (T1’, T2’) is a superset of type (T1, T2).

Covariance requires that there be a mechanism to control what can be stored in an array. Otherwise, the type system could be subverted by mutation. For example, suppose

1. variable v1 contains a list with type T1[]
2. type T2 is a supertype of T1
3. variable v2 is of type T2[]
4. v1 is assigned to v2, which is allowed because T2 is a supertype of T1 and array types are covariant
5. a member of the list is changed via v2 to contain a value x of type T3 that belongs to T2 but not T1
6. the value x is now read via v1, but x is not of type T1

To deal with this problem, what can be stored in a list is constrained by a type descriptor that is part of the runtime value of a list; this type descriptor is called the storage type of the list. When a list value v has a storage type of T, any attempt to mutate v that would result in v no longer having type T will result in a runtime exception. So in the above example, step 5 will result in a runtime exception.

This implies that when a list has a storage type that is either an array type with a length or a tuple, then the list is of fixed length i.e. mutation cannot change the list’s length.

A list that is not fixed length must have a storage type T[] i.e. an array type without a length. In this case attempting to write a member of a list at an index *i* that is greater than or equal to the current length of the list will first increase the length of the list to *i* + 1, with the newly added members of the array having the implicit initial value of T.

Mappings

A mapping value is a mutable mapping from keys, which are strings, to values. We use the term *field* to mean a key together with the value to which it is mapped: the name of the field is the key, and the value of the field is the value to which the key is mapped.

The type of mapping values can be described by two kinds of type descriptors.

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

Mutability for mapping values is managed in a similar way to lists. The type of a mapping values says what can be read from the mapping. The names and values of fields that can be stored in a mapping value are constrained by a storage type that is part of the mapping value.

A mapping is iterable as sequence of fields, where each field is represented by a 2-tuple `(s, val)` where `s` is a string for the name of a field, and `val` is its value. The order of the fields in the sequence is implementation-dependent, but implementations are encouraged to preserve and use the order in which the fields were added.

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.)

A `map<T>` is iterable as a sequence of values of type `(string, T)`.

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

Record types

A record type descriptor describes a type of mapping value by specifying separately for each individual key the type of the value to which the key is mapped.

```
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 := [ record-rest-type ... ; ]
```

```
record-rest-type := type-descriptor | !
```

Each `individual-field-descriptor` specifies an additional constraint that a mapping value must satisfy for it to belong to the described type. The constraint depends on whether ? is present:

- if ? is not present, then the constraint is that the mapping value must have a field with the specified field-name and with a value belonging to the specified type-descriptor; this is called a required field;
- if ? is present, then the constraint is that if the mapping value has a field with the specified field-name, then its value must belong to the specified type-descriptor; this is called an optional field.

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.

The `record-rest-descriptor` determines whether a value of the described type may contain extra fields, that is fields other than those named by individual type descriptors, and, if so, the type of the values of the extra fields

- if the `record-rest-descriptor` is empty, then the value may contain extra fields with any value
- if the `record-rest-type` is !, then the value may not contain extra fields
- if the `record-rest-type` is a type descriptor, then the value may contain extra fields, and the type descriptor specifies the type of the values of the extra fields

A record type including !... is called *closed*; a record type that is not closed is called *open*.

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. Note that it does not pull in the `record-rest-descriptor` from the referenced 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 1 <= `i` <= `n`. In determining whether one record type is a subtype of another, each field in one record type is compared to the field with the same name in the other record type, regardless of the order in which the fields were specified in the record type descriptors, since field order is not significant for record types.

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 allow only non-nil simple values.

Like lists and mappings, a table value contains a storage type as part of its runtime value. The storage type for a type descriptor is a table type descriptor. Any attempt to mutate a table that would cause the table value no longer to have the type specified by the storage type will result in a runtime exception.

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.

A table is iterable as a sequence of mapping values, one for each row, where each mapping value has a field for each column, with the column as the field name and the value of that column in that row as the field value. The type and storage type of the mapping values will be a closed record type.

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 an immutable sequence of zero or more of the items that can occur inside an XML element, specifically:

- elements
- characters
- processing instructions
- comments

The attributes of an element are represented by a map<string>. The content of each element in the sequence is itself a distinct XML value. Although the sequence is immutable, an element can be mutated to change its content to be another XML value.

An XML value is iterable as a sequence of its items, where each character item is represented by a string with a single code point and other items are represented by a singleton XML value. A single XML item, such as an element, is represented by a sequence consisting of just that item.

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

Error

```
error-type-descriptor :=
 error [<reason-type-descriptor[, detail-type-descriptor]>]
reason-type-descriptor := type-descriptor
detail-type-descriptor := type-descriptor
```

An error value is a distinct basic type, used only for representing errors. It contains the following information:

- a reason, which is a string identifier for the category of error
- a detail, which is a mapping providing additional information about the error
- a stack trace

An error-type-descriptor describes an error value whose reason-string belongs to reason-type-descriptor and whose detail mapping belongs to detail-type-descriptor.

It is a compile error if reason-type-descriptor is not a subtype of string or if detail-type-descriptor is not a subtype of `map<any>`.

If the reason-type-descriptor is omitted, it defaults to `string`; if the detail-type-descriptor is omitted, it defaults to `map<any>`.

Error is the only structured basic type that is not iterable.

An error type does not have an implicit initial value.

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 function-signature
function-signature := ( param-list ) return-type-descriptor
return-type-descriptor := [ returns annots type-descriptor ]
param-list :=
   [ individual-param-list [, rest-param]]
   | rest-param
individual-param-list := individual-param (, individual-param)*
individual-param :=
   annots 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
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 argument 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 functions based only on the arguments they are declared to accept and the values they are declared to return. Other aspects of a function value, such as how the return value depends on the argument, or what the side-effects of calling the function are, are not considered as part of the function's type.

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 fields along with a set of associated functions, called methods, that can be used to manipulate them. An object's methods are associated with the object when the object is constructed and cannot be changed thereafter.

An object type performs two roles:

- it describes the type of each field and each method
- it defines a way of constructing an object of this type, in particular it provides the definitions that are associated with the object when it is constructed.

It is also possible to have an object type which only performs the first of these two roles; this is called an abstract object type.

```
object-type-descriptor :=
   [abstract] object {
      object-member-descriptor*
      [object-ctor-function object-member-descriptor*]
    }
object-member-descriptor :=
   object-field-descriptor
   | object-method
   | object-type-reference
```

If the keyword `abstract` is present, then the object type is an abstract object type. An abstract object type must not have an object-ctor-function and does not have an implicit initial value.

