Ballerina Language Specification, v0.990:2019-01-16

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 are listed in Appendix C.)

Language and document status

The design of the Ballerina language is approaching stability.

Some language features described by this specification are less stable than the rest of the language. These are marked with either as having either “preview” or “experimental” status. Preview status means that we expect the final design to be close enough to the current design that it will be straightforward to update code that makes uses the current design to the final design. Experimental status means that we believe that we want to have similar functionality, but we are not yet confident about how close the final design will be to the feature as currently described.

In addition, we know there are many areas where the specification needs to provide more details about the semantics of the language.

Comments on the language design are welcome. 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 programming language intended for network distributed applications. It is a statically typed, concurrent programming language with all functionality expected of a modern, general purpose programming language. But it also has several unusual aspects that make it particularly suitable for its intended purpose.

First, it provides language constructs specifically for consuming and providing network services. Future versions of Ballerina will add language constructs for other functionality often needed by network distributed applications such as security, stream processing, distributed transactions and reliable messaging.

Second, it is designed to take advantage of sequence diagrams as a way of describing the interactions within network distributed applications. There is a close correspondence between the function-level concurrency-related syntax and sequence diagrams; this syntax is in effect a syntax for writing sequence diagrams. This makes it possible to provide an editable graphical representation of a function as a sequence diagram.

Third, it has a type system that is more flexible and allows for looser coupling that traditional statically typed languages. The type system 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 when combining data from multiple, independently-designed systems. In addition, the type system provides union types and open records. This flexibility allows the type system 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.

Ballerina is not a research language. 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. It also borrows ideas from many other existing programming languages including TypeScript, Go, Rust, D, Kotlin, TypeScript, Swift, Python and Perl.

Ballerina is designed for modern development practices with a module based development model with namespace management via module repositories, including a globally shared central repository. Module 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.

Ballerina is a "batteries included" language: it comes with a standard library, which includes not only the usual low-level, general-purpose functionality, but also support for a wide variety of network protocols, interface standards, data formats and authentication/authorization standards, which make writing secure, resilient distributed applications significantly easier than with other languages. The standard library is not specified in this document.

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 ordered 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 floating point numbers, which are not constructed from other values;
- structured values, like maps and arrays, which create structures from other values;
- behavioral values, like functions, which allow parts of Ballerina programs to be handled in a uniform way with other values

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. A reference value has an identity determined by its storage location. References make it possible for distinct members of a structure to refer to values that are identical, in the sense that they are stored in the same location. Thus values in Ballerina represent not just trees but graphs.

Simple values are inherently immutable because they have no identity distinct from their value. All basic types of structural values, with the exception of the XML, are mutable, meaning the value referred to by a particular reference can be changed. Whether a behavioural value is mutable depends on its basic type: some of the behavioural basic types allow mutation, and some do not. Mutation cannot change the basic type of a value. Mutation makes it possible for the graphs of references between values to have cycles.

Ballerina programs use types to categorize values both at compile-time and runtime. Types deal with an abstraction of values, which does not consider storage location or mutability. This abstraction is called a *shape*. A type denotes a set of shapes. A type S is a subtype of type T if the set of shapes denoted by S is a subset of the set of shapes denoted by T. Every value has a corresponding shape. A shape is specific to a basic type: if two values have different basic types, then they have different shapes. Since shapes do not deal with storage location, they have no concept of identity; shapes therefore represent trees rather graphs. For simple values, there is no difference between a shape and a value, except for one case: positive and negative zero of a floating point basic type are distinct values but have the same shape. There are two important relations between a value and a type:

- a value *looks like* a type at a particular point in the execution of a program if its shape at that point is a member of the type;
- a value *belongs to* a type if it looks like the type, and it will necessarily continue to look like the type no matter how the value is mutated.

For an immutable value, looking like a type and belonging to a type are the same thing.

When a Ballerina program declares a variable to have a compile-time type, this means that the Ballerina compiler together with the runtime system will ensure that the variable will only ever contain a value that belongs to the type. Ballerina also provides mechanisms that take a value that looks like a type and use it to create a value that belongs to a type.

Ballerina provides a rich variety of type descriptors, which programs use to describe types. For example, there is a type descriptor for each simple basic type; there is a type descriptor that describes as type as a union of two types; there is a type descriptor that uses a single value to describe a type that contains a single shape. This means that values can look like and belong to arbitrarily many types, even though they look like or belong to exactly one *basic* type.

Most types, including all simple basic types, have an implicit initial value, which is used to initialize structure members.

Most basic types of structured values (along with one basic type of 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. Experimental features are not included.

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	tuple where each member

		list	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		
	service		
	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	()
	anydata		()
	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

The multiple bit patterns that IEEE 754 treats as NaN are considered to be the same value in Ballerina. Positive and negative zero of a floating point basic type are distinct values, following IEEE 754, but are defined to have the same shape, so that they will usually be treated as being equal.

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

There are five basic types of structured value. First, there are three container basic types: list, mapping and table. Second, there is the xml basic type and the error basic type, which are both special in different ways.

Values of the container basic types are containers for other values, which are called their members. Containers are mutable: the members contained in a particular container can be changed. However, a container value can also be frozen at runtime or compile time, which prevents any change to its members. A frozen container value can refer only to immutable values: either other frozen values or values of basic types that are always immutable. Once frozen, a container value remains frozen. Values of basic type xml can also be frozen as described below.

The shape of the members of a container value contribute to the shape of the container. Mutating a member of a container can thus cause the shape of the container to change. A type descriptor for a container basic type describe the shape of the container in terms of the shapes of its members. A container has an inherent type, which is a type descriptor which is part of the container's runtime value. At runtime, the container prevents any mutation that might lead to the container having a shape that is not a member of its inherent type. Thus a container value belongs to a type if and only if that type is its inherent type or a subset of its inherent type.

A frozen container value belongs to a type if and only if the type contains the shape of the value. Thus after a container value is frozen, its inherent type does not provide additional information that cannot be derived from the value. In other words, freezing a container narrows its inherent type to a type that consists of just its current shape.

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

The following table summarizes the type descriptors for structured types.

	Integer index	String key
Basic type	list	mapping
Type descriptor with uniform member type	array	map
Type descriptor with separate member types	tuple	record

A value is defined to be *pure* if it either

- is a simple value, or
- is a structured value, all of whose members are also pure values.

A shape is pure if it is the shape of a pure value. A type is pure if it contains only pure shapes.

Lists

A list value is a container that keeps its members in an ordered 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. The shape of a list value is an ordered list of the shapes of its members.

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
```

The inherent type of a list value must be a `list-type-descriptor`. The inherent type of a list value determines a type T_i for a member with index *i*. The runtime system will enforce a constraint that a value written to index *i* will belong to type T_i. Note that the constraint is not merely that the value looks like T_i.

Both kinds of type descriptor are covariant in the types of their members.

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
   | constant-reference-expr
   | implied-array-length
implied-array-length := ! ...
```

A type `T[]` contains a list shape if all members of the list shape are in `T`. A type `T[n]` contains a list shape if in addition the length of the list shape is n.

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.)

A `constant-reference-expr` in an `array-length` must evaluate to a non-negative integer. 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”); its meaning is the same as if the length was specified explicitly.

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.

When a list value has an inherent type that is an array without an array-length, then 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

Note also that `T[n]` is a subtype of `T[]`, and that if `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`; this is a consequence of the definition of subtyping in terms of subset inclusion of the corresponding sets of shapes.

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 type (T_1, T_2, … T_n) contains a list shape S if for each T_i the i-th member shape of S is in T_i and the length of S is n.

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.

