The Ceylon Language

Say more, more clearly

Version: Final release draft (1.0)

Table of Contents

Welcome to Ceylon

This project is the work of a team of people who are fans of Java and of the Java ecosystem, of its practical orientation, of its culture of openness, of its developer community, of its roots in the world of business computing, and of its ongoing commitment to portability. However, we recognize that the language and class libraries, designed more than 15 years ago, are no longer the best foundation for a range of today's business computing problems. We further recognize that Java failed in one environment it was originally promoted for: the web browser.

The goal of this project is to make a clean break with the legacy Java SE platform, by improving upon the Java language and class libraries, and by providing a modular architecture for a new platform based upon the Java Virtual Machine. A further goal is to bridge the gap between the web client and server by supporting execution on JavaScript virtual machines.

Of course, we recognize that the ability to interoperate with existing Java code, thereby leveraging existing investment in the Java ecosystem, is a critical requirement of any successor to the Java platform.

Java is a simple language to learn and Java code is easy to read and understand. Java provides a level of typesafety that is appropriate for business computing and enables sophisticated tooling with features like refactoring support, code completion, and code navigation. Ceylon aims to retain the overall model of Java, while getting rid of some of Java's warts, and improving upon Java's facilities for creating abstractions and writing generic libraries and frameworks.

Ceylon has the following goals:

- to be appropriate for large scale development, but to also be *fun*,
- to execute on the JVM, and on JavaScript virtual machines, and to interoperate with Java and JavaScript code,
- to provide language-level modularity,
- to be easy to learn for Java and C# developers,
- to eliminate some of Java's verbosity, while retaining its readability—Ceylon does *not* aim to be the most concise/cryptic language around,
- to provide an elegant and more flexible syntax to support frameworks, declarative programming, and meta-programming, and, in particular
- to provide a declarative syntax for expressing hierarchical information like user interface definition, externalized data, and system configuration, thereby eliminating Java's dependence upon XML,
- to support and encourage a more functional style of programming with immutable objects and first class functions, alongside the familiar imperative mode,
- to expand compile-time typesafety with compile-time safe handling of null values, compile-time safe typecasts, and a more typesafe approach to reflection, and
- to make it easy to *get things done*.

Unlike other alternative JVM languages, Ceylon aims to completely replace the legacy Java SE class libraries.

Therefore, the Ceylon SDK provides:

- a compiler that compiles Ceylon and Java source to Java bytecode, and cross-compiles Ceylon to JavaScript,
- command-line tooling for compiling modules and documentation, and managing modules in module repositories,
- Eclipse-based tooling for developing, compiling, testing, and debugging programs written in Ceylon,
- a module runtime for modular programs that execute on the Java Virtual Machine, and
- a set of class libraries that provides much of the functionality of the Java SE platform, together with the core functionality of the Java EE platform.

Chapter 1. Introduction

This document defines the syntax and semantics of the Ceylon language. The intended audience includes compiler implementors, interested parties who wish to contribute to the evolution of the language, and experienced developers seeking a precise definition of language constructs. However, in light of the newness of the language, we will begin with an overview of the main features of the language and SDK. A brief introduction to programming in the language may be found at the following address:

```
http://ceylon-lang.org/documentation/tour/
```

1.1. Language overview

Ceylon is a general-purpose programming language featuring a syntax similar to Java and C#. It is imperative, statically-typed, block-structured, object-oriented, and higher-order. By *statically-typed*, we mean that the compiler performs extensive type checking, with the help of type annotations that appear in the code. By *object-oriented*, we mean that the language supports user-defined types and features a nominative type system where a type is a set of named attributes and operations, and that it supports inheritance and subtype polymorphism. By *higher-order*, we mean that every referenceable program element (every attribute, every operation, and every type) is also a value. By *block-structured*, we mean to say that the language features lexical scoping and an extremely regular recursive syntax for declarations and statements.

Ceylon improves upon the Java language and type system to reduce verbosity and increase typesafety compared to Java and C#. Ceylon encourages a more functional, somewhat less imperative style of programming, resulting in code which is easier to reason about, and easier to refactor.

1.1.1. Runtime and platform

Ceylon programs execute in any standard Java Virtual Machine or on any JavaScript virtual machine, and take advantage of the memory management and concurrency features of the virtual machine in which they execute. Ceylon programs are packaged into *modules* with well-defined inter-module dependencies, and always execute inside a runtime environment with module isolation.

The Ceylon compiler is able to compile Ceylon code that calls Java classes or interfaces, and Java code that calls Ceylon classes or interfaces. JavaScript code is able to interact with Ceylon classes and functions compiled to JavaScript. Via a special *dynamic* mode, code written in Ceylon may call functions defined natively in JavaScript.

Moreover, Ceylon provides its own native SDK as a replacement for the Java platform class libraries. Certain SDK modules depend upon services available only on the Java platform. Other SDK modules, including the core *language module*, are cross-platform and may also be used in a JavaScript virtual machine.

1.2. Type system

Ceylon, like Java and C#, features a hybrid type system with both subtype polymorphism and parameteric polymorphism. A type is either a stateless *interface*, a stateful *class*, a *type parameter*, or a *union* or *intersection* of other types. A class, interface, or type parameter may be defined as a subtype of another type. A class or interface may declare type parameters, which abstract the definition of the class or interface over all types which may be substituted for the type parameters.

Like C#, and unlike Java, Ceylon's type system is fully reified. In particular, generic type arguments are reified, eliminating many problems that result from type erasure in Java.

There are no primitive types or arrays in Ceylon—every Ceylon type can be represented within the language itself. So all values are instances of the type hierarchy root `Anything`, which is a class. However, the Ceylon compiler is permitted to optimize certain code to take advantage of the optimized performance of primitive types on the Java or JavaScript VM.

Furthermore, all types inferred or even computed internally by the Ceylon compiler are expressible within the language itself. Within the type system, *non-denoteable* types simply do not arise. The type system is based upon computation of *principal types*. There is no way to construct an expression which does not have a unique principal type expressible within the language. The principal type of an expression is a subtype of all other types to which the expression could be soundly assigned.

1.2.1. Mixin inheritance

Ceylon supports a restricted form of multiple inheritance, often called *mixin inheritance*. A class must extend exactly one other class. But a class or interface may satisfy (extend or implement) an arbitrary number of interfaces.

Classes hold state and define logic to initialize that state when the class is instantiated. A concrete class is a class that contains only concrete member definitions. Concrete classes may be directly instantiated. An abstract class may contain formal member declarations. Abstract classes may not be instantiated.

Interfaces may define concrete members, but may not hold state (references to other objects) or initialization logic. This restriction helps eliminate the problems traditionally associated with multiple inheritance. Ceylon never performs any kind of "linearization" of the supertypes of a type. Interfaces may not be directly instantiated.

1.2.2. Algebraic types, self types, and type families

Ceylon does not feature Java-style enumerated types as a first-class construct. Instead, any abstract type may specify its *cases*—an enumerated list of instances and/or subtypes. This facility is used to simulate both enumerated types and functional-style "algebraic" (sum) types.

```
interface Identity of Person | Organization { ... }
```

A closely related feature is support for *self types* and *type families*. A self type is a type parameter of an abstract type (like `Comparable`) which represents the type of a concrete instantiation (like `String`) of the abstract type, within the definition of the abstract type itself. In a type family, the self type of a type is declared not by the type itself, but by a containing type which groups together a set of related types.

1.2.3. Simplified generics

Ceylon does not support Java-style wildcard type parameters, raw types, or any other kind of existential type. And the Ceylon compiler never even uses any kind of "non-denotable" type to reason about the type system. So generics-related error messages are understandable to humans.

Instead of wildcard types, Ceylon features *declaration-site variance*. A type parameter may be marked as covariant or contravariant by the class or interface that declares the parameter.

Ceylon has a somewhat more expressive system of generic type constraints with a cleaner, more regular syntax. The syntax for declaring constraints on a type parameter looks very similar to a class or interface declaration. Ceylon supports upper bound type constraints and also *enumerated bounds*.

```
interface Producer<out Value, in Rate>
        given Value satisfies Object
        given Rate of Float|Decimal { ... }
```

1.2.4. Union and intersection types

A *union type*, for example `String|Number`, or *intersection type*, for example `Identifiable&List<String>`, may be formed from two or more types defined elsewhere.

Union types make it possible to write code that operates polymorphically over types defined in disparate branches of the type heirarchy without the need for intermediate adaptor classes. Intersection types make it possible to operate polymorphically over all subtypes of a list of types. Union and intersection types provide some of the benefits of structural ("duck") typing, within the confines of a nominative type system, and therefore certain Ceylon idioms are reminiscent of code written in dynamically-typed languages.

Union and intersection types play a central role in generic type argument inference and therefore underly the whole system of principal typing. For example, the following expression has type `HashMap<String,Integer|Float>`:

```
HashMap { "float"->0.0, "integer"->0 }
```

1.2.5. Type aliases and type inference

Type aliases and type inference help reduce the verbosity of code which uses generic types, eliminating the need to repeatedly specify generic type arguments.

A *type alias* is similar to a C-style `typedef`.

```
interface Strings => Sequence<String>;
```

```
alias Number => Integer|Float|Whole|Decimal;
```

Local *type inference* allows a type annotation to be eliminated altogether. The type of a block-local value or function is inferred from its definition if the keyword `value` or `function` occurs in place of the type declaration.

```
value name = person.name;
```

```
function sqrt(Float x) => x^0.5;
```

The type of a control-structure variable also may be inferred.

```
for (n in 0..max) { ... }
```

Ceylon features an especially elegant approach to generic type argument inference, making it possible to instantiate container types, even inhomogeneous container types, without the need to explicitly mention any types at all.

```
value numbers = { -1, 0, -1, -1.0, 0.0, 1.0 };
```

By limiting type inference to local declarations, Ceylon ensures that all types may be inferred by the compiler in a single pass of the source code. Type inference works in the "downward" and "outward" directions. The compiler is able to determine the type of an expression without considering the rest of the statement or declaration in which it appears.

1.2.6. Metaprogramming

In other statically typed languages, runtime metaprogramming, or *reflection*, is a messy business involving untypesafe strings and typecasting. Even worse, in Java, generic type arguments are erased at runtime, and unavailable via reflection. Ceylon, uniquely, features a *typesafe metamodel* and typed *metamodel expressions*. Since generic type arguments are reified at runtime, the metamodel fully captures generic types at both compile time and execution time.