Fields

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

An `object-field-descriptor` specifies a field of the object. The names of all the fields of an object must be distinct.

Methods

Methods are functions that are associated to the object and are called via a value of that type using a `method-call-expr`.

```
object-method := method-decl | method-defn
method-decl :=
   metadata
   [visibility-qual]
   function method-name function-signature ;
method-defn :=
   metadata
   method-defn-quals
   function method-name function-signature method-body
method-defn-quals :=
  [extern] [visibility-qual] | [visibility-qual] extern
method-name := identifier
```

```
method-body := function-body-block | ;
```

The names of all the methods of an object must be distinct: there is no method overloading. But the fields and methods of an object are in separate namespaces, so it is possible for an object to have a field and a method with the same name. Method names beginning with two underscores are reserved for use by built-in object types (see section XXX).

Within the function-body-block of a method, the fields and methods of the object are not implicitly in-scope; instead the keyword `self` is bound to the object and can be used to access fields and methods of the object.

All the methods of an object type must be either declared or defined in the object. A `method-decl` declares a method without defining it; a `method-defn` also supplies a definition. In a `method-defn`, the `method-body` must be a bare semicolon if and only if `method-defn-quals` includes `extern`. The presence of `extern` means that the implementation of the method is not provided in the Ballerina source module. An abstract object type descriptor must not contain a `method-defn`.

Any method that is declared but not defined within an object that is not abstract must be defined outside the object at the top-level of the module:

```
outside-method-defn :=
   function qual-method-name function-signature function-body-block
qual-method-name := object-type-name . method-name
```

Constructor function

An object type descriptor that is not abstract may have a constructor function that will be invoked to create objects of this type; if it includes a field of a type that does not have an implicit initial value, then it must include a constructor function.

```
object-ctor-function :=
   metadata
   function-defn-quals new ( ctor-param-list ) method-body
 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.

In an `object-ctor-function`, a `method-body` must be a bare semicolon if and only if `function-defn-quals` includes `extern`.

Implicit initial value

If an object type includes a constructor function and the constructor function has no required parameters, then the implicit initial value for the object type is the result of calling the constructor function with no arguments. Otherwise, if the object type is not abstract and the type of every field of the object type has an implicit initial value, then the implicit initial value is an object with each field initialized to its implicit initial value. Otherwise, the object type does not have an implicit initial value.

Object type references

```
object-type-reference :=
   * type-descriptor-reference ;
```

The `type-descriptor-reference` in an `object-type-reference` must reference a an abstract object type. The object-member-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 a non-abstract object type OT has a type reference to an abstract object type AT, then each method declared in AT must be defined in OT using either a `method-defn` or an `outside-method-defn`.

Visibility

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

Visibility of fields, object initializer and methods 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`. The visibility of a method or field of an abstract object type cannot be private.

Visibility of methods is determined by the declaration in the object type descriptor; external method 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-descriptor
   | singleton-type-descriptor
   | union-type-descriptor
   | optional-type-descriptor
   | any-type-descriptor
   | byte-type-descriptor
   | json-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
```

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 or decimal is 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`.

The rules for the implicit initial value of a union type are as follows:

- if () is in the value space of a union type, then the implicit initial value for the union type is ();
- otherwise if the value space only contains only values of a single basic type, and one of those values is the implicit initial value of that basic type, then the implicit initial value for the union is the implicit initial value of the basic type of its values (for example, the implicit initial value for 0|1 is 0)
- otherwise 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.

Byte type

```
byte-type-descriptor := byte
```

The byte type is an abbreviation for a union of the int values in the range 0 to 255 inclusive.

Following the rules for the implicit initial value of union types implies that the implicit initial value of byte is 0.

JSON types

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

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

Built-in object types

There are several abstract object types that are built-in in the sense that the language treats objects with these types specially. Note that it is only the types that are built-in; the names of these types are not built-in.

Note It is likely that a future version of this specification will provide generic types, so that a library can provide definitions of these built-in types.

Iterator

A value that is iterable as a sequence of values of type T provides a way of creating an iterator object that matches the type

```
abstract object {
   public next() returns record { T value; !...; }?;
}
```

In this specification, we refer to this type as Iterator<T>.

Conceptually an iterator represents a position between members of the sequence. Possible positions are at the beginning (immediately before the first member if any), between members and at the end (immediately after the last member if any). A newly created iterator is at the beginning position. For an empty sequence, there is only one possible position which is both at the beginning and at the end.

The `next()` method behaves as follows:

- if the iterator is currently at the end position, return nil
- otherwise
    - move the iterator to next position, and
    - return a record `{ value: v }` where v is the member of the sequence between the previous position and the new position

Note that it is not possible for the `next()` method simply to return a member of the sequence, since a nil member would be indistinguishable from the return value for the end position.

Iterable

An object can make itself be iterable as a sequence of values of type T by providing a method named `__iterator` which returns a value that is a subtype of Iterator<T>. In this specification, we refer to this type as Iterable<T>.

Collection

An object can be declare itself to be a collection of values of type V indexed by keys of type K, but defining a `__get(K k)` method returning a value of type V, that returns the value associated with key k. If the collection is mutable, then the object can also declare a `__put(K k, V v)` method that changes the value associated with key k to to value v. In this specification, we refer to these types as ImmutableCollection<T> and MutableCollection<T>.

Binding patterns and variables

Binding patterns

Binding patterns are used to support destructuring, which allows different parts of a single structured value each to be assigned to separate variables at the same time.

```
binding-pattern :=
   simple-binding-pattern
   | structured-binding-pattern
simple-binding-pattern := variable-name | ignore
variable-name := identifier
ignore := _
structured-binding-pattern :=
   | tuple-binding-pattern
   | record-binding-pattern
   | error-binding-pattern
tuple-binding-pattern := ( binding-pattern (, binding-pattern)+ )
record-binding-pattern := { entry-binding-patterns }
entry-binding-patterns :=
   field-binding-patterns [, rest-binding-pattern]
   | [ rest-binding-pattern ]
field-binding-patterns :=
  field-binding-pattern (, field-binding-pattern)*
field-binding-pattern :=
   field-name : binding-pattern
   | variable-name
rest-binding-pattern := ... variable-name | ! ...
```

```
error-binding-pattern :=
   error ( simple-binding-pattern [, error-detail-binding-pattern] )
error-detail-binding-pattern :=
   simple-binding-pattern
   | mapping-binding-pattern
```

A binding pattern may succeed or fail in matching a value. A successful match causes values to be assigned to all the variables occurring the binding-pattern.