Mappings

A mapping value is a container where each member has a key, which is a string, that uniquely identifies within the mapping. We use the term *field* to mean the member together its key; the name of the field is the key, and the value of the field is that value of the member; no two fields in a mapping value can have the same name.

The shape of a mapping value is an unordered collection of field shapes one for each field. The field shape for a field f has a name, which is the same as the name of f, and a shape, which is the shape of the value of f.

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 the value of the field. 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.

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

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

The inherent type of a mapping value must be a `mapping-type-descriptor`. The inherent type of a mapping value determines a type T_f for the value of the field with name *f*. The runtime system will enforce a constraint that a value written to field *f* will belong to type T_f. Note that the constraint is not merely that the value looks like T_f.

Both kinds of type descriptor are covariant in the types of their members.

Map types

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

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

A type `map<T>` contains a mapping shape *m* if every field shape in *m* has a value shape that is in `T`.

A value belonging to type `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 a type separately for the value of each field.

```
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 shape must satisfy for it to be a member of the described type. The constraint depends on whether ? is present:

- if ? is not present, then the constraint is that the mapping value shape must have a field shape with the specified field-name and with a value shape that is a member of the specified type-descriptor; this is called a required field;
- if ? is present, then the constraint is that if the mapping value shape has a field shape with the specified field-name, then its value shape must be a member of 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.

For a mapping value shape and a record-type-descriptor, let the extra field shapes be the field shapes of the mapping value shapes whose names are not the same as field-name of any individual-field-descriptor. The `record-rest-descriptor` specifies the constraint that the extra fields shapes must satisfy in order for the mapping value shape to be a member of the described type, as follows:

- if the `record-rest-descriptor` is empty, then the value shape of the extra field shapes must be pure
- if the `record-rest-type` is !, then there must not be any extra field shapes
- if the `record-rest-type` is a type descriptor T, then the value shape of every extra field shape must be a member of T

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.

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.

[Preview] 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. The type descriptor for the column must be a pure type. If a column is part of a primary key, then the type descriptor for the column must also allow only non-nil simple values.

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 mapping values will belong to 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?

]

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.

Error

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

An error value belongs to the error basic type, which is a basic type which is distinct from other structured types and is used only for representing errors. An error value contains the following information:

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

An error value is immutable, since the detail is frozen and the stack trace information cannot be changed.

The shape of an error value consists of the shape of the reason and the shape of the detail; the stack trace is not part of the shape. A type descriptor error<r, d> contains an error shape if r contains the shape's reason, and d, if present, contains the shape's detail. The bare type error contains all error shapes. The reason-type-descriptor must be a subtype of string; the detail-type-descriptor must be a subtype of `record { }` (which is equivalent to `map<anydata|error>`).

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 :=
   error-type-descriptor
   | function-type-descriptor
   | object-type-descriptor
   | future-type-descriptor
   | service-type-descriptor
   | stream-type-descriptor
   | typedesc-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 := const-expr
rest-param := type-descriptor ... [param-name]
```

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

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

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

A function can be passed any number of arguments. Each argument is passed in one of three ways: as a positional argument, as a named argument or as a rest argument. A positional argument is identified by its position relative to other positional arguments. A named 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 initializing an object of this type, in particular it provides the method 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 :=
   object-type-quals object {
      object-member-descriptor*    }
object-type-quals :=
   [abstract] [client] | [client] abstract
object-member-descriptor :=
```

```
    object-field-descriptor
    | object-method
    | object-type-reference
```

If `object-type-quals` contains the keyword `abstract`, 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.

If `object-type-quals` contains the keyword `client`, then the object type is a client object type. A client object type may have remote methods; other objects types must not.

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][remote]
    function method-name function-signature ;
method-defn :=
    metadata
    method-defn-quals
    function method-name function-signature method-body
method-defn-quals :=
   [extern] [visibility-qual] [remote]
   | [visibility-qual] extern [remote]
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`.

A method that is declared or defined with the remote qualifier is a remote method. A remote method is allowed only in a client object. A remote method is invoked using a different syntax from a non-remote method.

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 :=
   [visibility-qual] [remote]
   function qual-method-name function-signature function-body-block
qual-method-name := object-type-name . method-name
```

The method-defn-quals on an outside-method-defn must contain the same qualifiers (both `remote` and visibility qualifiers) as the declaration of the method within the object.

Initialization

A non-abstract object type provides a way to initialize an object of the type. An object is initialized by:

1. allocating storage for the object
2. initializing each field with its implicit initial value, if there is one defined for the type of the field
3. initializing the methods of the object using the type's method definitions
4. calling the object's `__init` method, if there is one

The return type of the `__init` method must be a subtype of the union of error and nil; if `__init` returns an error, it means that initialization of the object failed. The `__init` method can declare parameters in the same way as any other method.

When an object is initialized to its implicit initial value, no parameters are available, and `__init` must not return an error. Thus a non-object type has an implicit initial value unless it has a `__init` method with a non-nil return type or required parameters.

At any point in the body of a `__init` method, the compiler determines which fields are potentially uninitialized. A field is potentially uninitialized at some point if it is a field of a type that does not have an implicit initial value and it is not definitely assigned at that point. It is a compile error if a `__init` method:

- accesses a field at a point where it is potentially initialized, or
- at a point where there is any potentially uninitialized value
    - returns nil, or

- uses the self variable other than to access or modify the value of a field.

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 the `__init` method cannot be specified to be `private`. The visibility of a method or field of an abstract object type cannot be private.

Futures

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

A future value represents a value to be returned by a named worker. A future value belongs to a type `future<T>` if the value to be returned belongs to T.

[Preview] Services

```
service-type-descriptor := service
```

A service value contains resources and methods. A service method is similar to an object method. A resource is a special kind of method, with associated configuration data, that is invoked in response to network input received by a Listener.

All service values belong to the type `service`.

It is planned that a future version of Ballerina will provide a mechanism that allows more precise typing of services. In the meantime, implementations can use annotations on type definitions to support this.

[Preview] 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. T must be a pure type.

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

Type descriptors

```
typedesc-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
   | anydata-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 := simple-const-expr
```

A singleton type is a type containing a single shape. A singleton type is described using an compile-time constant expression for a single value: the type contains the shape of that value. Note that it is possible for the variable-reference within the simple-const-expr to reference a structured value.

The implicit initial value for a singleton type is the single value used to specify the type.

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 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 alwpuays be ().

Any Type

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

The type descriptor `any` describes the type consisting of all values other than errors. A value belongs to the any type if and only if its basic type is not error. Thus all values belong to the type `any|error`. Note that a structure with members that are errors belongs to the `any` type.

Anydata type

```
anydata-type-descriptor := anydata
```

The type descriptor `anydata` describes the type of all pure values other than errors. The type `anydata` contains a shape if and only if the shape is pure and is not the shape of an error value.

Thus the type `anydata|error` is the supertype of all pure types. The type `anydata` is equivalent to the union

```
() | boolean | int | float | decimal | string
| (anydata|error)[] | map<anydata|error>
```

```
    | xml | table
```

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
```
`json` means `() | boolean | int | float | decimal | string | json[] | map<json>`

Built-in abstract 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>.

Listener

The Listener type is defined as follows.