Ceylon's support for program element *annotations* is based around this metamodel. Annotations are more flexible than in Java or C#, and have a much cleaner syntax.

Ceylon does not support macros or any other kind of compile-time metaprogramming.

1.3. Object-oriented programming

The primary unit of organization of an object-oriented program is the class. But Ceylon, unlike Java, doesn't require that *every* function or value belong to a class. It's perfectly normal to program with a mix of classes and toplevel functions. Contrary to popular belief, this does not make the program less object-oriented. A function is, after all, an object.

1.3.1. Class initialization and instantiation

Ceylon does not feature any Java-like constructor declaration and so each Ceylon class has a parameter list, and exactly one *initializer*—the body of the class. This helps reduce verbosity and results in a more regular block structure.

```
class Point(Float x, Float y) { ... }
```

The Ceylon compiler guarantees that the value of any attribute of a class is initialized before it is used in an expression.

A class may be a member of an outer class. Such a member class may be refined (overridden) by a subclass of the outer class. Instantiation is therefore a polymorphic operation in Ceylon, eliminating the need for a factory method in some circumstances.

Ceylon provides a streamlined syntax for defining an anonymous class which is only instantiated in exactly the place it is

defined. Among other uses, the `object` declaration is useful for creating singleton objects or method-local interface implementations.

```
object origin extends Point(0.0, 0.0) {}
```

1.3.2. Functions, methods, values, and attributes

Functions and *values* are the bread and butter of programming. Ceylon functions are similar to Java methods, except that they don't need to belong to a class. Ceylon values are polymorphic, and abstract their internal representation, similar to C# properties.

```
String name => firstName + " " + lastName;
```

The Ceylon compiler guarantees that any value is initialized before it is used in an expression.

A function belonging to a type is called a *method*. A value belonging to a type is called an *attribute*. There are no `static` members. Instead, *toplevel* functions and values are declared as direct members of a package. This, along with certain other features, gives the language a more regular block structure.

By default, an attribute or value may not be assigned a new value after its initial value has been specified. Mutable attributes and variable values must be explicitly declared using the `variable` annotation.

Ceylon does not support function overloading. Each method of a type has a distinct name.

1.3.3. Defaulted parameters and variadic parameters

Instead of method and constructor overloading, Ceylon supports parameters with default values and *variadic* parameters.

```
void addItem(Product product, Integer quantity=1) { ... }
```

```
String join(String* strings) { ... }
```

Furthermore, a generic method may be used to emulate parameter type overloading.

```
Number sum<Number>(Number* numbers) given Number of Integer | Float { ... }
```

Therefore, a single method in Ceylon may emulate the signatures of several overloaded methods in Java.

1.3.4. First-class functions and higher-order programming

Ceylon supports first-class function types and higher-order functions, with minimal extensions to the traditional C syntax. A function declaration may specify a *callable parameter* that accepts references to other functions with a certain signature.

```
String find(Boolean where(String string)) { ... }
```

The argument of such a callable parameter may be either a reference to a named function declared elsewhere, or a new function defined inline as part of the method invocation. A function may even return an invocable reference to another function.

```
value result = { "C", "Java", "Ceylon" }.find((String s) => s.size>1);
```

The type of a function is expressed within the type system as an instantiation of the interface `Callable`. The parameter types are expressed as a tuple type. So the type of the function `(String s) => s.size>1` is `Callable<Boolean,[String]>`, which may be abbreviated to `Boolean(String)`.

Methods and attributes may also be used as functions.

```
value names = people.map(Person.name);
```

```
value values = keys.map(keyedValues.get);
```

1.3.5. Naming conventions, annotations, and inline documentation

The Ceylon compiler enforces the traditional Smalltalk naming convention: type names begin with an initial uppercase letter—for example, `Liberty` or `RedWine`—member names and local names with an initial lowercase letter or underscore—for example, `blonde`, `immanentize()` or `boldlyGo()`.

This restrictions allows a much cleaner syntax for program element annotations than the syntax found in either Java or C#. Declaration "modifiers" like `shared`, `abstract`, and `variable` aren't keywords in Ceylon, they're ordinary annotations.

```
"Base type for higher-order abstract stuff."
shared abstract class AbstractMetaThingy() { ... }
```

The documentation compiler reads inline documentation specified using the `doc` annotation.

1.3.6. Named arguments and tree-like structures

Ceylon's named argument lists provide an elegant means of initializing objects and collections. The goal of this facility is to replace the use of XML for expressing hierarchical structures such as documents, user interfaces, configuration and serialized data.

```
Html page = Html {
    Head {
        title="Hello";
    };
    Body {
        P {
            css = "greeting";
            "Hello, World!"
        };
    };
}
```

An especially important application of this facility is Ceylon's built-in support for program element annotations.

1.3.7. Modularity

Toplevel declarations are organized into *packages* and *modules*. Ceylon features language-level access control via the `shared` annotation which can be used to express block-local, package-private, module-private, and public visibility for a program element. There's no equivalent to Java's `protected`.

A module corresponds to a versioned packaged archive. Its *module descriptor* expresses its dependencies to other modules. The tooling and execution model for the language is based around modularity and module archives.

1.4. Language module

The Ceylon language module defines a set of built-in types which form the basis for several powerful features of the language. The following functionality is defined as syntactic "sugar" that makes it easier and more convenient to interact with the language module.

1.4.1. Operators and operator polymorphism

Ceylon features a rich set of operators, including most of the operators supported by C and Java. True operator overloading is not supported. However, each operator is defined to act upon a certain class or interface type, allowing application of the operator to any class which extends or satisfies that type. For example, the `+` operater may be applied to any class that satisfies the interface `Summable`. This approach is called *operator polymorphism*.

1.4.2. Numeric and character types

Ceylon's numeric type system is much simpler than C, C# or Java, with exactly two built-in numeric types (compared to six in Java and eleven in C#). The built-in types are classes representing integers and floating point numbers. `Integer` and `Float` values are 64 bit by default, and may be optimized for 32 bit architectures via use of the `small` annotation.

The module `ceylon.math` provides two additonal numeric types representing arbitrary precision integers and arbitrary pre-

cision decimals.

Ceylon has `Character` and `String` classes, and, unlike Java or C#, every character is a full 32-bit Unicode codepoint. Conveniently, a `String` is a `List<Character>`.

1.4.3. Compile-time safety for optional values and type narrowing

There is no primitive null in Ceylon. The null value is an instance of the class `Null` that is not assignable to user-defined class or interface types. An *optional type* is a union type like `Null|String`, which may be abbreviated to `String?`. An optional type is not assignable to a non-optional type except via use of the special-purpose `if (exists ... )` construct. Thus, the Ceylon compiler is able to detect illegal use of a null value at compile time. Therefore, there is no equivalent to Java's `NullPointerException` in Ceylon.

Similarly, there are no C-style typecasts in Ceylon. Instead, the `if (is ... )` and `case (is ... )` constructs may be used to narrow the type of an object reference without risk of a `ClassCastException`. The combination of `case (is ... )` with algebraic types amounts to a kind of language-level support for the visitor pattern.

Alternatively, *type assertions*, written `assert (is ... )` or `assert (exists ... )` may be used to narrow the type of a reference.

1.4.4. Iterable objects and comprehensions

The interface `Iterable` represents a stream of values, which might be evaluated lazily. This interface is of central importance in the language module, and so the language provides a syntactic abbreviation for the type of an iterable object. The abbreviation `{String*}` means `Iterable<String>`. There is a convenient syntax for instantiating an iterable object, given a list of values:

```
{String*} words = {"hello", "world", "goodbye"};
```

A *nonempty iterable* is an iterable object which always produces at least one value. A nonempty iterabe type is written `{String+}`. Distingushing nonempty streams of values lets us correctly express the type of functions like `max()`:

```
{Float+} oneOrMore = .... ;
{Float*} zeroOrMore = .... ;
Float maxOfOneOrMore = max(oneOrMore); //never null
Float? maxOfZeroOrMore = max(zeroOrMore); //might be null
```

Comprehensions are an expressive syntax for filtering and transforming streams of values. For example, they may be used when instantiating an iterable object or collection:

```
value adults = { for (p in people) if (p.age>18) p.name };
```

```
value peopleByName = HashMap { for (p in people) p.name->p };
```

Comprehensions are evaluated lazily.

1.4.5. Sequences and tuples

Sequences are Ceylon's version of arrays. However, the `Sequential` interface does not provide operations for mutating the elements of the sequence—sequences are considered immutable. Because this interface is so useful, a type like `Sequential<String>` may be abbreviated to `[String*]`, or, for the sake of tradition, to `String[]`.

A *nonempty sequence* is a kind of sequence which always has at least one element. A nonempty sequence type is written `[String+]`. The special-purpose `if (nonempty ... )` construct narrows a sequence type to a nonempty sequence type.

Tuples are a kind of sequence where the type of each element is encoded into the static type of the tuple. `Tuple` is just an ordinary class in Ceylon, but syntactic abbreviations let us write down tuple types in using a streamlined syntax. For example, `[Float,Float]` is a pair of `Floats`. There is also a convenient syntax for instantiating tuples and accessing their elements.

```
[Float,Float] origin = [0.0, 0.0];
Float x = origin[0];
Float y = origin[1];
Null z = origin[2]; //only two elements!
```

Chapter 2. Lexical structure

Every Ceylon source file is a sequence of Unicode characters. Lexical analysis of the character stream, according to the grammar specified in this chapter, results in a stream of tokens. These tokens form the input of the parser grammar defined in the later chapters of this specification. The Ceylon lexer is able to completely tokenize a character stream in a single pass.

2.1. Whitespace

Whitespace is composed of strings of Unicode `SPACE`, `CHARACTER TABULATION`, `FORM FEED (FF)`, `LINE FEED (LF)` and `CARRIAGE RETURN (CR)` characters.

```
Whitespace: " " | Tab | Formfeed | Newline | Return
```

```
Tab: "\{CHARACTER TABULATION}"
```

```
Formfeed: "\{FORM FEED (FF)}"
```

```
Newline: "\{LINE FEED (LF)}"
```

```
Return: "\{CARRIAGE RETURN (CR)}"
```

Outside of a comment, string literal, or single quoted literal, whitespace acts as a token separator and is immediately discarded by the lexer. Whitespace is not used as a statement separator.