A binding pattern matches a value in any of the following cases.

- a simple-binding-pattern matches any value; if the simple-binding-pattern consists of a variable-name then it causes the matched value to be assigned to the named variable;
- a tuple-binding-pattern (p1, p2, …, pn) matches a list value of length n [v1, v2, …, vn] if pi matches vi for each i in 1 to n;
- a record-binding-pattern { f1: p1, f2: p2, …, fn: pn, r } matches a mapping value m that has fields f1, f2, ... , fn if pi matches the value of field fi for each i in 1 to n, and if the rest-binding-pattern r, if present, matches a new mapping x consisting of all the fields other than f1, f2,...,fn
    - if r is `!...`, it matches x if x is empty
    - if r is `...v` it matches any mapping x and causes x to be assigned to v
    - if r is missing, the match of the record-binding-pattern is not affected by x
    - a field-binding-pattern consisting of just a variable name `x` is equivalent to a field-binding-pattern `x: x`
- an error-binding-pattern error(rp, dp) matches an error value if rp matches the error reason and dp matches the error detail record;
- an error-binding-pattern error(rp) is equivalent to error(rp, _)

All the variables in a binding-pattern must be distinct e.g. (x, x) is not allowed.

Given a type descriptor for every variable in a binding-pattern, there is a type descriptor for the binding-pattern that will contain a value just in case that the binding pattern successfully matches the value causing each variable to be assigned a value belonging to the type descriptor for that variable.

- for a simple-binding-pattern that is a variable name, the type descriptor is the type descriptor for that variable;
- for a simple-binding-pattern that is _, the type descriptor is any
- for a tuple-binding-pattern, the type descriptor is a tuple type descriptor;
- for a record-binding-pattern, the type descriptor is a record type descriptor;
- for an error-binding-pattern, the type descriptor is an error type descriptor.

Typed binding patterns

```
typed-binding-pattern := impliable-type-descriptor binding-pattern
impliable-type-descriptor := type-descriptor | var
```

A typed-binding-pattern combines a type-descriptor and a binding-pattern, and is used to create the variables occurring in the binding-pattern. If var is used instead of a type-descriptor, it means the type is implied. How the type is implied depends on the context of the typed-binding-pattern.

The simplest and most common form of a typed-binding-pattern is for the binding pattern to consist of just a variable name. In this cases, the variable is constrained to contain only values matching the type descriptor.

When the binding pattern is more complicated, the binding pattern must be consistent with the type-descriptor, so that the type-descriptor unambiguously determines a type for each variable occurring in the binding pattern.

For a typed-binding-pattern to match a value, the binding-pattern must match the value and the value must belong to the type.

Each syntactic construct that uses a typed-binding-pattern does so in one of two ways:

- conditionally, meaning at runtime it may succeed or fail in matching a value;
- unconditionally, meaning the the compiler will give an error if the static types do not guarantee that it will successfully match any value that that could possibly be matched with it in the context in which it occurs.

Variable scoping

For every variable, there is place in the program that creates it. Variables are lexically scoped: every variable has a scope which determines the region of the program within which the variable can be referenced.

There are two kinds of scope: module scope and block scope. A variable with module scope can be referenced anywhere within a module. Identifiers with module scope are used to identify not only variables but other module-level entities such as functions.

A variable with block scope can be referenced only within a particular block (always delimited with curly braces). Block-scope variables are created by a variety of different constructs, many of which use a typed-binding-pattern. Parameters are treated as read-only variables with block scope.

A variable with block scope can have the same name as a variable with module scope; the former variable will hide the latter variable while the former variable is in scope. However, it is a compile error if a variable with block scope has the same name as another variable with block scope and the two scopes overlap.

6. Expressions

```
expression :=
   literal
   | array-constructor-expr
   | tuple-constructor-expr
   | mapping-constructor-expr
   | table-constructor-expr
   | error-constructor-expr
   | string-template-expr
   | xml-expr
   | new-expr
   | variable-reference-expr
   | field-access-expr
   | index-expr
   | xml-attributes-expr
   | function-call-expr
   | method-call-expr
   | anonymous-function-expr
   | arrow-function-expr
   | explicit-type-expr
   | unary-expr
   | multiplicative-expr
   | additive-expr
   | shift-expr
   | range-expr
   | numeric-comparison-expr
   | equality-expr
   | binary-bitwise-expr
   | logical-expr
   | conditional-expr
   | but-expr
   | await-expr
   | table-query-expr
   | ( expression )
```

For simplicity, the expression grammar is ambiguous. The following table shows the various types of expression in increasing order of precedence, together with associativity.

Operator	**Associativity**

`x.k`
`f(x)`
`x.f(y)`
`x[y]`
`new T(x)`	

`+x`
`-x`
`~x`
`!x`	

`untaint x`
`<T> x`
`await x`	

`x * y`
`x / y`
`x % y`
	left

`x + y`
`x - y`
	left

`x << y`
`x >> y`
`>>>`
	left

`x ... y`
`x ..< y`
	non

`x < y`
`x > y`
`x <= y`
`x >= y`
	non

`x == y`
`x != y`
	left

`x & y`	left
`x ^ y`	left
`x | y`	left
`x && y`	left
`x || y`	left

`x ?: y`
`but`
	right

`x ? y : z`	right
`(x) => y`	right

Static typing of expressions

A type is computed for every expression at compile type; this is called the static type of the expression. The compiler and runtime together guarantee that when an expression is evaluated at runtime the value will belong to the static type.

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.

Contextually expected type

In Ballerina's type system, values do not belong to just a single type. It is accordingly convenient to allow an expression that constructs a value to have a different static type in different contexts. The context of every expression determines a type that the static type of the expression must be a subtype of. This is called the contextually expected type. For example, the contextually expected type of an expression that initializes a variable of type T is T. In some cases, there will be no constraint on the type of an expression; this is equivalent to having a contextually expected type of any.

The contextually expected type of an expression that constructs a value can affect what value the expression creates when it is evaluated.

Precise and broad types

There is an additional complexity relating to inferring types. Expressions in fact 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 floating-point-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 or decimal types, so the precise and broad types for a floating-point-literal are both the basic type `float`.) For example, the precise type of the string literal `"X"` is the singleton type `"X"`, but the broad type is `string`.

For an explicit-type-expression, the precise type and the broad type are the type specified in the cast.