```
abstract object {
   function __attach (service s, string? name = ()) returns error?;
   function __start () returns error?;
   function __stop () returns error?;
}
```

Note that if an implementation does precise service typing using annotations on type definitions, it will need to treat Listener as being parameterized in the precise service type that is used to the first argument to `__attach`.

Abstract operations

These section specifies a number of operations that can be performed on values. These operations are for internal use by the specification. These operations are named in CamelCase to distinguish them from built-in methods.

Freeze

Freeze(v) is defined for any pure value v. The result of Freeze(v) is always v. It recursively freezes any structural values and their members, thus maintaining the invariant that frozen values can only refer to immutable values. Freeze(v) for a value v is already frozen or is of

an immutable type (i.e. a simple type of error) just returns v without modifying it. Freeze(v) must terminate for any pure value v, even if v has cycles.

Clone

Clone(v) is defined for any pure value v. It performs a deep copy, recursively copying all structural values and their members. Clone(v) for a value v that is frozen or of an immutable type (ie a simple type or error) returns v. If v is a container, Clone(v) has the same inherent type as v. The graph of references of Clone(v) must have the same structure as that of v. This implies that the number of distinct references reachable from Clone(v) must be the same as the number of distinct references reachable from v. Clone(v) must terminate for any pure value v, even if v has cycles.

Clone(v) cannot be implemented simply by recursively calling Clone on all members of v. Rather Clone must maintain a map that records the result of cloning each reference value. When a Clone operation starts, this map as empty. When cloning a reference value, it must use the result recorded in the map if there is one.

UnfrozenClone

UnfrozenClone works the same as Clone except for its treatment of frozen values. Whereas for a frozen value v, Clone(v) is v, UnfrozenClone(v) returns a new value that is not frozen and that has the same shape as v. UnfrozenClone handles values of immutable types (simple types and errors) in the same way as Clone. Like Clone, UnfrozenClone preserves graph structure.

SameShape

SameShape(v1, v2) is defined for any pure values v1, v2. It returns true or false depending of whether v1 and v2 have the same shape. SameShape(v1, v2) must terminate for any pure values v1 and v2, even if v1 or v2 have cycles. SameShape(v1, v2) returns true if v1 and v2 have the same shape, even if the graphs of references of v1 and v2 have different structures. If two values v1 and v2 have different basic types, then SameShape(v1, v2) will be false.

The possibility of cycles means that SameShape cannot be implemented simply by calling SameShape recursively on members. Rather SameShape must maintaom a mapping that records for each pair of references whether it is already in process of comparing those references. When a SameShape operation starts, this map is empty. Whenever it starts to compare two references, it should see whether it has already recorded that pair (in either order), and, if it has, proceed on the assumption that they compare equal.

SameShape(Clone(x), x) is guaranteed to be true for any pure value.

NumericConvert

NumericConvert(t, v) is defined if t is the typedesc for float, decimal or int, and v is a numeric value. It converts v to a value in t, or returns an error, according to the following table.

from \ to	float	decimal	int
float	unchanged	closest math value	round, error for NaN or out of int range
decimal	closest math value	unchanged	
int	same math value	same math value	unchanged

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
   | record-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. A binding pattern occurring in a typed-binding-pattern must also be irrefutable with respect to the type of value against which it is to be matched. In other words, the compiler will ensure that matching such a binding pattern against a value will never fail at runtime.

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
   | service-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
    | type-cast-expr
    | unary-expr
    | untaint-expr
    | multiplicative-expr
    | additive-expr
    | shift-expr
    | range-expr
    | numeric-comparison-expr
    | is-expr
    | equality-expr
    | binary-bitwise-expr
    | logical-expr
    | conditional-expr
    | check-expr
    | trap-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`	

`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`
`x is y`
	non

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

`x & y`	left
`x ^ y`	left
`x | y`	left
`x && y`	left
`x || y`	left
`x ?: y`	right
`x ? y : z`	right
`(x) => y`	right

Expression evaluation

When the evaluation of an expression completes normally, it produces a result, which is a value. The evaluation of an expression may also complete abruptly. There are two kinds of abrupt completion: check-fail and panic. With both kinds of abrupt completion there is an associated value, which always has basic type error.

The following sections describes how each kind expression is evaluated, assuming that evaluation of subexpressions complete normally. Except where explicitly stated to the contrary, expressions handle abrupt completion of subexpressions as follows. If in the course of evaluating an expression E, the evaluation of some subexpression E1 completes abruptly, then then evaluation of E also completes abruptly in the same way as E1.

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 if the evaluation of an expression at runtime completes normally, then the resulting value will belong to the static type. A type is also computed for check-fail abrupt completion, which will be a (possibly empty) subtype of error; however, for panic abrupt completion, no type is computed.

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 member type of the inherent type of a container.

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 type-cast-expr, 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.

Constant expressions

```
const-expr :=
   literal
   | array-constructor-expr
   | tuple-constructor-expr
   | mapping-constructor-expr
   | table-constructor-expr
   | string-template-expr
   | xml-expr
   | constant-reference-expr
   | type-cast-expr
   | unary-expr
   | multiplicative-expr
   | additive-expr
   | shift-expr
   | range-expr
   | numeric-comparison-expr
   | is-expr
   | equality-expr
   | binary-bitwise-expr
   | logical-expr
   | conditional-expr
   | ( const-expr )
```

Within a `const-expr`, any nested expression must also be a const-expr.

```
constant-reference-expr := variable-reference-expr
```

A `constant-reference-expr` must reference a constant defined with `module-const-decl`.

A const-expr is evaluated at compile-time. The result of a `const-expr` is automatically frozen. Note that the syntax of const-expr does not allow for the construction of error values.

```
simple-const-expr :=
  nil-literal
  | boolean
  | [Sign] int-literal
  | [floating-point-type] [Sign] floating-point-literal
  | string-literal
  | variable-reference
floating-point-type := < (decimal|float) >
```

A simple-const-expr is a restricted form of const-expr used in contexts where various forms of constructor expression would not make sense. Its semantics are the same as a const-expr.

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 inherent 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 inherent 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 inherent 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 inherent 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 inherent 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 inherent 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 inherent 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 inherent 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 inherent 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 inherent type of the constructed value. If the contextually expected type is neither a record type nor a map type, the inherent type is inferred from the types of the expressions occurring in the constructor.

[Preview] 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 inherent type of the constructed value.

For example,

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

Error constructor

```
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. The static type of reason-expr must be a subtype of string. The static type of detail-expr, if present, must be a subtype of `record { }` (i.e. it must be a pure type and a mapping). The result of evaluating reason-expr is cloned using the Clone abstract operation and then frozen using the Freeze abstract operation.

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.

Service constructor

```
service-constructor-expr := service service-body-block
service-body-block := { [service-method-defn* }
service-method-defn :=
   metadata
   service-method-defn-quals
   function identifier function-signature function-body-block
service-method-defn-quals :=
   [visibility-qual] | resource
```

A `service-constructor-expr` constructs a service value. The result of evaluating a service-constructor-expr is a value of type service. If a service-method-defn contains a resource qualifier, then it defines a resource, otherwise it defines a method. The self variable can be used in a function-body-block of a service-method-defn in the same way as for objects.

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. 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.

A new-object-expr initializes an object, passing the supplied arg-list to the object's `__init` method. 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 `__init` method. If the result of calling the `__init` method is an error value e, then the result of evaluating the new-object-expr is e; otherwise the result is the newly initialized object. If an object type does not have an `__init` method, then `new T()` is the same as `new T`.

If the type-descriptor in a new-expr specifies a type T, then the static type of a new-expr is T, except that, in the case of a new-object-expr, if the type of the __init method is E?, where E is a subtype of error, then the type of the new-object-expr is T|E.

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

Variable reference expression

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

A variable-reference can refer to a variable, a parameter, a constant (defined with a module constant declaration) or a type (defined with a type definition).

If the variable-reference references a type defined with a type definition, then the result of evaluating the variable-reference-expr is a typedesc value for that type.

Field access expression

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

{MISSING}

Index expression

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

{MISSING}
Evaluation of an index-expr that attempts to accessing a list with an out of bounds index will complete abruptly with a panic.