Source text is divided into lines by *line-terminating character sequences*. The following Unicode character sequences terminate a line:

- `LINE FEED (LF)`,
- `CARRIAGE RETURN (CR)`, and
- `CARRIAGE RETURN (CR)` followed by `LINE FEED (LF)`.

2.2. Comments

There are two kinds of comments:

- a *multiline comment* begins with `/*` and extends until `*/`, and
- an *end-of-line comment* begins with `//` or `#!` and extends until the next line terminating character sequence.

Both kinds of comments can be nested.

```
LineComment: ("//"|"#!") ~(Newline | Return)* (Return Newline | Return | Newline)?
```

```
MultilineComment: "/*" (MultilineCommmentCharacter | MultilineComment)* "*/"
```

```
MultilineCommmentCharacter: ~("/"|"*") | ("/" ~"*") => "/" | ("*" ~"/") => "*"
```

The following examples are legal comments:

```
//this comment stops at the end of the line
```

```
/*
   but this is a comment that spans
   multiple lines
*/
```

```
#!/usr/bin/ceylon
```

Comments are treated as whitespace by both the compiler and documentation compiler. Comments may act as token separators, but their content is immediately discarded by the lexer and they are not visible to the parser.

2.3. Identifiers and keywords

Identifiers may contain upper and lowercase letters, digits and underscores.

```
LowercaseChar: LowercaseLetter | "_"
```

```
UppercaseChar: UppercaseLetter
```

```
IdentifierChar: LowercaseChar | UppercaseChar | Number
```

The lexer classifies Unicode uppercase letters, lowercase letters, and numeric characters depending on the general category of the character as defined by the Unicode standard. A `LowercaseLetter` is any character whose general category is `Ll`. An `UppercaseLetter` is any character whose general category is `Lu`, `Lt`, or `Lo`. A `Number` is any character whose general category is `Nd`, `Nl`, or `No`.

All identifiers are case sensitive: `Person` and `person` are two different legal identifiers.

The lexer distinguishes identifiers which begin with an initial uppercase character from identifiers which begin with an initial lowercase character or underscore. Additionally, an identifier may be qualified using the prefix `\i` or `\I` to disambiguate it from a reserved word or to explicitly specify whether it should be considered an initial uppercase or initial lowercase identifier.

```
LIdentifier: LowercaseChar IdentifierChar* | "\i" IdentifierChar+
```

```
UIdentifier: UppercaseChar IdentifierChar* | "\I" IdentifierChar+
```

The following examples are legal identifiers:

```
Person
```

```
name
```

```
personName
```

```
_id
```

```
x2
```

```
\I_id
```

```
\Iobject
```

```
\iObject
```

```
\iclass
```

The prefix `\I` or `\i` is not considered part of the identifier name. Therefore, `\iperson` is just an initial lowercase identifier named `person` and `\Iperson` is an initial *uppercase* identifier named `person`.

The following reserved words are not legal identifier names unless they appear escaped using `\i` or `\I`:

```
assembly module package import alias class interface object given value assign void function new of
extends satisfies abstracts in out return break continue throw assert dynamic if else switch case for
while try catch finally then let this outer super is exists nonempty
```

Note: `abstracts`, `new`, and `let` are reserved for possible use in a future release of the language.

2.4. Literals

A *literal* is a single token that represents a Unicode character, a character string, or a numeric value.

2.4.1. Numeric literals

An *integer literal* may be expressed in decimal, hexadecimal, or binary notation:

```
IntegerLiteral: DecimalLiteral | HexLiteral | BinLiteral
```

A *decimal literal* has a list of digits and an optional magnitude:

```
DecimalLiteral: Digits Magnitude?
```

Hexadecimal literals are prefixed by `#`:

```
HexLiteral: "#" HexDigits
```

Binary literals are prefixed by `$`:

```
BinLiteral: "$" BinDigits
```

A *floating point literal* is distinguished by the prescence of a decimal point or fractional magnitude:

```
FloatLiteral: NormalFloatLiteral | ShortcutFloatLiteral
```

Most floating point literals have a list of digits including a decimal point, and an optional exponent or magnitude.

```
NormalFloatLiteral: Digits "." FractionalDigits (Exponent | Magnitude | FractionalMagnitude)?
```

The decimal point is optional if a fractional magitude is specified.

```
ShortcutFloatLiteral: Digits FractionalMagnitude
```

Decimal digits may be separated into groups of three using an underscore.

```
Digits: Digit+ | Digit{1..3} ("_" Digit{3})+
```

```
FractionalDigits: Digit+ | (Digit{3} "_")+ Digit{1..3}
```

Hexadecimal or binary digits may be separated into groups of four using an underscore. Hexadecimal digits may even be separated into groups of two.

```
HexDigits: HexDigit+ | HexDigit{1..4} ("_" HexDigit{4})+ | HexDigit{1..2} ("_" HexDigit{2})+
```

```
BinDigits: BinDigit+ | BinDigit{1..4} ("_" Digit{4})+
```

A digit is a decimal, hexadecimal, or binary digit.

```
Digit: "0".."9"
```

```
HexDigit: "0".."9" | "A".."F" | "a".."f"
```

```
BinDigit: "0"|"1"
```

A floating point literal may include either an *exponent* (for scientific notation) or a *magnitude* (an SI unit prefix). A decimal integer literal may include a magnitude.

```
Exponent: ("E"|"e") ("+"|"-")? Digit+
```

```
Magnitude: "k" | "M" | "G" | "T" | "P"
```

```
FractionalMagnitude: "m" | "u" | "n" | "p" | "f"
```

The magnitude of a numeric literal is interpreted as follows:

- `k` means `e+3`,
- `M` means `e+6`,
- `G` means `e+9`,
- `T` means `e+12`,
- `P` means `e+15`,
- `m` means `e-3`,
- `u` means `e-6`,
- `n` means `e-9`,
- `p` means `e-12`, and
- `f` means `e-15`.

The following examples are legal numeric literals:

```
69
```

```
6.9
```

```
0.999e-10
```

```
1.0E2
```

```
10000
```

```
1_000_000
```

```
12_345.678_9
```

```
1.5k
```

```
12M
```

```
2.34p
```

```
5u
```

```
$1010_0101
```

```
#D00D
```

```
#FF_FF_FF
```

The following are *not* valid numeric literals:

```
.33  //Error: floating point literals may not begin with a decimal point
```

```
1.  //Error: floating point literals may not end with a decimal point
```

```
99E+3  //Error: floating point literals with an exponent must contain a decimal point
```

```
12_34  //Error: decimal digit groups must be of length three
```

```
#FF.00  //Error: floating point numbers may not be expressed in hexadecimal notation
```

2.4.2. Character literals

A single *character literal* consists of a Unicode character, inside single quotes.

```
CharacterLiteral: "'" Character "'"
```

```
Character: ~("'" | "\") | EscapeSequence
```

A character may be identified by an *escape sequence*. Every escape sequence begins with a backslash. An escape sequence is replaced by its corresponding Unicode character during lexical analysis.

```
EscapeSequence: "\" (SingleCharacterEscape | "{" CharacterCode "}")
```

```
SingleCharacterEscape: "b" | "t" | "n" | "f" | "r" | "\" | """ | "'" | "`"
```

The single-character escape sequences have their traditional interpretations as Unicode characters:

- `\b` means `BACKSPACE`,
- `\t` means `CHARACTER TABULATION`,
- `\n` means `LINE FEED (LF)`,
- `\f` means `FORM FEED (FF)`,
- `\r` means `CARRIAGE RETURN (CR)`, and
- `\\`, `\``, `\'`, and `\"` mean `REVERSE SOLIDUS`, `GRAVE ACCENT`, `APOSTROPHE`, and `QUOTATION MARK`, respectively.

A Unicode codepoint escape is a four-digit or eight-digit hexadecimal literal, or a Unicode character name, surrounded by braces, and means the Unicode character with the specified codepoint or character name.

```
CharacterCode: "#" ( HexDigit{4} | HexDigit{8} ) | UnicodeCharacterName
```

Legal Unicode character names are defined by the Unicode specification.

The following are legal character literals:

```
'A'
```

```
'#'
```

```
' '
```

```
'\n'
```

```
'\{#212B}'
```

```
'\{ALCHEMICAL SYMBOL FOR GOLD}'
```

2.4.3. String literals

A character *string literal* is a sequence of Unicode characters, inside double quotes.

```
StringLiteral: """ StringCharacter* """
```

```
StringCharacter: ~( "\" | """ | "`" ) | "`" ~"`" | EscapeSequence
```

A string literal may contain escape sequences. An escape sequence is replaced by its corresponding Unicode character during lexical analysis.

A sequence of two backticks is used to delimit an interpolated expression embedded in a string template.

```
StringStart: """ StringCharacter* "``"
```

```
StringMid: "``" StringCharacter* "``"
```

```
StringEnd: "``" StringCharacter* """
```

A *verbatim string* is a character sequence delimited by a sequence of three double quotes. Verbatim strings do not contain escape sequences or interpolated expressions, so every character occurring inside the verbatim string is interpreted literally.

```
VerbatimStringLiteral: """"" VerbatimCharacter* """""
```

```
VerbatimCharacter: ~""" | """ ~""" | """ """ ~"""
```

The following are legal strings:

```
"Hello!"
```

```
"\{00E5}ngstr\{00F6}ms"
```

```
" \t\n\f\r,;:"
```

```
"\{POLICE CAR} \{TROLLEYBUS} \{WOMAN WITH BUNNY EARS}"
```

```
"""This program prints "hello world" to the console."""
```

The column in which the first character of a string literal occurs, excluding the opening quote characters, is called the *initial column* of the string literal. Every following line of a multiline string literal must contain whitespace up to the initial column. That is, if the string contents begin at the `n`th character in a line of text, the following lines must start with `n` whitespace characters. This required whitespace is removed from the string literal during lexical analysis.