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 always requires programmers to make conversions explicit, even between different types of number; there are no implicit conversions.

Literals

```
literal :=
   nil-literal
   | boolean-literal
   | int-literal
   | floating-point-literal
   | string-literal
   | byte-array-literal
```

The type of an int-literal depends on the contextually expected numeric type, where the contextually expected numeric type is the intersection of the contextually expected type with `int|float|decimal`:

- if the contextually expected numeric type is a subtype of float, then the type of the int-literal is a float
- if the contextually expected numeric type is a subtype of decimal, then the type of the int-literal is decimal
- otherwise, the broad type of the int-literal is int, and the narrow type is a singleton int type

Similarly the type of a floating-point-literal is determined as follows:

- if the contextually expected numeric type is a subtype of decimal, the type of the floating-point-literal is a decimal,
- otherwise, the type of the floating-point literal is float

```
byte-array-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*
```

The static type of byte-array-literal is `byte[N]`, where N is the number of bytes encoded by the Base16Literal or Base64Literal. The storage type of the array value created is also `byte[N]`.

Array constructor

```
array-constructor-expr := [ [ expr-list ] ]
expr-list = expression (, expression)*
```

Creates a new list value. The members of the list come from evaluating each expression in the expr-list in order. The storage type of the newly created list will be an array type. If the contextually expected type is an array type, then that type will be used as the storage type, and the contextually expected type for each expression in the expr-list is the member type of the array type. If the contextually expected type is an array type with a length, then it is a compile error if the length of the array type is not equal to the number of expressions in the expr-list. If the contextually expected type is not an array type, then the member type of the storage type is the union of the broad types of the expressions for the members. In both cases, the static type of the array-constructor-expr will be the same as the storage type.

Tuple constructor

```
tuple-constructor-expr := ( expr-list2 )
expr-list2 := expression (, expression)+
```

Creates a new list value. The members of the list come from evaluating each expression in `expr-list2` in order. The storage type of the newly created list will be a tuple type. If the contextually expected type is a tuple type, then that type is used as the storage type, and the contextually expected type for each expression in `expr-list2` is the corresponding member type of the tuple type; it is a compile error if the number of members of the contextually expected tuple type is not equal to the number of expressions in `expr-list2`. If the contextually expected type is not a tuple, then the storage type will be a tuple where each member type is the broad type of the corresponding expressions in `expr-list2`. In both cases, the static type of the `tuple-constructor-expr` will be the same as the storage type.

Mapping constructor

```
mapping-constructor-expr := { [field (, field)*] }
field := (literal-field-name | computed-field-name) : value-expr
literal-field-name := field-name | string-literal
computed-field-name := [ expression ]
value-expr := expression
```

A mapping-constructor-expr creates a new mapping value. An expression can be used to specify the name of a field by enclosing the expression in square brackets.

The contextually expected type of mapping-constructor-expr determines the storage type of the constructed value. If the contextually expected type is neither a record type nor a map type, 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)* }
```

The contextually expected type of the table-constructor-expr determines the storage type of the constructed value.

For example,

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

Error constructor expression

```
error-constructor-expr := error ( reason-expr [, detail-expr] )
reason-expr := expression
detail-expr := expression
```

An error-constructor-expr constructs a new error value with the specified reason string and detail mapping. If detail-expr is omitted, then the detail mapping for the error is a new empty mapping. The stack trace describes the stack at the point where the error constructor was evaluated.

String template expression

```
string-template-expr := string BacktickString
BacktickString :=
  ` BacktickItem* [BacktickFinalOneChar] `
BacktickItem :=
   BacktickOneChar
   | BacktickTwoChars
   | [BacktickFinalOneChar] interpolation
interpolation := ${ expr }
BacktickOneChar := ^ ( ` | $ )
BacktickTwoChars :=  $ ^ ( { | ` | $)
BacktickFinalOneChar := $
```

A `string-template-expr` interpolates the results of evaluating expressions into a literal string. The static type of the expression in each interpolation must be a simple type and must not be nil. Within a `BacktickString`, every character that is not part of an `interpolation` is interpreted as a literal character. A string-template-expr is evaluated by evaluating the expression in each interpolation in the order in which they occur, and converting the result of the each evaluation to a string as if using by `<string>`. The result of evaluating the `string-template-expr` is a string comprising the literal characters and the results of evaluating and converting the interpolations, in the order in which they occur in the `BacktickString`.

A literal ` can be included in string template by using an interpolation `${"`"}`.

XML expression

```
xml-expr := xml BacktickString
```

An XML expression creates an XML value as follows:

1. The backtick string is parsed to produce a string of literal characters with interpolated expressions
2. The result of the previous step is parsed as XML content. More precisely, it is parsed using the production `content` in the W3C XML Recommendation. For the purposes of parsing as XML, each interpolated expression is interpreted as if it were an additional character allowed by the CharData and AttValue productions but no other. The result of this step is an XML Infoset consisting of a ordered list of information items such as could occur as the [children] property of an element information item, except that interpolated expressions may occur as Character Information Item or in the [normalized value] of an Attribute Information Item. Interpolated expressions are not allowed in the value of a namespace attribute.
3. This infoset is then converted to an XML value, together with an ordered list of interpolated expressions, and for each interpolated expression a position within the XML value at which the value of the expression is to be inserted.

4. The static type of an expression occurring in an attribute value must be a simple type and must not be nil. The static type type of an expression occurring in content can either be xml or a non-nil simple type.
5. When the xml-expr is evaluated, the interpolated expressions are evaluated in the order in which they occur in the `BacktickString`, and converted to strings if necessary. A new copy is made of the XML value and the result of the expression evaluations are inserted into the corresponding position in the newly created XML value. This XML value is the result of the evaluation.

New expression

```
new-expr := new-value-expr | new-object-expr
new-value-expr : new [type-descriptor]
new-object-expr := new [type-descriptor] ( arg-list )
```

A new-value-expr allocates storage for a reference type and initializes it with the implicit initial value for that type.

A new-object-expr allocates storage for an object type and initializes it by calling the object type's constructor.

If the type descriptor is omitted in either kind of new-expr, then the contextually expected type is used instead.

It is a compile error if the type-descriptor for a new-value-expr, whether explicitly specified or not, is not a reference type with an implicit initial value. It is a compile error if the type-descriptor for a new-object-expr, whether explicitly specified or not, is not an object type or if the arg-list does not match the signature of the object type's constructor.

Variable reference expression

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

Field access expression

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

Index expression

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

Attempting to read a member of a list using an out of bounds index will result in a runtime exception.

XML attributes expression

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

Returns the attributes map of an singleton xml value, or nil if the operand is not a singleton xml value. The result type is `map<string>?`.

Function call expression