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 service or 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. A method-call-expr cannot be used to call a remote method; it can only be called by a remote-method-call-action. A method-call-expr cannot be used to invoke a resource.

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.

Type cast expression

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

A `type-cast-expr` casts the value resulting from evaluating `expression` to the type described by the type-descriptor, performing a numeric conversion if required.

A type-cast-expr is evaluated by first evaluating `expression` resulting in a value v. Let T be the type described by `type-descriptor`. If v belongs T, then the result of the type-cast-expr is v. Otherwise, if T includes shapes from exactly one basic numeric type N and v is belongs to another basic numeric type, then let n be NumericConvert(N, v); if n is not an error and n belongs to T, then the result of the type-cast-expr is n. Otherwise, the evaluation of the type-cast-expr completes abruptly with a panic.

Let T be the static type described by `type-descriptor`, and let TE be the static type of `expression`. Then the static type of the `type-cast-expr` is the intersection of T and TE', where TE' is TE with its numeric shapes transformed to take account to the possibility of the numeric conversion specified in the previous paragraph.

The `expression` is interpreted with `type-descriptor` as the contextually expected type.

Unary expression

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

[Preview] Untaint expression

```
untaint-expression := untaint expression
```

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

Type test expression

```
is-expr :=
   expression is type-descriptor
```

An is-expr tests where a value belongs to a type.

An is-expr is evaluated by evaluating the expression yielding a result v. If v belongs to the type denoted by type-descriptor, then the result of the is-expr is true; otherwise the result of the is-expr is false.

Equality expression

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

An equality-expr tests whether two values are equal. For all four operators, it is a compile time error if the intersection of the static types of the operands is empty.

The === operator tests for exact equality. The !== operator results in the negation of the result of the === operator. Two values are exactly equal if they have the same basic type T and

- if T is simple type, the values are identical values within the set of values allowed for T
- if T is a reference type, the values refer to the same storage

The == operator tests for deep equality. The != operator results in the negation of the result of the == operator. For both == and !=, both operands must have a static type that is pure. Two values v1, v2 are deeply equal if SameShape(v1, v2) is true.

Note that === and == are the same for simple values except as regards floating point zero: == treats positive and negative zero from the same basic type as equal whereas === treats

them as unequal. Both == and === treat two NaN values from the same basic type as equal. This means that none of these operators correspond to operations defined by IEEE 754-2008, because they do not treat NaN in the special way defined for those operations.

Binary bitwise expression

```
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.

Check expression

```
check-expression := check expression
```

Evaluates expression resulting in value v. If v has basic type error, then check-expression completes abruptly with a check-fail with associated value v. Otherwise evaluation completes normally with result v.

If the static type of expression e is T|E, where E is a subtype of error, then the static type of check e is T.

Trap expression

The trap expression stops a panic and gives access to the error value associated with the panic.

```
trap-expr := trap expression
```

Semantics are:

- Evaluate `expression` resulting in value v
    - If evaluation completes abruptly with panic with associated value e, then result of trap-exp is e
    - Otherwise result of trap-expr is v
- If type of `expr` is T, then type of `trap expr` is T|error.

7. Actions and statements

Actions

```
action :=
   remote-method-call-action
   | worker-receive-action
   | wait-action
   | flush-action
   | synchronous-send-action
   | check-action
   | trap-action
   | ( action )
action-or-expr := action | expression
check-action := check action
trap-action := trap action
```

Actions are an intermediate syntactic category between expressions and statements. Actions are similar to expressions, in that they yield a value. However, an action cannot be nested inside an expression; it can only occur as part of a statement or nested inside other actions. This is because actions are shown in the sequence diagram in the graphical syntax.

An action is evaluated in the same way as an expression. Static typing for actions is the same as for expressions.

A `check-action` and `trap-action` is evaluated in the same way as a `check-expr` and `trap-expr` respectively.

Function and worker execution

```
function-body-block :=
   { [default-worker-init, named-worker-decl+] default-worker }
default-worker-init := sequence-stmt
default-worker := sequence-stmt
named-worker-decl :=
   worker worker-name return-type-descriptor { sequence-stmt }
worker-name := identifier
```

A worker represents a thread of control running within a function. A statement is always executed in the context of a current worker. A worker is in one of three states: running, suspended or terminated. When a worker is in the terminated state, it has a termination value. A worker terminates either normally or abnormally. An abnormal termination results from a panic, and in this case the termination value is always an error. If the termination value of a normal termination is an error, then the worker is said to have terminated with failure; otherwise the worker is said to have terminated with success. Note that an error termination value resulting from a normal termination is distinguished from an error termination value resulting from an abnormal termination.

A function always has a single default worker, which is unnamed. When a function is called, the current worker is suspended, and a default worker for the called function is started. When the default worker terminates, the function returns to its caller, and the caller's worker is resumed. If the default worker terminates normally, then its termination value provides the return value of the function. If the default worker terminates abnormally, then the evaluation of the function call expression completes abruptly with a panic and the default worker's termination value provides the associated value for the abrupt completion of the function call. The function's return type is the same as the return type of the function's default worker.

The default worker of a function and the worker that called the function belong to the same thread of control, which is called a *strand*. Only one worker in a strand is running at any given time.

A function also has zero or more named workers. Each named worker runs on its own new strand. The termination of a function is independent of the termination of its named workers. The termination of a named worker does not automatically result in the termination of its function. When a default worker terminates causing the function to terminate, the function does not automatically wait for the termination of its named workers. There is a wait-action that allows one worker to explicitly wait for the termination of another worker.

A named worker has a return type. If the worker terminates normally, the termination value will belong to the return type. If the return type of a worker is not specified, it defaults to nil as for functions.

When a function has named workers, the default worker executes in three stages, as follows:

1. The statements in default-worker-init are executed.
2. All the named workers are started. Each named worker executes its sequence-stmt on its strand.
3. The statements in default-worker are executed. This happens without waiting for the termination of the named workers started in stage 2.

Variables declared in default-worker-init are in scope within named workers, whereas variables declared in default-worker are not.

The execution of a worker's sequence-stmt may result in the execution of a statement that causes the worker to terminate. For example, a return statement causes the worker to terminate. If this does not happen, then the worker terminates as soon as it has finished executing its sequence-stmt. In this case, the worker terminates normally, and the termination value is nil. In other words, falling off the end of a worker is equivalent to `return;`, which is in turn equivalent to `return ();`.

The parameters declared for a function are in scope in the function-body-block. They are implicitly final: they can be read but not modified. They are in scope for named workers as well as for the default worker.

The name of a worker is in-scope as a final local variable. The scope is the function-body-block with the exception of the default-worker-init. When the worker name is accessed using a variable-reference-expr, it has type future<T>, where T is the return type of the worker.

In the above, function includes method, and function call includes method call.

Statement execution

```
statement :=
   action-stmt
   | block-stmt
   | local-var-decl-stmt
   | xmlns-decl-stmt
   | assignment-stmt
   | compound-assignment-stmt
   | destructuring-assignment-stmt
   | call-stmt
   | if-else-stmt
   | match-stmt
   | foreach-stmt
```

```
    | while-stmt
    | break-stmt
    | continue-stmt
    | fork-stmt
    | panic-stmt
    | lock-stmt
    | async-send-stmt
    | return-stmt
    | transaction-stmt
    | transaction-control-stmt
```

The execution of any statement may involve the evaluation of actions and expressions, and the execution of substatements. The following sections describes how each kind of statement is evaluated, assuming that the evaluation of those expressions and actions completes normally, and assuming that the execution of any substatements does not cause termination of the current worker. Except where explicitly stated to the contrary, statements handle abrupt completion of the evaluation of expressions and actions as follows. If in the course of executing a statement, the evaluation of some expression or action completes abruptly with associated value e, then the current worker is terminated with termination value e; if the abrupt termination is a check-fail, then the termination is normal, otherwise the termination is abnormal. If the execution of a substatement causes termination of the current worker, then the execution of the statement terminates at that point.

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

A `sequence-stmt` executes its statements sequentially. A `block-stmt` is executed by executing its `sequence-stmt`.

Fork statement

```
fork-stmt := fork { named-worker-decl+ }
```

The fork statement starts one or more named workers, which run in parallel with each other, each in its own new strand.

Variables and parameters in scope for the fork-stmt remain in scope within the workers (similar to the situation with parameters and workers in a function body).

The scope of the worker name declared in a named-worker-decl includes both other workers in the same fork-stmt and the block containing the fork-stmt starting from the point immediately after the fork-stmt. When a worker-name is in scope it can be accessed using a variable-reference-expr, resulting in a value of type future<T>, where T is the return type of that worker.

Wait action

A wait-action waits for one or more workers to terminate, and gives access to their termination values.