2.5. Operators and delimiters

The following character sequences are operators and/or punctuation:

```
, ; ... { } ( ) [ ] ` ? . ?. *. = => + - * / % ^ ** ++ -- .. : -> ! && || ~ & | === == != < > <= >= <=> += -= /= *= %= |= &= ~= ||= &&=
```

Certain symbols serve dual or multiple purposes in the grammar.

Chapter 3. Type system

Every value in a Ceylon program is an instance of a type that can be expressed within the Ceylon language as a *class*. The language does not define any primitive or compound types that cannot, in principle, be expressed within the language itself.

A class, fully defined in §4.5 Classes, is a recipe for producing new values, called *instances* of the class (or simply *objects*), and defines the operations and attributes of the resulting values. A class instance may hold references to other objects, and has an identity distinct from these references.

Each class declaration defines a type. However, not all types are classes. It is often advantageous to write generic code that abstracts the concrete class of a value. This technique is called *polymorphism*. Ceylon features two different kinds of polymorphism:

- *subtype polymorphism*, where a subtype `B` inherits a supertype `A`, and
- *parametric polymorphism*, where a type definition `A<T>` is parameterized by a *generic type parameter* `T`.

Ceylon, like Java and many other object-oriented languages, features a single inheritance model for classes. A class may directly inherit at most one other class, and all classes eventually inherit, directly or indirectly, the class `Anything` defined in the module `ceylon.language`, which acts as the root of the class hierarchy.

A truly hierarchical type system is much too restrictive for more abstract programming tasks. Therefore, in addition to classes, Ceylon recognizes the following kinds of type:

- An *interface*, defined in §4.4 Interfaces, is an abstract type schema that cannot itself be directly instantiated. An interface may define concrete members, but these members may not hold references to other objects. A class may inherit one or more interfaces. An instance of a class that inherits an interface is also considered an instance of the interface.
- A *generic type parameter*, defined in §3.5 Generic type parameters, is considered a type within the declaration that it parameterizes. In fact, it is an abstraction over many types: it generalizes the declaration to all types which could be assigned to the parameter.
- An *applied type*, defined in §3.6 Generic type arguments, is formed by specifying arguments for the generic type parameters of a parameterized type.
- A *union type*, defined in §3.2.3 Union types, is a type to which each of an enumerated list of types is assignable.
- An *intersection type*, defined in §3.2.4 Intersection types, is a type which is assignable to each of an enumerated list of types.

Although we often use the term *parameterized type* or even *generic type* to refer to a parameterized type definition, it is important to keep in mind that a parameterized type definition is not itself a type. Rather, it is a *type constructor*, a function that maps types to types. Given a list of type arguments, the function yields an applied type.

In light of the fact that Ceylon makes it so easy to construct new types from existing types *without the use of inheritance*, by forming unions, intersections, and applied types, it's often useful to assign a name to such a type.

- A *type alias*, defined in §4.6 Type aliases, §4.5.9 Class aliases, and §4.4.4 Interface aliases, is a synonym for an expression involving other types or generic types. A type alias may itself be generic.

The Ceylon type system is much more complete than most other object oriented languages. In Ceylon, it's possible to answer questions that might at first sound almost nonsensical if you're used to languages with more traditional type systems. For example:

- What is the type of a variable that may or may not hold a value of type `Element`?
- What is the type of a parameter that accepts either an `Integer` or a `Float`?
- What is the type of a parameter that accepts values which are instances of both `Persistent` and `Printable`?
- What is the type of a function which accepts any non-null value and returns a `String`?

- What is the type of a function that accepts one or more `Strings` and returns an iterable object producing at least one `String`?
- What is the type of a sequence consisting of a `String` followed by two `Floats`?
- What is the type of a list with no elements?

The answers, as we shall see, are: `Element?`, `Integer|Float`, `Persistent&Printable`, `String(Object)`, `{String+}(String+)`, `[String,Float,Float]`, and `List<Nothing>`.

It's important that there is always a unique "best" answer to questions like these in Ceylon. The "best" answer is called the *principal type of an expression*. Every other type to which the expression is assignable is a supertype of the principal type.

Thus, every legal Ceylon expression has a unique, well-defined type, representable within the type system, without reference to how the expression is used or to what type it is assigned. This is the case even when type inference or type argument inference comes into play.

Neither this specification nor the internal implementation of the Ceylon compiler itself use any kind of "non-denotable" types. Every type mentioned here or inferred internally by the compiler has a representation within the language itself. Thus, the programmer is never exposed to confusing error messages referring to mysterious types that are not part of the syntax of the language.

3.1. Identifier naming

The Ceylon compiler enforces identifier naming conventions. Types must be named with an initial uppercase letter. Methods, attributes, parameters, and locals must be named with an initial lowercase letter or underscore. The grammar for identifiers is defined by §2.3 Identifiers and keywords.

```
TypeName: UIdentifier
```

```
MemberName: LIdentifier
```

A package or module name is a sequence of identifiers, each with an initial lowercase letter or underscore.

```
PackageName: LIdentifier
```

Ceylon defines three identifier namespaces:

- classes, interfaces, type aliases, and type parameters share a single namespace,
- functions, values, and parameters share a single namespace, and
- packages have their own dedicated namespace.

The Ceylon parser is able to unambiguously identify which namespace an identifier belongs to.

An identifier that begins with an initial lowercase letter may be *forced* into the namespace of types by prefixing the identifier `\I`. An identifier that begins with an initial uppercase letter may be forced into the namespace of methods and attributes by prefixing the identifier `\i`. A keyword may be used as an identifier by prefixing the keyword with either `\i` or `\I`. This allows interoperation with languages like Java which do not enforce these naming conventions.

3.2. Types

A *type* or *type schema* is a name (an initial uppercase identifier) and an optional list of type parameters, with a set of:

- value schemas,
- function schemas, and
- class schemas.

The value, function, and class schemas are called the *members* of the type.

Speaking formally:

- A *value schema* is a name (an initial lowercase identifier) with a type and mutability.
- A *function schema* is a name (an initial lowercase identifier) and an optional list of type parameters, with a type (often called the *return type*) and a sequence of one or more parameter lists.
- A *class schema* is a type schema with exactly one parameter list.
- A *parameter list* is a list of names (initial lowercase identifiers) with types. The *signature* of a parameter list is formed by discarding the names, leaving the list of types.

Speaking slightly less formally, we usually refer to an *attribute*, *method*, or *member class* of a type, meaning a value schema, function schema, or class schema that is a member of the type.

A function or value schema may occur outside of a type schema. If it occurs directly in a compilation unit, we often call it a *toplevel function* or *toplevel value*.

A value schema, function schema, or parameter list with a missing type or types may be defined. A value schema, function schema, or parameter list with a missing type is called *partially typed*.

Two signatures are considered identical if they have exactly the same types, at exactly the same positions, and missing types at exactly the same positions.

3.2.1. Member distinctness

Overloading is illegal in Ceylon. A type may not have:

- two attributes with the same name,
- a method and an attribute with the same name,
- two methods with the same name, or
- two member classes with the same name.

3.2.2. Subtyping

A type may be a *subtype* of another type. Subtyping obeys the following rules:

- Identity: `X` is a subtype of `X`.
- Transitivity: if `X` is a subtype of `Y` and `Y` is a subtype of `Z` then `X` is a subtype of `Z`.
- Noncircularity: if `X` is a subtype of `Y` and `Y` is a subtype of `X` then `Y` and `X` are the same type.
- Single root: all types are subtypes of the class `Anything` defined in the module `ceylon.language`.

Every interface type is a subtype of the class `Object` defined in `ceylon.language`.

If `X` is a subtype of `Y`, then:

- For each non-`variable` attribute of `Y`, `X` has an attribute with the same name, whose type is assignable to the type of the attribute of `Y`.
- For each `variable` attribute of `Y`, `X` has a `variable` attribute with the same name and the same type.
- For each method of `Y`, `X` has a method with the same name, with the same number of parameter lists, with the same signatures, and whose return type is assignable to the return type of the method of `Y`.
- For each member class of `Y`, `X` has a member class of the same name, with a parameter list with the same signature, that

is a subtype of the member class of `Y`.

Furthermore, we say that `X` is *assignable* to `Y`.

3.2.3. Union types

For any types `X` and `Y`, the *union*, or *disjunction*, `X|Y`, of the types may be formed. A union type is a supertype of both of the given types `X` and `Y`, and an instance of either type is an instance of the union type.

The union type constructor `|` is associative, so the union of three types, `X`, `Y`, and `Z`, may be written `X|Y|Z`.

```
UnionType: IntersectionType ("|" IntersectionType)*
```

If `X` and `Y` are both subtypes of a third type `Z`, then `X|Y` inherits all members of `Z`.

```
void write(String|Integer|Float printable) { ... }
```

Union types satisfy the following rules, for any types `X`, `Y`, and `Z`:

- Commutativity: `X|Y` is the same type as `Y|X`.
- Associativity: `X|(Y|Z)` is the same type as `(X|Y)|Z`.
- Simplification: if `X` is a subtype of `Y`, then `X|Y` is the same type as `Y`.
- Subtypes: `X` is a subtype of `X|Y`.
- Supertypes: if both `X` and `Y` are subtypes of `Z`, then `X|Y` is also a subtype of `Z`.

The following results follow from these rules:

- `X|Nothing` is the same type as `X` for any type `X`, and
- `X|Anything` is the same type as `Anything` for any type `X`.

Finally:

- If `X<T>` is covariant in the type parameter `T`, then `X<U>|X<V>` is a subtype of `X<U|V>` for any types `U` and `V` that satisfy the type constraints on `T`.
- If `X<T>` is contravariant in the type parameter `T`, then `X<U>|X<V>` is a subtype of `X<U&V>` for any types `U` and `V` that satisfy the type constraints on `T`.

3.2.4. Intersection types

For any types `X` and `Y`, the *intersection*, or *conjunction*, `X&Y`, of the types may be formed. An intersection type is a subtype of both of the given types `X` and `Y`, and any object which is an instance of both types is an instance of the intersection type.

The intersection type constructor `&` is associative, so the intersection of three types, `X`, `Y`, and `Z`, may be written `X&Y&Z`.

```
IntersectionType: PrimaryType ("&" PrimaryType)*
```

The intersection `X&Y` inherits all members of both `X` and `Y`.

```
void store(Persistent&Printable&Identifiable storable) { ... }
```

Intersection types satisfy the following rules, for any types `X`, `Y`, and `Z`:

- Commutativity: `X&Y` is the same type as `Y&X`.
- Associativity: `X&(Y&Z)` is the same type as `(X&Y)&Z`.