```
function-call-expr := function-reference ( arg-list )
function-reference := variable-reference
arg-list :=
   [ individual-arg-list [, rest-arg]]
   | rest-arg
individual-arg-list := individual-arg (, individual-arg)*
individual-arg := [arg-name :] expression
arg-name := identifier
rest-arg := ... expression
```

When a function to be called results from the evaluation of an expression that is not merely a variable reference, the function can be called either by first storing the value of the expression in a variable or using the `call` built-in method on functions.

Method call expression

```
method-call-expr := expression . method-name ( arg-list )
```

The basic type of the value that results from evaluating expression determines how the method-name is used to lookup the method to call. If the basic type is object, then the method is looked up in the object's methods; otherwise, the method is looked up in the built-in methods of that basic type.

Built-in methods

The following built-in methods are available.

Method name	**Basic types**	**Return value**	**Semantics**
length	list	int	length of this array
	string		length (number of code points) of this string
	table		number of rows of this table

	mapping		number of entries in this mapping
	XML		number of content items in this XML value; each character, element, comment and processing instructions is counted as a single item
iterator	list	Iterator<T>	new iterator over the list; T is the type of the members of the list
	mapping	Iterator <(string,T)>	new iterator over the mapping; T is the type of the field values
	table	Iterator<R>	new iterator over table; R is record representing a single row
	string	Iterator<string>	new iterator over the string
	XML	Iterator <string|XML>	new iterator over the XML value
call	function	same as returned by function	calls this function; arguments to call become arguments to function; return value of called function is return value of call
isFinite	float, decimal	boolean	true iff this is neither infinity nor NaN
isInfinite	float, decimal	boolean	true iff this is infinity (plus or minus)
isNaN	float, decimal	boolean	true iff this is NaN
reason	error	same as reason type parameter	returns the reason string for this error
detail	error	same as detail type parameter	returns the detail mapping for this error
stackTrace	error	object	returns the stack trace for this error

In the above description, “this” refers to the object on which the built-in method is called.

Anonymous function expression

```
anonymous-function-expr :=
  function function-signature function-body-block
```

Evaluating an anonymous-function-expr creates a closure, whose basic type is function. If function-body-block refers to a block-scope variable defined outside of the function-body-block, the closure will capture a reference to that variable; the captured reference will refer to the same storage as the original reference not a copy.

Arrow function expression

```
arrow-function-expr := arrow-param-list => expression
arrow-param-list :=
   identifier
   | ([identifier (, identifier)*])
```

Arrow functions provide a convenient alternative to anonymous function expressions that can be used for many simple cases. An arrow function can only be used in a context where a function type is expected. The types of the parameters are inferred from the expected function type. The scope of the parameters is `expression`. The static type of the arrow function expression will be a function type whose return type is the static type of `expression`. If the contextually expected type for the `arrow-function-expr` is a function type with return type T, then the contextually expected type for `expression` is T.

Explicitly typed expression

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

An explicit-type-expr specifies a static type for an expression explicitly.

The static type of an `explicit-type-expr` is the type specified in its `type-descriptor`. The type specified in type-descriptor is also the contextually expected type used to interpret `expression`.

Normally, it is a compile time error if expression is not of the type explicitly specified in the type descriptor. However, if the static type of `expression` is a a subtype of a non-nil basic type and type-descriptor is a non-nil simple basic type, then a conversion will be performed at runtime according to the following table. Note that these conversions never fail at runtime.

from \ to	<string>	<float>	<decimal>	<int>	<boolean>
string	unchanged	not allowed			
float	decimal numeral	unchanged	closest math value	round, exception for NaN or out of int range	true iff non-zero
decimal		closest math value	unchanged		
int	decimal numeral	same math value		unchanged	

boolean	"true" or "false"	1.0 or 0.0	1.0 or 0.0	1 or 0	unchanged

Unary expression

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

The unary - operator performs negation. The static type of the operand must be a number; the static type of the result is the same basic type as the static type of the operand. The semantics for each basic type are as follows:

- int: negation for int is the same as subtraction from zero; an exception is thrown on overflow, which happens when the operand is -2^{63}
- float, decimal: negation for floating point types corresponds to the negate operation as defined by IEEE 754-2008 (this is not the same as subtraction from zero); no exceptions are thrown

The unary + operator returns the value of its operand expression. The static type of the operand must be a number, and the static type of the result is the same as the static type of the operand expression.

The ~ operator inverts the bits of its operand expression. The static type of the operand must be int, and the static type of the result is an int.

The `!` operator performs logical negation. The static type of the operand expression must be boolean. The `!` operator returns true if its operand is false and false if its operand is true.

Multiplicative expression

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

The `*` operator performs multiplication; the `/` operator performs division; the `%` operator performs remainder.

The static type of both operand expressions is required to be the same basic type; this basic type will be the static type of the result. The following basic types are allowed:

- int
  - * performs integer multiplication; an exception will be thrown on overflow
  - / performs integer division, with any fractional part discarded ie with truncation towards zero; an exception will be thrown on division by zero or on overflow (which happens if the first operand is -2^{63} and the second operand is -1)
  - % performs integer remainder consistent with integer division, so that if x/y does not throw an exception, then (x/y)*y + (x%y) is equal to x; an exception will be thrown if the second operand is zero; if the first operand is -2^{63} and the second operand is -1, then the result is 0
- float, decimal
  - `*` performs the multiplication operation with the destination format being the same as the source format, as defined by IEEE 754-2008; no exceptions are thrown
  - `/` performs the division operation with the destination format being the same as the source format, as defined by IEEE 754-2008; no exceptions are thrown
  - `%` performs a remainder operation; the remainder is not the IEEE-defined remainder operation but is instead a remainder operation analogous to integer remainder; more precisely,
    - if x is NaN or y is NaN or x is an infinity or y is a zero, then x % y is NaN
    - for finite x, and infinite y, x % y is x
    - for finite x and finite non-zero y, x % y is equal to the result of rounding x - (y × n) to the nearest representable value using the roundTiesToEven rounding mode, where n is the integer that is nearest to the mathematical quotient of x and y without exceeding it in magnitude; if the result is zero, then its sign is that of x
    - no exceptions are thrown

Additive expression

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

The `+` operator is used for both addition and concatenation; the `-` operator is used for subtraction.

It is required that either

- the static type of both operand expressions is the same basic type, in which case this basic type will be the static type of the result, or
- the static type of one operand expression must be xml and of the other operand expression must be string, in which case the static type of the result is xml

The semantics for each basic type is as follows:

- int

    - + performs integer addition; an exception will be thrown on overflow
    - - peforns integer subtraction; an exception will be thrown on overflow
- float, decimal
    - + performs the addition operation with the destination format being the same as the source format, as defined by IEEE 754-2008; no exceptions are thrown
    - - performs the subtraction operation with the destination format being the same as the source format, as defined by IEEE 754-2008; no exceptions are thrown
- string, xml
    - if both operands are a string, then the result is a string that is the concatenation of the operands
    - if both operands are xml, then the result is a new xml sequence that is the concatenation of the operands; the new xml sequence contains a shallow copy of both operands; this operation does not perform a copy of the content or attributes of any elements in sequence
    - if one operand is a string and one is xml, the string is treated as if it were an xml sequence with one character item for each code point in the string

Shift expression