```
wait-action :=
   single-wait-waction
   | multiple-wait-action
   | alternate-wait-action

wait-future-expr := expression but not mapping-constructor-expr
```

A wait-future-expr is used by a wait-action to refer to the worker to be waited for. Its static type must be future<T> for some T. It can use a variable reference to refer to an in-scope named-worker-decl, which will be treated as a reference to a variable of type future<T> where T is the return value of the worker.

Note that it is only possible to wait for a named worker, which will start its own strand. It is not possible to wait for a default worker, which runs on an existing strand.

A mapping-constructor-expr is not recognized as a wait-future-expr (it would not type-check in any case).

Single wait action

```
single-wait-action := wait wait-future-expr
```

A single-wait-action waits for a single future.

A single-wait-action is evaluated by evaluating wait-future-expr resulting in a value f, which must be of basic type future. It then waits until the strand of the future has terminated. If the strand terminates normally, the single-wait-action completes normally with the termination value of the strand as the result. Otherwise, the single-wait-action completes abruptly with a panic, with the associated value being the termination value of the strand, which will be an error.

If the static type of the wait-future-expr is future<T> , then the static type of the single-wait-action is then T.

Multiple wait action

```
multiple-wait-action := wait { wait-field (, wait-field)* }
wait-field :=
  variable-name
  | field-name : wait-future-expr
```

A multiple-wait-action waits for multiple futures, returning the result as a record.

A wait-field that is a variable-name `v` is equivalent to a wait-field `v: v`, where `v` must be the name of an in-scope named worker.

A multiple-wait-action is evaluated by evaluating all of the wait-future-exprs resulting in a value of type future for each wait-field. It then waits for all of these futures. If all the futures complete normally, then it constructs a record with a field for each wait-field, whose name is the field-name and whose value is the completion value of the strand.

Alternate wait action

```
alternate-wait-action := wait wait-future-expr (| wait-future-expr)+
```

An alternate-wait-action waits for one of multiple futures to terminate.

An alternate-wait-action is evaluated by first evaluating all of the wait-future-exprs, resulting in a set of future values. It then starts waiting for all of the futures. As soon as one of the futures completes normally with a non-error value v, the alternate-wait-action completes normally with result v. If all of the futures complete normally with an error, then it completes normally with result e, where e is the termination value of the last future to complete.

If the static type of the wait-future-exprs is future<T1>, future<T2>, …, future<Tn>, then the static type of the alternative-wait action is T1|T1|. . .Tn

Worker message passing

Messages can be sent between workers.

Sends and receives are matched up at compile-time. This allows the connection between the send and the receive to be shown in the sequence diagram. It is also guarantees that any sent message will be received, provided that neither the sending nor the receiving worker terminate abnormally or with an error.

Messages can only be sent between workers that are peers of each other. The default worker and the named workers in a function are peers of each other. The workers created in a fork-stmt are also peers of each other. The workers created in a fork-stmt are not peers of the default worker and named workers created by a function.

```
peer-worker := worker-name | default
```

A worker-name refers to a worker named in a named-worker-decl, which must be in scope; default refers to the default worker. The referenced worker must be a peer worker.

Each worker maintains a separate logical queue for each peer worker to which it sends messages; a sending worker sends a message by adding it to the queue; a receiving worker

receives a message by removing it from the sending worker's queue for that worker; messages are removed in the order in which they were added to the queue.

Send action and send statement

```
sync-send-action := expression ->> peer-worker
async-send-stmt := expression -> peer-worker ;
```

The sync-send-action and async-send-stmt send a message to another worker. In both cases, the message is the result of applying the Clone abstract operation to the result of evaluating expr. The message is sent to the worker identified by peer-worker.

In both cases, the message is added to the message queue maintained by the sending worker for messages to be sent to the sending worker. Conceptually, the message is added to the queue even if the receiving worker has already terminated.

For each async-send-stmt and sync-send-action S, the compiler determines a unique corresponding receive-action R, such that a message sent by S will be received by R, unless R's worker has terminated abnormally or with failure. It is a compile-time error if this cannot be done. The compiler determines a *failure type* for the corresponding receive-action. If the receive-action was not executed and its worker terminated normally, then the termination value of the worker will belong to the failure type. The failure type will be a (possibly empty) subtype of error.

The execution of the async-send-stmt completes as soon as the message is added to the queue. A subsequent flush action can be used to check whether the message was received.

The sync-send-action is evaluated by waiting until the receiving worker either executes a receive action that receives the queued message or terminates. The evaluation of sync-send-action completes as follows:

- if the queued message was received, then normally with result nil;
- otherwise
    - if the receiving worker terminated with failure, then normally with the result being the the termination value of the receiving worker, which will be an error;
    - if the receiving worker terminated abnormally, then abruptly with a panic, with the associated value being the termination value of the receiving worker.

The static type of the sync-send-action is F|() where F is the failure type of the corresponding receive action. If F is empty, then this static type will be equivalent to ().

The contextually expected type used to interpret `expression` is the contextually expected type from the corresponding receive-action.

If the receive-action corresponding to an async-send-stmt has a non-empty failure type, then it is a compile-time error unless it can be determined that a sync-send-action or a flush-action will be executed before the sending worker terminates with success.

If a worker W is about to terminate normally and there are messages still to be sent in a queue (which must be the result of executing an async-send-stmt), then the worker waits until the messages have been received or some receiving worker terminates. If a receiving worker R terminates without the message being received, R must have terminated abnormally, because the rule in the preceding paragraph. In this case, W terminates abnormally and W will use R's termination value as its termination value.

Receive action

```
receive-action := single-receive-action | multiple-receive-action
```

Single receive action

```
single-receive-action := <- peer-worker
```

A single-receive-action receives a message from a single worker.

For each single-receive-action R receiving from worker W, the compiler determines a *corresponding send set*. The corresponding send set S is a set of sync-send-actions and async-send-stmts in W, such that

- in any execution of W that terminates successfully, exactly one member of S is executed and is executed once only
- if R is evaluated, it will receive the single message sent by a member of S, unless W has terminated abnormally or with failure.

The compiler terminates a failure type for the corresponding send set. If no member of the corresponding send set was executed/evaluated and the sending worker terminated normally, then the termination value of the sending worker will belong to the failure type. The failure type will be a (possibly empty) subtype of error.

A single-receive-action is evaluated by waiting until there is a message available in the queue or the sending worker terminates. The evaluation of single-receive-action completes as follows:

- if a message becomes available in the queue, then the first available message is removed and the evaluation completes normally with the result being that message;
- otherwise
    - if the sending worker terminated with failure, then normally with the result being the the termination value of the sending worker, which will be an error;
    - if the sending worker terminated abnormally, then abruptly with a panic, with the associated value being the termination value of the sending worker.

The static type of the single-receive-action is T|F where T is the union of the static type of the expressions in the corresponding send set and F is the failure type of the corresponding send set.

Multiple receive action