- Simplification: if `X` is a subtype of `Y`, then `X&Y` is the same type as `X`.
- Supertypes: `X` is a supertype of `X&Y`.
- Subtypes: if both `X` and `Y` are supertypes of `Z`, then `X&Y` is also a supertype of `Z`.
- Distributivity over union: `X&(Y|Z)` is the same type as `(X&Y)|(X&Z)`.

The following results follow from these rules:

- `X&Nothing` is the same type as `Nothing` for any type `X`, and
- `X&Anything` is the same type as `X` for any type `X`.

Finally:

- If `X<T>` is covariant in the type parameter `T`, then `X<U>&X<V>` is a supertype of `X<U&V>` for any types `U` and `V` that satisfy the type constraints on `T`.
- If `X<T>` is contravariant in the type parameter `T`, then `X<U>&X<V>` is a supertype of `X<U|V>` for any types `U` and `V` that satisfy the type constraints on `T`.

3.2.5. The bottom type

The special type `Nothing`, sometimes called the *bottom type*, represents:

- the intersection of all types, or, equivalently
- the empty set.

`Nothing` is assignable to all other types, but has no instances.

The type schema for `Nothing` is empty, that is, it is considered to have no members.

`Nothing` is considered to belong to the module `ceylon.language`. However, it cannot be defined within the language.

Because of the restrictions imposed by Ceylon's mixin inheritance model:

- If `X` and `Y` are classes, and `X` is not a subclass of `Y`, and `Y` is not a subclass of `X`, then the intersection type `X&Y` is equivalent to `Nothing`.
- If `X` is an interface, the intersection type `X&Null` is equivalent to `Nothing`.
- If `X` is an interface, and `Y` is a `final` class, and `Y` is not a subtype of `X`, then the intersection type `X&Y` is equivalent to `Nothing`.
- If `X<T>` is invariant in its type parameter `T`, and the distinct types `A` and `B` do not involve type parameters, then `X<A>&X<B>` is equivalent to `Nothing`.

TODO: Should the name of this type be a keyword, perhaps `nothing`, to emphasize that it is defined primitively?

3.2.6. Principal typing

An expression, as defined in Chapter 6, *Expressions*, occurring at a certain location, may be *assignable* to a type. In this case, every evaluation of the expression at runtime produces an instance of a class that is a subtype of the type, or results in a thrown exception, as defined in Chapter 8, *Execution*.

Given an expression occurring at a certain location, a type `T` is the *principal type* of the expression if, given any type `U` to which the expression is assignable, `T` is a subtype of `U`. Thus, the principal type is the "most precise" type for the expression. The type system guarantees that every expression has a principal type. Thus, we refer uniquely to *the type of an expression*, meaning its principal type at the location at which it occurs.

3.2.7. Type expressions

Function and value declarations usually declare a type, by specifying a *type expression*.

```
Type: UnionType | EntryType
```

Type expressions are formed by combining types using union, intersection, and type abbreviations.

Type expressions support grouping using angle brackets:

```
GroupedType: "<" Type ">"
```

Applied types are identified by the name of the type (a class, interface, type alias, or type parameter), together with a list of type arguments if the type declaration is generic.

```
TypeNameWithArguments: TypeName TypeArguments?
```

Type names are resolved to type declarations according to §5.1.7 Unqualified reference resolution and §5.1.8 Qualified reference resolution.

The type argument list, if any, must conform, as defined by §3.6.1 Type arguments and type constraints, to the type parameter list of the realization of the type declaration, as defined by §3.7.6 Realizations.

Note: this is too heavy-handed. There is no reason to enforce types constraint in any place other than generic class instantiations, generic function invocations, `extends`, and `satisfies`. However, this restriction makes interoperation with Java generics more straightforward.

If the type is a class, interface, or type alias nested inside a containing class or interface, the type must be fully qualified by its containing types, except when used inside the body of a containing type.

```
QualifiedType: TypeNameWithArguments ("." TypeNameWithArguments)*
```

If a type declaration is generic, a type argument list must be specified. If a type declaration is not generic, no type argument list may be specified.

```
BufferedReader.Buffer
```

```
Entry<Integer,Element>
```

Note: the name of a type may not be qualified by its package name. Alias imports, as defined in §4.2.3 Alias imports may be used to disambiguate type names.

3.2.8. Type abbreviations

Certain important types may be written using an abbreviated syntax.

```
PrimaryType: AtomicType | OptionalType | SequenceType | CallableType
```

```
AtomicType: QualifiedType | EmptyType | TupleType | IterableType | GroupedType
```

First, there are postfix-style abbreviations for *optional types*, *sequence types*, and *callable types*.

```
OptionalType: PrimaryType "?"
```

```
SequenceType: PrimaryType "[" "]"
```

```
CallableType: PrimaryType "(" TypeList? ")"
```

- `X?` means `Null|X` for any type `X`,
- `X[]` means `Sequential<X>` for any type `X`, and

- `X(Y,Z)` means `Callable<X,[Y,Z]>` where `Y,Z` is a list of types of any length.

More precisely, the type meant by a callable type abbreviation is `Callable<X,T>` where `X` is the type outside the parentheses in the the callable type abbreviation, and `T` is the tuple type formed by the types listed inside the parentheses.

Next, abbreviations for *iterable types* are written using braces.

```
IterableType: "{" UnionType ("*"|"+") "}"
```

- `{X*}` means `Iterable<X,Null>` for any type `X`, and
- `{X+}` means `Iterable<X,Nothing>` for any type `X`.

Next, abbreviations for *sequence types* and *tuple types* may be written using brackets.

```
EmptyType: "[" "]"
```

```
TupleType: "[" TypeList "]"
```

```
TypeList: (EntryType ",")* UnionType ("*"|"+")?
```

- `[X*]` means `Sequential<X>` for any type `X`,
- `[]` means `Empty`,
- `[X+]` means `Sequence<X>` for any type `X`, and
- `[X,Y]` means `Tuple<X|Y,X,Tuple<Y,Y,[]>>` where `X,Y` is a list of types of any length.

More precisely:

- A tuple type abbreviation of form `[X, ... ]` means the type `Tuple<X|Y,X,T>` where `T` is the type meant by the type abbreviation formed by removing the first type `X` from the list of types in the original tuple type abbreviation, and `T` has the principal instantiation `Y[]`, as defined in §3.7 Principal instantiations and polymorphism.

Finally, an *entry type* may be abbreviated using an arrow.

```
EntryType: UnionType "->" UnionType
```

- `X->Y` means `Entry<X,Y>`, for any types `X`, `Y`.

Note: the abbreviations `T[]` and `[T*]` are synonyms. The syntax `T[]` is supported for reasons of nostalgia.

Abbreviations may be combined:

```
String?[] words = { "hello", "world", null };
String? firstWord = words[0];

String->[Integer,Integer] onetwo = "onetwo"->[1, 2];

[Float+](Float x, Float[] xs) add = (Float x, Float[] xs) => [x, *xs];
```

When a type appears in a value expression, these abbreviations cannot be used (they cannot be disambiguated from operator expressions).

3.2.9. Type inference

Certain declarations which usually require an explicit type may omit the type, forcing the compiler to infer it, by specifying the keyword `value`, as defined in §4.8.4 Value type inference, or `function`, as defined in §4.7.4 Function return type inference, where the type usually appears.

```
value names = people*.name;
```

```
function parse(String text) => text.split(" .!?,:;()\n\f\r\t".contains);
```

Type inference is only allowed for declarations which are referred to only by statements and declarations that occur within the lexical scope of the declaration, as specified by §5.1.6 Type inference and block structure. A `value` or `function` declaration may not:

- be annotated `shared`, as defined in §5.1.3 Visibility,
- occur as a toplevel declaration in a compilation unit, as defined in §4.1.1 Toplevel and nested declarations, or
- be referred to by statements or declarations that occur earlier in the body containing of the declaration, as defined in §5.1 Block structure and references.

Nor may a parameter or forward-declared value, as defined in §4.8.5 Forward declaration of values, or of a forward-declared function, as defined in §4.7.5 Forward declaration of functions, have an inferred type.

These restrictions allow the compiler to infer undeclared types in a single pass of the code.

Note: in future releases of the language, the inferred type will be context-dependent, that is, in program elements immediately following an assignment or specification, the inferred type will be the type just assigned. When conditional execution results in definite assignment, the inferred type will be the union of the conditionally assigned types. This will allow us to to relax the restriction that forward-declared functions and values can't have their type inferred. For example:

```
value one;
if (float) {
    one = 1.0;
    Float float = one;
}
else {
    one = 1;
    Integer int = one;
}
Float|Integer num = one;
```

An inferred type never involves an anonymous class, as defined in §4.5.7 Anonymous classes. When an inferred type would involve an anonymous class type, the anonymous class is replaced by the intersection of the class type it extends with all interface types it satisfies.

TODO: properly define how expressions with no type occurring in a `dynamic` block affect type inference.

3.2.10. Type alias elimination

A *type alias* is a synonym for another type. A generic type alias is a type constructor that produces a type alias, given a list of type arguments.

Every type alias must be reducible to an equivalent type that does not involve any type aliases by recursive replacement of type aliases with the types they alias. Thus, circular type alias definitions, as in the following example, are illegal:

```
alias X => List<Y>;
alias Y => List<X>;
```

Replacement of type aliases with the types they alias occurs at compile time, so type aliases are not reified types, as specified in §8.1.2 Type argument reification.

3.3. Inheritance

Inheritance is a static relationship between classes, interfaces, and type parameters:

- a class may *extend* another class, as defined by §4.5.4 Class inheritance,
- a class may *satisfy* one or more interfaces, as defined by §4.5.4 Class inheritance,
- an interface may *satisfy* one or more other interfaces, as defined by §4.4.2 Interface inheritance, or

- a type parameter may *satisfy* a class and/or one or more interfaces or type parameters, as defined by §3.5.3 Generic type constraints.

If a type declaration extends or satisfies a type, we say it *inherits* the type.

Inheritance relationships may not produce cycles, since that would violate the noncircularity rule for subtyping. Thus, a class, interface, or type parameter may not, directly or indirectly, inherit itself.