```
shift-expr :=
   expression << expression
   expression >> expression
   expression >>> expression
```

A shift-expr performs a bitwise shift. Both operand expressions must have static type that is an int, and the static type of the result is int. The left hand operand is the value to be shifted; the right hand value is the shift amount; all except the bottom 6 bits of the shift amount are masked out (as if by x & 0x3F). Then a bitwise shift is performed depending on the operator:

- << performs a left shift, the bits shifted in on the right are zero
- >> performs a signed right shift; the bits shifted in on the left are the same as the most significant bit
- >>> performs a unsigned right shift, the bits shifted in on the left are zero

Range expression

```
range-expr :=
   expression ... expression
   | expression ..< expression
```

A range-expr results in a new array of integers in increasing order including all integers n such that

- the value of the first expression is less than or equal to n, and
- n is
    - if the operator is `...`, less than or equal to the value of the second expression

    - if the operator is `..<`, less than the value of the second expression

It is a compile error if the static type of either expression is not a subtype of int.

A range-expr is designed to be used in a foreach statement, but it can be used anywhere.

Numerical comparison expression

```
numerical-comparison-expr :=
   expression < expression
   | expression > expression
   | expression <= expression
   | expression >= expression
```

A numerical-comparison-expr compares two numbers.

The static type of both operands must be of the same basic type, which must be int, float or decimal. The static type of the result is boolean.

Floating point comparisons follow IEEE, 754-2008, so
- if either operand is NaN, the result is false
- positive and negative zero compare equal

Equality expression

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

An equality-expr tests whether two values are equal.

With the == operator, an equality-expr evaluates to true if the results of evaluating the two expressions are equal and otherwise evaluates to false, where two values are equal if and only if they have the same basic type T and
- if T is a reference type, the values refer to the same storage
- if T is a floating point type, the values are numerically equal
- otherwise (a simple type that is not a floating point type), the values are identical values in T’s value space

Numerical equality for floating point types follows IEEE 754-2008:
- positive and negative zero are numerically equal
- NaN is not numerically equal to any floating point value (including NaN)

An equality-expr with the != operator evaluates to true or false according as the same expression with == operator would evaluate to false or true, except that, following IEEE 754-2008, it evaluates to false even if both operands are NaN.

It is a compile error if the intersection of the static types of the operands is empty.

Binary bitwise operators

```
binary-bitwise-expr :=
   bitwise-and-expr
   | bitwise-xor-expr
   | bitwise-or expr
bitwise-and-expr = expression & expression
bitwise-xor-expr = expression ^ expression
bitwise-or-expr = expression | expression
```

A binary-bitwise-expr does a bitwise AND, XOR, or OR operation on its operands.

The static type of both operands must be int or a subtype. The static type of the result is as follows:

- for AND, if one operand has type byte, then the result has type byte; otherwise, the result has type int;
- for OR, if both operands have type byte, then the result has type byte; otherwise, the result has type int;
- for XOR, the result has type int.

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-clause (, but-clause)* }
but-clause := type-test => expression
```

A but-expr selects between expressions based on the type that a value belongs to.

The scope of the variables created in the typed-binding-pattern of a type-test is the expression in that but-clause.

A but-expr is executed as follows:

1. the expression is evaluated resulting in some value v;
2. for each but-clause in order:
    a. the value v is tested using the type-test
    b. if the type-test succeeds, causing assignment to the variables in its binding-pattern, then the expression in the but-clause is evaluated resulting in a value that becomes the result of the but-expr
3. if no type-test succeeds, the result of the but-expr is v

When a type-test in a but-clause uses var, the type for the binding-pattern is inferred from the type of the expression preceding `but`, and the type-test of preceding but-clauses. The semantics of type-test is defined in the section for the match statement.

Await expression

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

The static type of `expression` must be a subtype of future. Await blocks the calling worker until the function represented by the future completes or throws an exception. If it throws an exception, then the result of the await-expr is the thrown error value. Thus, if the static type of `expression` is `future<T>`, then the static type of await-expr T | error.

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.

7. Statements

```
statement :=
   block-stmt
   | local-var-decl-stmt
   | xmlns-decl-stmt
   | assignment-stmt
   | compound-assignment-stmt
   | destructuring-assignment-stmt
   | checked-stmt
   | function-call-stmt
   | method-call-stmt
   | action-invocation-stmt
   | if-else-stmt
   | match-stmt
   | foreach-stmt
   | while-stmt
   | break-stmt
   | continue-stmt
   | fork-stmt
   | try-stmt
   | throw-stmt
   | transaction-stmt
   | lock-stmt
   | worker-interaction-stmt
   | forever-stmt
   | return-stmt
   | done-stmt
```

Block statement

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

A `block-stmt` executes its statements sequentially.

Local variable declaration statements

```
local-var-decl-stmt :=
   local-init-var-decl-stmt
   | local-no-init-var-decl-stmt
local-init-var-decl-stmt := annots var-decl-init ;
```

A `local-var-decl-stmt` is used to declare variables with a scope that is local to the block in which they occur.