```
multiple-receive-action :=
```

```
   <-  { receive-field (, receive-field)* }
receive-field :=
   peer-worker
   | field-name : peer-worker
```

A multiple-receive-action receives a message from multiple workers.

A peer-worker can occur at most once in a multiple-receive-action.

A receive-field consisting of a peer-worker `W` is equivalent to a field `W:W`.

The compiler determines a corresponding send set for each receive field, in the same way as for a single-receive-action. A multiple-receive-action is evaluated by waiting until there is a message available in the queue for every peer-worker. If any of the peer workers W terminate before a message becomes available, then the evaluation of the multiple-receive-action completes as follows

- if the sending worker terminated with failure, then normally with the result being the the termination value of the sending worker, which will be an error;
- if the sending worker terminated abnormally, then abruptly with a panic, with the associated value being the termination value of the sending worker.

Otherwise, the result of the evaluation of multiple-receive-action completes by removing the first message from each queue and constructing a record with one field for each receive-field, where the value of the record is the message received.

The contextually expected typed for the multiple-receive-action determines a contextually expected type for each receive-field, in the same way as for a mapping constructor. The contextually expected type for each receive-field provides the contextually expected type for the expression in each member of the corresponding send set.

The static type of multiple-receive-action is R|F where

- R is a record type, where R is determined in the same way as for a mapping constructor, where the static type of each field comes from the union of the static types of the expressions in each member of the corresponding send set and the contextually expected type is the contextually expected type of the multiple-receive-action
- F is the union of the failure types for the corresponding send set for each receive-field

Flush action

```
flush-action := flush [peer-worker]
```

If peer-worker is specified, then flush waits until the queue of messages to be received by peer-worker is empty or until the peer-worker terminates.

Send-receive correspondence for async-send-stmt implies that the queue will eventually become empty, unless the peer-worker terminates abnormally or with failure. The evaluation of flush-action completes as follows:

- if the queue of messages is empty, then normally with result nil;
- otherwise
    - if the peer-worker terminated with failure, then normally with the result being the the termination value of the peer-worker, which will be an error;
    - if the peer-worker terminated abnormally, then abruptly with a panic, with the associated value being the termination value of the peer-worker.

If the flush-action has a preceding async-send-stmt without any intervening sync-send-action or other flush-action, then the static type of the flush-action is F|(), where F is the failure type of the receive-action corresponding to that async-send-stmt. Otherwise, the static type of the flush-action is nil.

If peer-worker is omitted, then the flush-action flushes the queues to all other workers. In this case, the static type will be the union of the static type of flush on each worker separately.

Send-receive correspondence

This section provides further details about how compile-time correspondence is established between sends and receive. This is based on the concept of the index of a message in its queue: a message has index n in its queue if it is the nth message added to the queue during the current execution of the worker.

- A send action/statement has index i in its queue if the message that it adds to the queue is always the i-th message added to the queue during the execution of its worker. It is a compile time error if a send statement or action does not have an index in its queue.
- A receive action has index i in a queue if any message that it removes from the queue is always the i-th message removed from the queue during the execution of its worker. It is a compile time error if a receive action does not have an index in each of its queues.
- A send action/statement and a receive action correspond if they have the same index in a queue.
- It is a compile time error if two or more receive actions have the same index in a queue.
- A send action/statement is in the same send set as another send action/statement if they have the same index in a queue. It is allowed for a send set to have more than one member.
- The maximum index that a receive action has in a queue must be the same as the maximum index that a send action or statement has in that queue.
- It is a compile time error if it is possible for a worker to terminate with success before it has executed all its receive actions.
- It is a compile time error if it is possible for a worker to terminate with success before it has executed one member from every send set.

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 [final] typed-binding-pattern = action-or-expr ;
```

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.

A local-init-var-decl-stmt is executed by evaluating the action-or-expr 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 the action-or-expr; if the local-init-var-decl-stmt includes final, the precise type is used, and otherwise the broad type is used.

If `final` is specified, then the variables declared must not be assigned to outside the local-init-var-decl-stmt.

```
local-no-init-var-decl-stmt :=
   annots [final] 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. If `final` is specified, then the variable must not be assigned more than once.

Implicit variable type narrowing

Usually the type of a reference to a variable is determined by the variable's declaration, either explicitly specified by a type descriptor or inferred from the static type of the initializer. In addition, this section defines cases where a variable is used in certain kinds of boolean expression in a conditional context, and it can be proved at compile time that the value stored in local variable or parameter will, within a particular region of code, always belong to a type that is narrower that the static type of the variable. In these cases, references to the variable within particular regions of code will have a static type that is narrower that the variable type.

Given an expression E with static type boolean, and a variable x with static type Tx, we define how to determine

- a narrowed type for x implied by truth of E and
- a narrowed type for x implied by falsity of E

based on the syntactic form of E as follows.

- If E has the form x is T, then
    - the narrowed type for x implied by truth of E is the intersection of Tx and T
    - the narrowed type for x implied by falsity of E is Tx with T removed
- If E has the form x == E1 or E1 == x where the static type of E1 is an expression whose static type is a singleton simple type T1, then
    - the narrowed type for x implied by truth of E is the intersection of Tx and T1
    - the narrowed type for x implied by falsity of E is Tx with T1 removed
- If E has the form x != E1 or E1 != x where the static type of E1 is an expression whose static type is a singleton simple type T1, then
    - the narrowed type for x implied by truth of E is Tx with T1 removed
    - the narrowed type for x implied by falsity of E is the intersection of Tx and T1
- If E has the form !E1, then
    - the narrowed type for x implied by truth of E is the narrowed type for x implied by falsity of E1
    - the narrowed type for x implied by falsity of E is the narrowed type for x implied by truth of E1
- If E has the form E1 && E2
    - the narrowed type for x implied by truth of E is the intersection of T1 and T2, where T1 is the narrowed type for x implied by the truth of T1 and T2 is the narrowed type for x implied by the truth of T2
    - the narrowed type for x implied by falsity of E is T1|T2, where T1 is the narrowed type for x implied by the falsity of E1, and T2 is the intersection of T3 and T4, where T3 is the narrowed type for x implied by the truth of E1 and T4 is the narrowed type for x implied by the falsity of E2
- If E has the form E1 || E2, then
    - the narrowed type for x implied by truth of E is T1|T2, where T1 is the narrowed type for x implied by the truth of E1, and T2 is the intersection of T3 and T4, where T3 is the narrowed type for x implied by the falsity of E1 and T4 is the narrowed type for x implied by the truth of E2
    - the narrowed type for x implied by falsity of E is the intersection of T1 and T2, where T1 is narrowed type for x implied by the falsity of E1 and T2 is the narrowed type for x implied by the falsity of E2
- If E has any other form, then
    - the narrowed type for x implied by the truth of E is Tx
    - the narrowed type for x implied by the falsity of E is Tx

Narrowed types apply to regions of code as follows:

- in an expression E1 || E2, the narrowed types implied by the falsity of E1 apply within E2
- in an expression E1 && E2, the narrowed types implied by the truth of E1 apply within E2

- in an expression E1 ? E2 : E3, the narrowed types implied by the truth of E1 apply within E2 and the narrowed types implied by the falsity of E1 apply within E3
- in a statement if E1 { B1 } else { B2 }, the narrowed types implied by the truth of E1 apply within B1 and the narrowed types implied by the falsity of E1 apply within E2
- in a match-clause P if E => { B }, the narrowed types implied by the truth of E apply within B
- the narrowed type for a variable x no longer applies as soon as there is a possibility of x being assigned to

XML namespace declaration statement