Note: when a type declaration specifies a relationship to other types, Ceylon visually distinguishes between a list of types which conceptually represents a combination of (intersection of) the types, and a list of types which represents a choice between (union of) the types. For example, when a class `C` satisfies multiple interfaces, they are written as `X&Y&Z`. On the other hand, the cases of an enumerated class `E` are written as `X|Y|Z`. This syntax emphasizes that `C` is also a subtype of the intersection type `X&Y&Z`, and that `E` may be narrowed to the union type `X|Y|Z` using a `switch` statement or the `of` operator.

3.3.1. Inheritance and subtyping

Inheritance relationships between classes, interfaces, and type parameters result in subtyping relationships between types.

- If a type `X` inherits a type `Y`, then `X` is a subtype of `Y`.
- If a generic type `X` inherits a type `Y` that might involve the type parameters of `X`, then for any instantiation `U` of `X` we can construct a type `V` by, for every type parameter `T` of `X`, substituting the corresponding type argument of `T` given in `U` everywhere `T` occurs in `Y`, and then `U` is a subtype of `V`.

3.3.2. Extension

A class may extend another class, in which case the first class is a subtype of the second class and inherits its members.

```
ExtendedType: "extends" ("super" ".")? TypeNameWithArguments PositionalArguments
```

The `extends` clause must specify exactly one superclass.

- If the superclass is a parameterized type, the `extends` clause must also explicitly specify type arguments.
- The `extends` clause must specify arguments for the initializer parameters of the superclass.

The type arguments may *not* be inferred from the initializer arguments.

```
extends Person(name, org)
```

A member class annotated `actual` may use the qualifier `super` in the `extends` clause to refer to the member class it refines. When the qualifier `super` appears, the following class name refers to a member class of the superclass of the class that contains the member class annotated `actual`.

```
extends super.Buffer()
```

The root class `Anything` defined in `ceylon.language` does not have a superclass.

3.3.3. Satisfaction

The `satisfies` clause does double duty. It's used to specify that a class or interface is a direct subtype of one or more interfaces, and to specify upper bound type constraints applying to a type parameter.

Note: for this reason the keyword is not named "`implements`". It can't reasonably be said that a type parameter "implements" its upper bounds. Nor can it be reasonably said that an interface "implements" its super-interfaces.

- A class or interface may satisfy one or more interfaces, in which case the class or interface is a subtype of the satisfied interfaces, and inherits their members.
- A type parameter may satisfy one or more interfaces, optionally, a class, and optionally, another type parameter. In this case, the satisfied types are interpreted as upper bound type constraints on arguments to the type parameter.

Note: currently, a type parameter upper bound may not be specified in combination with other upper bounds. This restriction will likely be removed in future.

```
SatisfiedTypes: "satisfies" PrimaryType ("&" PrimaryType)*
```

The `satisfies` clause may specify multiple types. If a satisfied type is a parameterized type, the `satisfies` clause must specify type arguments.

```
satisfies Sequence<Element> & Collection<Element>
```

3.4. Case enumeration and coverage

Coverage is a static relationship between classes, interfaces, and type parameters, produced through the use of *case enumeration*:

- An `abstract` class or interface may be an *enumerated type*, with an enumerated list of disjoint subtypes called *cases*, as defined by §4.5.8 Classes with enumerated cases and §4.4.3 Interfaces with enumerated cases.
- A type parameter may have an *enumerated bound*, with an enumerated list possible type arguments, as defined by §3.5.3 Generic type constraints.
- An `abstract` class or interface may have a *self type*, a type parameter representing the concrete type of an instance.

3.4.1. Coverage

Coverage is a strictly weaker relationship than assignability:

- If a type is a subtype of a second type, then the second type covers the first type.
- If a type has a self type, then its self type covers the type.
- If a type `X` enumerates its cases `X1`, `X2`, etc, then the union `X1|X2|...` of its cases covers the type.
- If a generic type `X` enumerates its cases, `X1`, `X2`, etc, which might involve the type parameters of `X`, then for any instantiation `U` of `X`, and for each case `Xi`, we can construct a type `Ui` by, for every type parameter `T` of `X`, substituting the corresponding type argument of `T` given in `U` everywhere `T` occurs in `Xi`, and then the union type `U1|U2|...` of all the resulting types `Ui` covers `Y`.
- If a type `X` covers two types `A` and `B`, then `X` also covers their union `A|B`.
- Coverage is transitive. If `X` covers `Y` and `Y` covers `Z`, then `X` covers `Z`.

It follows that coverage obeys the identity property of assignability: a type covers itself. However, coverage does not obey the noncircularity property of assignability. It is possible to have distinct types `A` and `B` where `A` covers `B` and `B` covers `A`.

Case enumeration allows safe use of a type in a `switch` statement, or as the subject of the `of` operator. The compiler is able to statically validate that the `switch` contains an exhaustive list of all cases of the type, by checking that the union of cases enumerated in the `switch` covers the type, or that the second operand of `of` covers the type.

Note: however, a type is not considered automatically assignable to the union of its cases, or to its self type. Instead, the type must be explicitly narrowed to the union of its cases, or to its self type, using either the `of` operator or the `switch` construct. This narrowing type conversion can be statically checked—if `X` covers `Y` then `Y of X` is guaranteed to succeed at runtime. Unfortunately, and quite unintuitively, the compiler is not able to analyse coverage implicitly at the same time as assignability, because that results in undecidability!

3.4.2. Cases

The `of` clause does triple duty. It's used to define self types and type families, enumerated types, and enumerated type constraints. The `of` clause may specify multiple types, called *cases*.

```
CaseTypes: "of" CaseType ("|" CaseType)*
```

```
CaseType: MemberName | PrimaryType
```

If an interface or `abstract` class with an `of` clause has exactly one case, and it is a type parameter of the interface or `abstract` class, or of the immediately containing type, if any, then that type parameter is a *self type* of the interface or `abstract` class, and:

- the self type parameter covers the declared type within the body of the declaration,
- the type argument to the self type parameter in an instantiation of the declared type covers the instantiation, and
- every type which extends or satisfies an instantiation of the declared type must also be covered by the type argument to the self type parameter in the instantiation.

```
shared abstract class Comparable<Other>() of Other
        given Other satisfies Comparable<Other> {

    shared formal Integer compare(Other that);

    shared Integer reverseCompare(Other that) => that.compare(this) of Other;

}
```

```
Comparable<Item> comp = ... ;
Item item = comp of Item;
```

Otherwise, an interface or `abstract` class with an `of` clause may have multiple cases, but each case must be either:

- a subtype of the interface or `abstract` class, or
- a value reference to a toplevel anonymous class that is a subtype of the interface or `abstract` class.

Then the interface or `abstract` class is an *enumerated type*, and every subtype of the interface or `abstract` class must be a subtype of exactly one of the enumerated subtypes. A class or interface may not be a subtype of more than one case of an enumerated type.

```
of larger | smaller | equal
```

```
of Root<Element> | Leaf<Element> | Branch<Element>
```

A type parameter with an `of` clause may specify multiple cases, as defined in §3.5.3 Generic type constraints.

3.4.3. Generic enumerated types

If a generic enumerated type `X` has a case type `C`, then `C` must directly extend or satisfy an instantiation `Y` of `X`, and for each type parameter `T` of `X` and corresponding argument `A` of `T` given in `Y`, either:

- `X` is covariant in `T` and `A` is exactly `Nothing`,
- `X` is contravariant in `T` and `A` is exactly the intersection of all upper bounds on `T`, or `Anything` if `T` has no upper bounds, or
- `C` is an instantiation of a generic type `G` and `A` is exactly `S` for some type parameter `S` of `G`, and `S` must have the same variance as `T`.

For example, the following covariant enumerated type is legal:

```
interface List<out Element>
        of Cons<Element> | nil { ... }

class Cons<out Element>(Element element)
        satisfies List<Element> { ... }

object nil
        satisfies List<Nothing> { ... }
```

As is the following contravariant enumerated type:

```
interface Consumer<in Event>
        of Logger | Handler<Event>
        given Event satisfies AbstractEvent { ... }

interface Logger
        satisfies Consumer<AbstractEvent> { ... }

interface Handler<in Event>
        satisfies Consumer<AbstractEvent>
        given Event satisfies AbstractEvent { ... }
```

But the following enumerated type is not legal, since it is possible to choose a legal argument `T` of the type parameter `Type` of `Expression`, such that the case types `StringExpression` and `NumericExpression` aren't subtypes of the instantiation `Expression<T>`:

```
interface Expression<out Type>
        of Function<Type> | String | Number { ... }

interface Function<out Type>
        satisfies Expression<Type> { ... }

interface String
        satisfies Expression<String> { ... }

interface Number
        satisfies Expression<Integer|Float> { ... }
```

Note: these rules could be relaxed to allow the definition of generic enumerated types where the list of cases of an instantiation of a generic type depends upon the given type arguments (a "generalized" algebraic type).

3.4.4. Disjoint types

Two types are said to be *disjoint* if it is impossible to have a value that is an instance of both types. If `X` and `Y` are disjoint, then their intersection `X&Y` is the bottom type `Nothing`.

Two types `X` and `Y` are disjoint if either:

- `X` and `Y` are both classes and `X` is not a subclass of `Y` and `Y` is not a subclass of `X`,
- `X` is the class `Null` and `Y` is an interface,
- `X` is an anonymous class or an instantiation of a `final` class and `Y` is an instantiation of a class of interface, and `X` does not inherit `Y`,
- `X` is an anonymous class or a `final` class with no type parameters and `Y` is a type in which no type parameter reference occurs, and `X` is not a suptype of `Y`,
- `X` is a type parameter and `Y` and the intersection of the upper bounds of `X` are disjoint,
- `X` is an union type `A|B` and both `Y` and `A` are disjoint and `Y` and `B` are disjoint,
- `X` is an intersection type `A&B` and either `Y` and `A` are disjoint or `Y` and `B` are disjoint, or
- `X` and `Y` inherit disjoint instantiations of a generic type `Z`, that is, two instantiations of `Z` that have the intersection `Nothing`, as defined below, in §3.7.2 Principal instantiation inheritance.