The scope of variables declared in a `local-var-decl-stmt` starts immediately after the statement and continues to the end of the block statement in which it occurs.

```
var-decl-init := typed-binding-pattern = initializer
initializer := expression
```

A `var-decl-init` uses a `typed-binding-pattern` and an initializer expression to create and initialize one or more variables. A local-init-var-decl-stmt is executed by evaluating the initializer expression is resulting in a value, and then matching the typed-binding-pattern to the value, causing assignments to the variables occurring in the typed-binding-pattern. The typed-binding-pattern is used unconditionally, meaning that it is a compile error if the static types do not guarantee the success of the match. If the typed-binding-pattern uses `var`, then the type of the variable is inferred from the static type of initializer; in this case, the broad type is used.

```
local-no-init-var-decl-stmt :=
   annots type-descriptor variable-name ;
```

A local variable declared `local-no-init-var-decl-stmt` must be definitely assigned at each point that the variable is referenced. This means that the compiler must be able to verify that the local variable will have been assigned before that point.

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 CompoundAssignmentOperator expression ;
```

```
CompoundAssignmentOperator := BinaryOperator =
BinaryOperator := + | - | * | / | & | | | ^ | << | >> | >>>
```

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 :=
   structured-binding-pattern = expression ;
```

A destructuring assignment is executed by evaluating the expression resulting in a value v, and then matching the structured binding pattern to v, causing assignments to the variables occurring in the structured binding pattern.

The binding-pattern has a static type implied by the static type of the variables occurring in it. The static type of expression must be a subtype of this type.

Checked statements

```
checked-stmt :=
   checked-assignment-stmt
   | checked-compound-assignment-stmt
   | checked-function-call-stmt
   | checked-method-call-stmt
checked-assignment-stmt := lhs = check expression ;
checked-compound-assignment-stmt :=
   lhs CompoundAssignmentOperator check expression ;
checked-function-call-stmt := check function-call-expr ;
checked-method-call-stmt := check method-call-expr ;
```

The check construct allows an error to be handled by returning from a function.

A checked statement is evaluated in a similar way to its corresponding unchecked statement, except that after the expression following check has been evaluated resulting in some value v, if v is an error, then v is returned as the result of the containing function, as if by a return statement. It is accordingly a compile-time error if the function return type does not include the appropriate error type.

The static type of the expression following check must include the error type. The static type of check is treated as the type of the following expression with the error type removed.

Function call statement

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

Method call statement

```
method-call-stmt := method-call-expr ;
```

Action invocation statement

```
action-invocation-stmt :=
   [lhs =] variable-reference -> identifier ( arg-list ) ;
```

Conditional statement

```
if-else-stmt :=
   if expression block-stmt
   [ else if expression block-stmt ]*
   [ else block-stmt ]
```

The if-else statement is used for conditional execution.

The static type of the expression following `if` must be boolean. When an expression is true then the corresponding block statement is executed and the if statement completes. If no expression is true then, if the else block is present, the corresponding block statement is executed.

Match statement

```
match-stmt := match expression { match-clause+ }
match-clause := type-test => block-stmt
```

A match statement selects a block statement to execute based on the type that a value belongs to.

The scope of any variables created in the typed-binding-pattern of a type-test is the block-stmt in the corresponding match-clause.

A match-stmt is executed as follows:

1. the expression is evaluated resulting in some value v;
2. for each match-clause in order:
    a. the value v is tested using the type-test
    b. if the type-test succeeds, causing assignments to any the variables in its binding-pattern, then the block-stmt is executed and execution of the match-stmt terminates

It is a compile error if the compiler cannot verify that at least one of the type-test clauses will match the expression.

```
type-test := typed-binding-pattern | type-descriptor
```

A typed-binding-pattern is a type-test is used conditionally. A type-test that consists of just a type-descriptor `T` is equivalent to a typed binding pattern `T _`.

When a type-binding-pattern in a match-clause uses var, the implied type is inferred from the type of the match expression and the type-test of preceding match-clauses, by subtracting the type of the preceding match-clauses from the type of the match expression.

Foreach statement

```
foreach-stmt :=
   foreach typed-binding-pattern in expression block-stmt
```

A foreach statement iterates over a sequence, executing a block statement once for each member of the sequence.

The scope of any variables created in typed-binding-pattern is block-stmt.

In more detail, a foreach statement executes as follows:

1. evaluate the expression resulting in a value c
2. create an iterator object i from c as follows
    a. if c is a basic type that is iterable, then i is the result of calling c.iterator()
    b. if c is an object and c belongs to Iterable<T> for some T, then i is the result of calling c.__iterator()
3. call i.next() resulting in a value n
4. if n is nil, then terminate execution of the foreach statement
5. match typed-binding-pattern to n.value causing assignments to any variables that were created in typed-binding-pattern
6. execute block-stmt with the variable bindings from step 5 in scope; in the course of so doing
    a. the execution of a break-stmt terminates execution of the foreach statement
    b. the execution of a continue-stmt causes execution to proceed immediately to step 3
7. go back to step 3

In step 2, the compiler will give an error if the static type of expression is not suitable for 2a or 2b.

In step 5, the typed-binding-pattern is used unconditionally, and the compiler will check that the static types guarantee that the match will succeed. If the typed-binding-pattern uses var, then the type will be inferred from the type of `expression`.

While statement

```
while-stmt := while expression block-stmt
```

A while statement repeatedly executes a block statement so long as a boolean-valued expression evaluates to true.

In more detail, a while statement executes as follows:

1. evaluate expression;
2. if expression evaluates to false, terminate execution of the while statement;
3. execute block-stmt; in the course of so doing
    a. the execution of a break-stmt results in termination of execution of the while statement
    b. the execution of a continue-stmt causes execution to proceed immediately to step 1
4. go back to step 1.

The static type of `expression` must be boolean.

Continue statement

```
continue-stmt := continue ;
```

A continue statement is only allowed if it is lexically enclosed within a while-stmt or a foreach-stmt. Executing a continue statement causes execution of the nearest enclosing while-stmt or foreach-stmt to jump to the end of the outermost block-stmt in the while-stmt or foreach-stmt.

Break statement

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

A break statement is only allowed if it is lexically enclosed within a while-stmt or a foreach-stmt. Executing a break statement causes the nearest enclosing while-stmt or foreach-stmt to terminate.

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 a subtype of 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.

Note *The design of exception handling, including the throw expression, is likely to change in a future version.*

Throw statement

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

A throw statement is used to throw an exception. The static type of `expression` must be a subtype of error.

Note *The design of exception handling, including the throw expression, is likely to change in a future version.*

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.