```
xmlns-decl-stmt := xmlns xml-namespace [ as identifier ] ;
xml-namespace := simple-const-expr
```

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. The static type of the simple-const-expr must be a subtype of string.

Assignment statement

```
assignment-stmt := lhs = action-or-expr ;
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 type that v's inherent type requires for field x, 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 member type of the inherent 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 action-or-expr ;
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 = action-or-expr ;
```

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.

Action statement

```
action-stmt := action ;
```

An action-stmt is a statement that is executed by evaluating an action and discarding the resulting value, which must be nil. It is a compile-time error if the static type of an action in an action-stmt is not nil.

Call statement

```
call-stmt := call-expr ;
call-expr :=
   function-call-expr
   | method-call-expr
   | check call-expr
   | trap call-expr
```

A call-stmt is executed by evaluating call-expr as an expression and discarding the resulting value, which must be nil. It is a compile-time error if the static type of the call-expr in an call-stmt is not nil.

Remote method call action

```
remote-method-call-action :=
  expr -> method-name ( arg-list )
```

Calls a remote method. This works the same as a method call expression, except that it is used only for a method with the `remote` modifier.

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 action-or-expr { match-clause+ }
match-clause :=
  match-pattern-list [match-guard] => block-stmt
match-guard := if expression
```

A match statement selects a block statement to execute based on which patterns a value matches.

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. a match of match-pattern against v is attempted
    b. if the attempted match fails, the execution of the match-stmt continues to the next match-clause
    c. if the attempted match succeeds, then the variables in match-pattern are creates
    d. if there is a match-guard, then the expression in match-guard is executed resulting in a value b
    e. if b is false, then the execution of the match-stmt continues to the next match-clause
    f. otherwise, the block-stmt in the match-clause is executed
    g. execution of the match-stmt completes

The scope of any variables created in a match-pattern-list of a match-clause is both the match-guard, if any, and the block-stmt in that match-clause. The static type of the expression in match-guard must be a subtype of boolean.

```
match-pattern-list :=
  match-pattern (| match-pattern)*
```

A match-pattern-list can be matched against a value. An attempted match can succeed or fail. A match-pattern-list is matched against a value by attempting to match each match-pattern until a match succeeds.

All the match-patterns in a given match-pattern-list must bind the same set of variables.

```
match-pattern :=
  var binding-pattern
  | ignore
  | const-pattern
  | structured-match-pattern
```

A match-pattern combines the destructuring done by a binding-pattern with the ability to match a constant value.

Note that an identifier can be interpreted in two different ways within a match-pattern:

- in the scope of a var, an identifier names a variable which is to be bound to a part of the matched value when a pattern match succeeds;
- outside the scope of a var, an identifier references a constant that a value must match for the pattern match to succeed.

A match-pattern must be linear: a variable that is to be bound cannot occur more than once in a match-pattern.

```
const-pattern := simple-const-expr
```

A const-pattern denotes a single value. Matching a const-pattern against a value succeeds if the value has the same shape as the value denoted by the const-pattern. A variable-reference in a const-pattern must refer to a constant. Successfully matching a const-pattern does not cause any variables to be created.

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

Foreach statement

```
foreach-stmt :=
   foreach typed-binding-pattern in action-or-expr 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. These variables are implicitly final.

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 a subtype of 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.

[Experimental] 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.*

Panic statement

```
panic-stmt := panic expression ;
```

A panic statement terminates the current worker abnormally. The result of evaluating `expression` provides the termination value of the worker.

The static type of `expression` must be a subtype of error.

Return statement

```
return-stmt := return [ action-or-expr ] ;
```

A return statement terminates the current worker normally.The result of evaluating the action-or-expr provides the termination value of the worker. If action-or-expr is omitted, then the termination value is nil.

8. Built-in methods

The following built-in methods are available.

Generic methods

The following methods are provided on a variety of different basic types, with a consistent semantics.

length

The length method returns an integer representing the number of items that a value contains, where the meaning of item depends on the basic type of value. The following list specifies the basic types for which it is provided, and the meaning on each basic type:

- string: is the number of code points in the string
- list: the number of members of the list
- mapping: the length of a mapping is the number of members in the mapping
- table: the number of rows in the table
- xml: the number of content items in this XML value, where each character, element, comment and processing instructions is counted as a single content item

iterator

The iterator method is provided for all iterable basic types. It returns a new object belonging to the Iterator<T> abstract type, where T depends on the basic type as follows:

- string: T is string; it will iterate over the substrings of length 1
- list: T is the union of the type of members of the list
- mapping: T is (string, M) where M is the union of the types of the members of the mapping
- table: T is a record representing a single row
- xml: T is string|xml

freeze, clone, unfrozenClone

The freeze, clone and unfrozenClone built-in methods work in a similar way. They are provided on all structural types other than error, and all basic types other than nil. When applied to a value v, the method starts by checking whether v is a pure value; if it is not, the

method returns an error. (If v belongs to type anydata, then it is guaranteed that v is a pure value.) Otherwise, the method returns Freeze(v), Clone(v) or UnfrozenClone(v).

The static type of the return type of applying one of these methods to an expression of type T is T if T is a subtype of anydata, and otherwise is T|error.

isFrozen

The isFrozen method is provided on all structural types other than error. When applied to a value v, it returns true if the value is frozen and false otherwise.

Floating point methods

The following built-in methods are provided on both the decimal and float basic types. They all return boolean.

- isFinite: true iff this is neither infinity nor NaN
- isInfinite: true iff this is infinity (plus or minus)
- isNaN: true iff this is NaN

Error methods

The following built-in methods are provided on the error basic type:

- reason: returns the error's reason string
- detail: returns the error's detail record as a frozen mapping
- stackTrace: returns an object representing the stack trace of the error

Function methods

The following built-in method is provided on the function basic type:

- call: `f.call(args)` means the same as `f(args)`; this allows f to be an arbitrary expression

Typedesc methods

The stamp and convert methods are provided on typedesc values. They both take a single parameter of type anydata, and return a value of type T|error where T is the type denoted by the typedesc value.

stamp

The stamp method takes a single argument of type anydata.

t.stamp(v) works as follows

1. It checks that v is a tree. If v is a reference value, and the graph structure of v is not a tree, then stamp returns an error.

2. If the shape of v is not a member of the set of shapes denoted by t (i.e. v does not look like t), then stamp will attempt to modify it so that it is by using numeric conversions (as defined by the NumericConvert operation) on structure members in v. If this fails or can be done in more than one way, then stamp will return an error. Frozen structures will not be modified, nor will new structures be created.
3. At this point, v looks like t. Now stamp narrows the inherent type of v, and recursively of any members of v, so that the v belongs to t, and then returns v. Any frozen values in v are left unchanged by this.

convert

The convert method takes a single argument of type anydata. It is similar to stamp, but it does not modify its argument.

t.convert(v) works as follows:

1. It checks that v is a tree. If v is a reference value, and the graph structure of v is not a tree, then convert returns an error.
2. convert now creates a new value that has the same shape as v, except possibly for differences in numeric types, but belongs to type

9. Module-level declarations

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

The import declarations must come before other declarations; apart from this, the order of the definitions and declarations at the top-level of a module is not significant.