3.5. Generic type parameters

Function, class, and interface schemas may be parameterized by one or more generic type parameters. A parameterized type schema defines a type constructor, a function that produces a type given a tuple of compatible type arguments. A parameterized class or function schema defines a function that produces the signature of an invokable operation given a tuple of compatible type arguments.

```
TypeParameters: "<" (TypeParameter ",")* TypeParameter ">"
```

A declaration with type parameters is called *generic* or *parameterized*.

- A type schema with no type parameters defines exactly one type. A parameterized type schema defines a template for producing types: one type for each possible combination of type arguments that satisfy the type constraints specified by the type. The types of members of the this type are determined by replacing every appearance of each type parameter in the schema of the parameterized type definition with its type argument.
- A function schema with no type parameters defines exactly one operation per type. A parameterized function declaration defines a template for producing overloaded operations: one operation for each possible combination of type arguments that satisfy the type constraints specified by the method declaration.
- A class schema with no type parameters defines exactly one instantiation operation. A parameterized class schema defines a template for producing overloaded instantiation operations: one instantiation operation for each possible combination of type arguments that satisfy the type constraints specified by the class declaration. The type of the object produced by an instantiation operation is determined by substituting the same combination of type arguments for the type parameters of the parameterized class schema.

Note: by convention, type parameter names should be constructed from meaningful words. The use of single-letter type parameter names is discouraged. The name of a type parameter should be chosen so that declarations within the body of the parameterized declaration read naturally. For example, `class Entry<Key,Item>` is reasonable, since `Key key` and `Item item` read naturally within the body of the `Entry` class. The following identifier names usually refer to a type parameter: `Element`, `Other`, `This`, `Value`, `Key`, `Item`, `Argument`, `Args` and `Result`. Avoid, where reasonable, using these names for interfaces and classes.

3.5.1. Type parameters and variance

A *type parameter* allows a declaration to be abstracted over a constrained set of types.

```
TypeParameter: Variance? TypeName ("=" Type)
```

Every type parameter has a name and a *variance*.

```
Variance: "out" | "in"
```

- A *covariant* type parameter is indicated using the keyword `out`.
- A *contravariant* type parameter is indicated using the keyword `in`.
- By default, a type parameter is *invariant*.

A type parameter may, optionally, have a *default type argument*. A type parameter with a default type argument must occur after all type parameters without default type arguments in the type parameter list. The default type argument must satisfy the constraints on the type parameter.

A default type argument may not involve the parameter for which it is the default argument, nor any type parameter of the declaration that occurs later in the list of type parameters.

Within the body of the schema it parameterizes, a type parameter is itself a type. The type parameter is a subtype of every upper bound of the type parameter. However, a class or interface may not extend or satisfy a type parameter.

```
<Key, out Item>
```

```
<in Message>
```

```
<out Element=Object>
```

```
<in Left, in Right, out Result>
```

3.5.2. Variance validation

A covariant type parameter may only appear in *covariant positions* of the parameterized schema. A contravariant type

parameter may only appear in *contravariant positions* of the parameterized schema. An invariant type parameter may appear in any position.

Furthermore, a type with a contravariant type parameter may only appear in a covariant position in an extended type, satisfied type, case type, or upper bound type constraint.

Note: this restriction exists to eliminate certain undecidable cases described in the paper Taming Wildcards in Java's Type System, by Tate et al.

To determine if a type expression occurs in a covariant or contravariant position, we first consider how the type occurs syntactically.

For a generic function we examine the return type of the function, which is a covariant position.

For a generic type schema we examine each `shared` member, along with extended/satisfied types and case types.

Note: since the visibility rules are purely lexical in nature, it is legal for a member expression occurring in the body of a class or interface to have a receiver expression other that is not a self-reference, as defined in §6.3 Self references and the current package reference, and refer to an un-`shared` member of the class or interface. In this special case, the member is treated as if it were `shared` for the purposes of the following variance validation rules.

- An extended type, satisfied type, or case type of the type schema itself is a covariant position.

In a `shared` method declaration of the parameterized type schema:

- The return type of the method is a covariant position.
- Any parameter type of the method is a contravariant position.
- Any upper bound of a type parameter of the method is a contravariant position.

In a `shared` attribute declaration that is not variable:

- The type of the attribute is a covariant position.

In a `shared` reference declaration that is variable:

- The type of the attribute is an invariant position.

In a `shared` nested class declaration of the parameterized type schema:

- Any initializer parameter type of the class is a contravariant position.
- Any upper bound of a type parameter of the class is a contravariant position.
- An extended type, satisfied type, or case type of the nested class is a covariant position.
- Every covariant position of the nested class schema is a covariant position of the containing type schema. Every contravariant position of the nested class schema is a contravariant position of the containing type schema.

In a `shared` nested interface declaration of the parameterized type schema:

- An extended type, satisfied type, or case type of the nested interface is a covariant position.
- Every covariant position of the nested interface schema is a covariant position of the containing type schema. Every contravariant position of the nested interface schema is a contravariant position of the containing type schema.

For parameters of callable parameters, we first determine if the callable parameter itself is covariant or contravariant:

- A callable parameter of a method or nested class is contravariant.
- A callable parameter of a covariant parameter is contravariant.
- A callable parameter of a contravariant parameter is covariant.

Then:

- The return type of a covariant callable parameter is a covariant position.
- The return type of a contravariant callable parameter is a contravariant position.
- The type of a parameter of a covariant callable parameter is a contravariant position.
- The type of a parameter of a contravariant callable parameter is a covariant position.

Finally, to determine if a type parameter that occurs as a type argument occurs in a covariant or contravariant position, we must consider the declared variance of the corresponding type parameter:

- A type argument of a covariant type parameter of a type in a covariant position is a covariant position.
- A type argument of a contravariant type parameter of a type in a covariant position is a contravariant position.
- A type argument of a covariant type parameter of a type in a contravariant position is a contravariant position.
- A type argument of a contravariant type parameter of a type in a contravariant position is a covariant position.
- A type argument of an invariant type parameter of a type in any position is an invariant position.
- A type argument of any type parameter of a type in an invariant position is an invariant position.

3.5.3. Generic type constraints

A parameterized method, class, or interface declaration may declare constraints upon ordinary type parameters using the `given` clause.

```
TypeConstraints: TypeConstraint+
```

There may be at most one `given` clause per type parameter.

```
TypeConstraint: "given" TypeName TypeConstraintInheritance
```

```
TypeConstraintInheritance: CaseTypes? SatisfiedTypes?
```

Note that the syntax for a type constraint is essentially the same syntax used for other type declarations such as class and interface declarations.

There are two different kinds of type constraint:

- An *upper bound*, `given X satisfies T`, specifies that the type parameter `X` is a subtype of a given type `T`.
- An *enumerated bound*, `given X of T|U|V` specifies that the type parameter `X` represents one of the enumerated types.

The types listed in an enumerated bound must be mutually disjoint, and each type must be a class or interface type.

TODO: Should we allow unions in upper bounds? Should we allow intersections in enumerated bounds?

A single `given` clause may specify multiple constraints on a certain type parameter. In particular, it may specify multiple upper bounds together with an enumerated bound. If multiple upper bounds are specified, at most one upper bound may be a class, and at most one upper bound may be a type parameter.

Note: in Ceylon 1.0, a type parameter with multiple upper bounds may not have an upper bound which is another type parameter.

```
given Value satisfies Ordinal & Comparable<Value>
```

```
given Quantities satisfies Correspondence<Key,Decimal>
```

```
given Argument of String | Integer | Float
```

A type parameter is a subtype of its upper bounds.

```
class Holder<Value>(shared Value value)
        extends Object
        given Value satisfies Object {
    shared actual Boolean equals(Object that) {
        if (is Holder<Value> that) {
            return value==that.value;
        }
        else {
            return false;
        }
    }
    shared actual Integer hash =>  value.hash;
}
```

Every type parameter has an implicit upper bound of type `Anything`.

An enumerated bound allows the use of an exhaustive `switch` with expressions of the parameter type.

```
Characters uppercase<Characters>(Characters chars)
       given Characters of String | Range<Character> {
    switch (Characters)
    case (satisfies String) {
        return chars.uppercased;
    }
    case (satisfies Range<Character>) {
        return chars.first.uppercased..chars.last.uppercased;
    }
}
```

TODO: Do we need lower bound type constraints? The syntax would be:

```
given T abstracts One|Two
```

With union types they don't appear to be anywhere near as useful. However, perhaps they are useful when combined with contravariant types. (A lower bound on a parameter which occurs as the argument of a contravariant type is more like an upper bound).

Note: since we have reified types, it would be possible to support a type constraint that allows instantiation of the type parameter.

```
given T(Object arg)
```

The problem with this is that then inferring `T` is fragile. And if we don't let it be inferred, we may as well pass `T` as an ordinary parameter. So Ceylon, unlike C#, doesn't support this.

3.6. Generic type arguments

A list of *type arguments* produces a new type schema from a parameterized type schema, or a new function schema from a from a parameterized function schema. In the case of a type schema, this new schema is the schema of an applied type.

A type argument list is a list of types.

```
TypeArguments: "<" (Type ",")* Type ">"
```

A type argument may itself be an applied type, or type parameter, or may involve unions and intersections.

```
<Key, List<Item>>
```

```
<String, Person?>
```

```
<String[], [{Object*}]>
```

Type arguments are assigned to type parameters according to the positions they occur in the list.

Given the schema of a generic declaration, we form the new schema by *type argument substitution*. Each type argument is

substituted for every appearance of the corresponding type parameter in the schema of the generic declaration, including:

- attribute types,
- method return types,
- method parameter types,
- initializer parameter types, and
- type arguments of extended classes and satisfied interfaces.

3.6.1. Type arguments and type constraints

A generic type constraint affects the type arguments that can be assigned to a type parameter:

- A type argument to a type parameter `T` with an upper bound must be a type which is a subtype of all upper bounds of `T`.
- A type argument to a type parameter `T` with an enumerated type bound must be a subtype of one of the enumerated types of `T`, or it must be a type parameter `A` with an enumerated type bound where every enumerated type of `A` is also an enumerated type of `T`.

A type argument list *conforms* to a type parameter list if, for every type parameter in the list, either:

- there is a type argument that satisfies the constraints of the type parameter, or
- there is no explicit type argument but the type parameter has a default type argument, in which case the type argument is defaulted by substituting the arguments of all type parameters that occur earlier in the list of type parameters of the declaration into this default type argument.