Note *The design of shared data access is likely to change in a future version.*

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-defn
   | outside-method-defn
   | 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-init ;
```

The scope of variables declared in a module-var-decl is the entire module. The semantics of var-decl-init is defined in the section for “Local variable declaration statements”.

Note *It is planned to add a way to make variables and values be immutable, and require this to be used for public module variables.*

Module endpoints

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

`type-descriptor` must be an object type implementing the endpoint interface.

Function definitions

```
function-defn :=
   metadata
   function-defn-quals
   function identifier function-signature function-body
function-defn-quals := [extern] [public]| public extern
function-body := function-body-block | ;
function-body-block :=
   { 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.

The presence of `extern` means that the implementation of the function is not provided in the Ballerina source module. The `function-body` must be a bare semicolon if and only if `function-defn-quals` includes `extern`.

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-block
```

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. Metadata

```
metadata := [DocumentationString] annots
```

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 [<annot-attach-point (, annot-attach-point)*>]
      identifier type-descriptor ;
annot-attach-point :=
   service
   | resource
   | function
   | object
   | type
   | endpoint
   | parameter

annots := annotation*
annotation :=
   @ variable-reference mapping-constructor-expr?
```

Documentation

A documentation string is an item of metadata that can be associated with module-level Ballerina constructs and with method declarations. The purpose of the documentation strings for a module is to enable a programmer to use the module. Information not useful for this purpose should be provided in in comments.

A documentation string has the format of one or more lines each of which has a # optionally preceded by blank space.

The documentation statement is used to document various Ballerina constructs.

```
DocumentationString := DocumentationLine +
DocumentationLine := BlankSpace* # [Space] DocumentationContent
DocumentationContent := (^ 0xA)* 0xA
BlankSpace := Tab | Space
Space := 0x20
Tab := 0x9
```

A `DocumentationString` is recognized only at the beginning of a line. The content of a documentation string is the concatenation of the `DocumentationContent` of each `DocumentationLine` in the `DocumentationString`. Note that a single space following the # is not treated as part of the DocumentationContent.

The content of a `DocumentationString` is parsed as Ballerina Flavored Markdown (BFM). BFM is also used for a separate per-module documentation file, conventionally called `Module.md`.

Ballerina Flavored Markdown

Ballerina Flavored Markdown is GitHub Flavored Markdown, with some additional conventions.

In the documentation string attached to a function or method, there must be documentation for each parameter, and for the return value if the return value is not nil. The documentation for the parameters and a return value must consist of a Markdown list, where each list item must have the form `ident - doc`, where ident is either the parameter name or return, and doc is the documentation of that parameter or of the return value.

The documentation for an object must contain a list of fields rather than parameters. Private fields should not be included in the list.

BFM also provides conventions for referring to Ballerina-defined names from within documentation strings in a source file. An identifier in backticks `` `X` ``, when preceded by one of the following words:

- `type`
- `endpoint`
- `service`
- `variable`
- `var`
- `annotation`
- `module`
- `function`
- `parameter`

is assumed to be a reference to a Ballerina-defined name of the type indicated by the word. In the case of `parameter`, the name must be unqualified and be the name of a parameter of the function to which the documentation string is attached. For other cases, if the name is unqualified it must refer to a public name of the appropriate type in the source file's module; if it is a qualified name M:X, then the source file must have imported M, and X must refer to a public name of an appropriate type in M. BFM also recognizes `` `f()` `` as an alternative to ``function `f` ``. In both cases, f can have any of the following forms (where m is a module import, x is a function name, t is an object type name, and y is a method name):

```
x()
m:x()
t.y()
m:t.y()
```

Example

```
# Adds parameter `x` and parameter `y`
# + x - one thing to be added
# + y - another thing to be added
# + return - the sum of them
```

```
    function add (int x, int y) returns int  { return x + y; }
```

12. Security

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

13. 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 754-2008
- GitHub Markdown

B. Changes since previous versions

Summary of changes from 0.970 to 0.980

1. The decimal type has been added.
2. There are no longer any implicit numeric conversions.
3. The type of a numeric literal can be inferred from the context.
4. The error type is now a distinct basic type.
5. The byte type has been added as a predefined subtype of int; blobs have been replaced by arrays of bytes.
6. The syntax of string templates and xml literals has been revised and harmonized.
7. The syntax of anonymous functions has been revised to provide two alternative syntaxes: a full syntax similar to normal function definitions and a more convenient arrow syntax for when the function body is an expression.
8. The cases of a match statement are required to be exhaustive.
9. The + operator is specified to do string and xml concatenation as well as addition.
10. Bitwise operators have been added (<<, >>, >>>, &, |, ^, ~) rather than = after the argument name.
11. In a function call or method call, named arguments have changed to use `:`

12. A statement with `check` always handles an error by returning it, not by throwing it.
13. `check` is allowed in compound assignment statements.
14. Method names are now looked up differently from field names; values of types other than objects can now have built-in methods.
15. The `lengthof` unary expression has been removed; the length built-in method can be used instead.
16. The semantics of <T>expr have been specified.
17. The value space for tuples and arrays is now unified, in the same way as the value space for records and maps was unified. This means that tuples are now mutable. Array types can now have a length.
18. The `next` keyword has been changed to `continue`.
19. The syntax and semantics of destructuring is now done in a consistent way for the but expression, the match statement, the foreach statement, destructuring assignment statements and variable declarations.
20. The implied initial value is not used as a default initializer in variable declarations. A local variable whose declaration omits the initializer must be initialized by an assignment before it is used. A global variable declaration must always have an initializer. A new expression can be used with any reference type that has an implicit initial value.
21. Postfix increment and decrement statements have been removed.
22. The `...` and `..<` operators have been added for creating integer ranges; this replaces the foreach statement's special treatment of integer ranges.
23. An object type can be declared to be abstract, meaning it cannot be used with `new`.
24. By default, a record type now allows extra fields other than those explicitly mentioned; `T...` requires extra fields to be of type T and `!...` disallows extra fields.
25. In a mapping constructor, an expression can be used for the field name by enclosing the expression in square brackets (as in ECMAScript).
26. Integer arithmetic operations are specified to throw an exception on overflow.
27. The syntax for documentation strings has changed.
28. The deprecated construct has been removed (data related to deprecation will be provided by an annotation; documentation related to deprecation will be part of the documentation string).
29. The order of fields, methods and constructors in object types is no longer constrained.
30. A function or method can be defined as `extern`. The `native` keyword has been removed.

C. Future work

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

1. Generic types
2. Provide a way to control mutability.
3. Revise exception handling.
4. Compensating transactions

5. 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)
6. Incorporate privacy aware coding concepts