```
module-part := import-decl* other-decl*
other-decl := metadata other-decl-item
other-decl-item :=
   type-defn
   | module-const-decl
   | module-var-decl
   | listener-decl
   | function-defn
   | outside-method-defn
   | service-decl
   | xmlns-decl-stmt
   | annotation-decl
```

Module and program execution

Creating a Ballerina program from a collection of modules requires that one module is identified as the *root* module. The modules comprising the program consist of the root module together with transitive closure of its imported modules.

If the root module of the program has a function called `main`, then it must be public and its return type must be a subtype of `error?`. Any parameters must have a type that is a subtype of anydata; they are not restricted to strings.

If the root module does not have a function called `main`, then there must be at least one service declared in at least one of the program's modules.

When a Ballerina program is executed, the arguments to main come from the command line made available by the operating system. The format of the command line is operating system dependent, but is typically some sort of string or array of strings. The implementation should make use of the declared parameter types of main in attempting to convert the supplied command line into an argument list acceptable to main. It may also make use of annotations on main.

The execution of a program can terminate in one of three ways: success, failure or abnormal. In the failure and abnormal cases, there is an associated error value. For the failure case, the type of the error value can be determined at compile time.

The execution of a Ballerina has two or three consecutive phases:

1. init
2. main
3. listen

There is always an init phase. There must be at least one of the main or listen phases, and there may be both.

In the init phase, every module comprising the program is initialized, in such an order than a module is always initialized after all the modules it imports have been initialized. If a module fails to initialize, then the program terminates.

The main phase happens only if the root module has a main function. In this case, the main function is executed with arguments converted from the supplied command line. If the main function returns an error, then the program terminates.

If the main function returns nil and there is no listener registered with any module, then the program terminates successfully.

Otherwise the listen phase happens. In this phase, all the listeners that have been registered with any of the modules are started, by calling their `__start` method. (If there was no main function, then the successful initialization of the required service declaration will ensure that at least one listener will have been registered.)

The listen phase continues to execute until either the program explicitly exits, by calling a standard library function, or the user explicitly requests the termination the program using an implementation-dependent operating system facility (such as a signal on a POSIX system). In the latter case, the program will call the __stop request on each registered listener before terminating.

Import declaration

```
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 definition

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

Module variable declaration

```
module-var-decl :=
   metadata
   [final]
   typed-binding-pattern = expression ;
```

The scope of variables declared in a module-var-decl is the entire module. Note that module variables are not allowed to be public. If final is specified, then it is not allowed to assign to the variable. If the typed-binding-pattern uses `var`, then the type of the variable is inferred from the static type of `expression`; if the module-var-decl includes `final`, the precise type is used, and otherwise the broad type is used.

Module constant declaration

```
module-const-decl :=
   metadata
   public? const [type-descriptor] identifier = const-expr ;
```

A module-const-decl declares a compile-time constant. A compile-time constant is an named immutable value, known at compile-time. A compile-time constant can be used like a variable, and can also be referenced in contexts that require a value that is known at compile-time, such as in a type-descriptor or in a match-pattern.

The type of the constant is the singleton type containing just the shape of the value named by the constant. The type of the constant determines the static type of a variable-reference-expr that references this constant.

If type-descriptor is present, then it provides the contextually expected type for the interpretation of const-expr. It is a compile-time error if the static type of const-expr is not a subtype of that type. The type-descriptor must specify a type that is a subtype of anydata. Note that the type-descriptor does not specify the type of the constant, although the type of the constant will all be a subtype of the type specified by the type-descriptor.

Listener declaration

```
listener-decl :=
   metadata
   public? listener [type-descriptor] identifier = expression ;
```

A `listener-decl` defines a module listener.

A module listener is an object value that belongs to the Listener abstract object type and is managed as part of the module's lifecycle. A module may have multiple listeners.

A module-listener can be referenced by a variable-reference, but cannot be modified. It is this similar to a final variable declaration, except that it also registers the value with the module as a listener.

A module listener has a static type, which must be a subtype of the Listener type. If the type-descriptor is present it specifies the module listener's static type; if it is not present, the the static type of the listener is the static type of `expression`.

Function definition

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

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`.

Service declaration

```
service-decl :=
  metadata
  service [identifier] on expression-list service-body-block
expression-list := expression (, expression)*
```

Creates a service and attaches it to one or more listeners.

This works as follows:

- service-body is the same as what goes inside a service constructor
- expression is evaluated to an object obj that matches the Listener interface or an error; if its an error then module initialization panics
- obj is registered with the module as one of its endpoints (registering the same object multiple times is the same as registering it once)
- service-body-block is evaluated as in a service-constructor to get a service value s; the parameter type of obj.__attach determines the contextually expected type
- identifier, if present, works as a final variable whose value is s
- s is attached to obj using obj.__attach; the identifier is supplied as an argument to obj.attach along with s (nil if not present)
- if __attach returns an error, then module initialization panics

10. [Experimental] Querying

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

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.

Streaming queries

```
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.

11. [Experimental] Transactions

```
transaction-stmt := transaction trans-conf? block-stmt trans-retry?
transaction-control-stmt := retry-stmt | abort-stmt
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
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].

12. Metadata

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

[Preview] 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
   | listener
   | 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; }
```

A. References

- Unicode
- XML
- JSON
- RFC 3629 UTF-8
- IEEE 754-2008
- GitHub Markdown

B. Changes since previous versions

Summary of changes from 0.980 to 0.990

Structural types and values

1. Concepts relating to typing of mutable structural values have been changed in order to make type system sound.
2. The `match` statement has been redesigned.
3. The `but` expression has been removed.
4. The `is` expression for dynamic type testing has been added.
5. The type-cast-expr <T>E now performs unsafe type casts.The only conversions it performs are numeric conversions.
6. The `anydata` type has been added, which is a union of simple and structural types.
7. Records are now by default open to `anydata|error`, rather than `any`.
8. Type parameters for built-in types (map, stream, future), which previously defaulted to any, are now required.
9. The type parameter for json (e.g. json<T>) is not allowed any more.
10. Type for table columns are restricted to subtype of anydata|error.
11. There are now two flavors of equality operator: == and != for deep equality (which is allowed only for `anydata`), and === and !== for exact equality.
12. There is a built-in clone operation for performing a deep copy on values of type anydata.
13. There is a built-in freeze operation for making structural values deeply immutable.
14. Compile-time constants (which are always a subtype of anydata and frozen) have been added.
15. Singleton types have been generalized: any compile-time constant can be made into a singleton value.
16. Variables can be declared final, with a similar semantic to Java.
17. Errors are now immutable.
18. Module variables are not allowed to be public: only compile-time constants can be public.

Error handling

19. The `any` type no longer includes `error`.
20. `check` is now an expression.
21. Exceptions have been replaced by panics
    a. the `throw` statement has been replaced by the `panic` statement
    b. the `try` statement has been replaced by the `trap` expression
22. Object constructors (which could not return errors) have been replaced by `__init` methods (which can return errors).

Concurrency

23. Workers in functions have been redesigned. In particular, workers now have a return value.
24. The `done` statement has been removed.
25. The fork/join statement has been redesigned.
26. A syntactic category between expression and statement, called action, has been added.
27. A synchronous message send action has been added.
28. A flush action has been added to flush asynchronously sent messages.
29. A wait action has been added to wait for a worker and get its return value.
30. Futures have been unified with workers. A future<T> represents a value to be returned by a named worker.

31. Error handling of message send/receive has been redesigned.

Endpoints and services

32. Client endpoints have been replaced by client objects, and actions on client endpoints have been replaced by remote methods on client objects. Remote methods are called using a remote method call action, which replaces the action invocation statement.
33. Module endpoint declaration has been replaced by module listener declaration, which uses the Listener built-in object type.
34. The service type has been added as a new basic type of behavioural value, together with service constructor expressions for creating service values.
35. Module service definitions have been redesigned.

Miscellaneous changes

36. Public/private visibility qualifiers must be repeated on an outside method definition.

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. 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