There must be at least as many type parameters as type arguments. There must be at least as many type arguments as type parameters without default values.

3.6.2. Applied types and and variance

If a type argument list conforms to a type parameter list, the combination of the parameterized type together with the type argument list is itself a type, called an *applied type*. We also call the applied type an *instantiation* of the generic type.

For a generic type `X`, the instantiations `Y` and `Z` of `X` represent the same type if and only if for every `A` in the list of type arguments specified in `Y` and corresponding `B` in the list of type arguments specified in `Z`, `A` is exactly the same type as `B`.

For a generic type `X`, and instantiations `Y` and `Z` of `X`, `Y` is a subtype of `Z` if and only if, for every type parameter `T` of `X`, and corresponding arguments `A` specified in `Y` and `B` specified in `Z`:

- `T` is a covariant type parameter, and `A` is a subtype of `B`, or
- `T` is a contravariant type parameter, and `B` is a subtype of `A`, or
- `T` is an invariant type parameter (neither covariant nor contravariant), and `A` and `B` are exactly the same type.

Note that if `T` is an invariant type parameter of `X<T>`, then a type `Z` is a subtype of `X<A>` if and only if `Z` has the principal instantiation `X<A>`.

3.6.3. Type argument inference

When a direct invocation expression, as defined by §6.6 Invocation expressions, for a generic function or a direct instantiation expression for a generic class does not explicitly specify type arguments, the type arguments are inferred from the argument expression types. The types of the argument expressions and the declared types of the corresponding parameters determine an *inferred lower bound* or *inferred upper bound* for each type parameter.

If a list of argument expressions has types `A1,A2,...` and the corresponding list of parameters has declared types `P1,P2,...`, the inferred lower bound for a type parameter `T` of the generic declaration is the conjunction of:

- all inferred lower bounds `Ai` on `Pi` for `T`.

The inferred upper bound for a type parameter `T` of the generic declaration is the conjunction of:

- all upper bounds `Xi` explicitly declared by a type constraint on `T` of form `given T satisfies Xi`, if any, with
- all inferred upper bounds `Ai` on `Pi` for `T`.

TODO: What should we do about upper bound constraints that involve other type parameters? Currently the typechecker simply ignores any upper bound that involves any type parameter.

Given types `A` and `P`, we determine the *inferred lower bound* `A` on `P` for `T` according to the nature of `A` and `P`:

- If `P` is exactly `T`, the inferred lower bound `A` on `P` for `T` is `T abstracts A`.
- If `P` is a union type `Q|R`, the lower bound `A` on `P` for `T` is the disjunction of the lower bound `A` on `Q` for `T` with the lower bound `A` on `R` for `T`. *Note: this case is special.*
- If `P` is an intersection type `Q&R`, the lower bound `A` on `P` for `T` is the conjunction of the lower bound `A` on `Q` for `T` with the lower bound `A` on `R` for `T`.
- If `A` is a union type `B|C`, the lower bound `A` on `P` for `T` is the conjunction of the lower bound `B` on `P` for `T` with the lower bound `C` on `P` for `T`.
- If `A` is an intersection type `B&C`, the lower bound `A` on `P` for `T` is the disjunction of the lower bound `B` on `P` for `T` with the lower bound `C` on `P` for `T`.
- If `P` is an applied type `Q<P1,P2,...>` of a parameterized type `Q`, and `A` is a subtype of an applied type `Q<A1,A2,..>`, the lower bound `A` on `P` for `T` is the conjunction of all lower bounds `Ai` on `Pi` for `T`.
- Otherwise, if `A` is not a union or intersection, and if `P` is neither an applied type, a union, or an intersection, nor exactly `T`, the lower bound `A` on `P` for `T` is *null*.

Where:

- the conjunction of a lower bound `T abstracts A` with a lower bound `T abstracts B` is the lower bound `T abstracts A|B`,
- the disjunction of a lower bound `T abstracts A` with a lower bound `T abstracts B` is the lower bound `T abstracts A&B`,
- the conjunction or disjunction of a lower bound `T abstracts A` with a null lower bound is `T abstracts A`, and
- the conjunction or disjunction of two null lower bounds is null.

Given types `A` and `P`, we determine the *inferred upper bound* `A` on `P` for `T` according to the nature of `A` and `P`:

- If `P` is exactly `T`, the inferred upper bound `A` on `P` for `T` is `T satisfies A`.
- If `P` is a union type `Q|R`, the upper bound `A` on `P` for `T` is the disjunction of the upper bound `A` on `Q` for `T` with the upper bound `A` on `R` for `T`. *Note: this case is special.*
- If `P` is an intersection type `Q&R`, the upper bound `A` on `P` for `T` is the conjunction of the upper bound `A` on `Q` for `T` with the upper bound `A` on `R` for `T`.
- If `A` is a union type `B|C`, the upper bound `A` on `P` for `T` is the disjunction of the upper bound `B` on `P` for `T` with the upper bound `C` on `P` for `T`.
- If `A` is an intersection type `B&C`, the upper bound `A` on `P` for `T` is the conjunction of the upper bound `B` on `P` for `T` with the upper bound `C` on `P` for `T`.
- If `P` is an applied type `Q<P1,P2,...>` of a parameterized type `Q`, and `A` is a subtype of an applied type `Q<A1,A2,..>`, the upper bound `A` on `P` for `T` is the conjunction of all upper bounds `Ai` on `Pi` for `T`.

- Otherwise, if `A` is not a union or intersection, and if `P` is neither an applied type, a union, or an intersection, nor exactly `T`, the upper bound `A` on `P` for `T` is *null*.

Where:

- the conjunction of an upper bound `T satisfies A` with an upper bound `T satisfies B` is the upper bound `T satisfies A&B`,
- the disjunction of an upper bound `T satisfies A` with an upper bound `T satisfies B` is the upper bound `T satisfies A|B`,
- the conjunction or disjunction of an upper bound `T satisfies A` with a null upper bound is `T satisfies A`, and
- the conjunction or disjunction of two null upper bounds is null.

The inferred type argument to a covariant or invariant type parameter `T` of the generic declaration is:

- `Nothing`, if the inferred lower bound for `T` is null, or, otherwise,
- the type `A`, where the inferred lower bound for `T` is `T abstracts A`.

The inferred type argument to a contravariant type parameter `T` of the generic declaration is:

- `Anything`, if the inferred upper bound for `T` is null, or, otherwise,
- the type `A`, where the inferred upper bound for `T` is `T satisfies A`.

An argument expression with no type occurring in a `dynamic` block, as defined in §5.3.12 Dynamic blocks, may cause type argument inference to fail. When combining bounds using union, any constituent bound with no type results in a bound with no type. When combining bounds using intersection, any constituent bound with no type is eliminated. If the resulting inferred upper or lower bound has no type, type argument inference is impossible for the type argument, and type arguments must be specified explicitly.

If the inferred type argument does not satisfy the generic type constraints on `T`, a compilation error results.

Consider the following invocation:

```
[Element+] prepend<Element>(Element head, Element[] sequence) { ... }
value result = prepend(null, {"hello", "world"});
```

The inferred type of `Element` is the union type `String?`.

Now consider:

```
class Bag<out Element>(Element* elements) {
    shared Bag<ExtraElement> with<ExtraElement>(ExtraElement* elements)
            given ExtraElement abstracts Element { ... }
}
Bag<String> bag = Bag("hello", "world");
value biggerBag = bag.with(1, 2, 5.0);
```

The inferred type of `ExtraElement` is the union type `Integer|Float|String`.

Finally consider:

```
interface Delegate<in Value> { ... }
class Consumer<in Value>(Delegate<Value>* delegates) { ... }
Delegate<String> delegate1 = ... ;
Delegate<Object> delegate2 = ... ;
value consumer = Consumer(delegate1, delegate2);
```

The inferred type of `Value` is `Consumer<String>`.

TODO: What about upper bounds in which the type parameter itself appears (the infamous self-type problem with `Comparable` and `Numeric`) or in which another type parameter appears?

3.7. Principal instantiations and polymorphism

Inheritance interacts with type parameterization to produce subtyping relationships between instantiations of generic types. The notion of an *inherited instantiation* and the notion of a *principal instantation* help us reason about these relationships.

Warning: this section is not for the faint of heart. Feel free to skip to Chapter 4, Declarations, unless you're really, really interested in precisely how the compiler reasons about inheritance of generic types.

3.7.1. Inherited instantiations

For a generic type `Y`, inheritance produces subtypes with *inherited instantiations* of the generic type.

- If a type `X` directly extends or satisfies an instantiation `V` of `Y`, then `X` has the inherited instantiation `V` of `Y`.
- If a generic type `X` extends or satisfies an instantiation `V` of `Y`, that may involve the type parameters of `X`, then for any instantiation `U` of `X`, we can construct an instantiation `W` of `Y` by, for every type parameter `T` of `X`, substituting the type argument of `T` given in `U` everywhere `T` occurs in `V`, and then `U` has the inherited instantiation `W` of `Y`.
- If a type `X` is a subtype of a type `Y`, and `Y` has an inherited instantiation `W` of a generic type `Z`, then `X` also has this inherited instantiation.

3.7.2. Principal instantiation inheritance

If a class or interface type `X` has the inherited instantiations `V` and `W` of some generic type `Y`, then:

- for every invariant type parameter `T` of `Y`, the type argument `A` of `T` given in `V` and the type argument `B` of `T` given in `W` must be exactly the same type, and, furthermore,
- `X` is a subtype of an instantiation `U` of `Y` such that `U` is a subtype of `V&W`.

Therefore, if a type `X` is a subtype of the instantiations `V` and `W` of some generic type `Y`, then either:

- for some invariant type parameter `T` of `Y`, the argument of `A` of `T` given in `V` and the argument `B` of `T` given in `W` are distinct types, and either `A` or `B` involves a type parameter, or
- if, for some invariant type parameter `T` of `Y`, the argument of `A` of `T` given in `V` and the argument `B` of `T` given in `W` are distinct types, and neither `A` nor `B` involve a type parameter, then the type `V&W` is the bottom type `Nothing`, and we say that `V` and `W` are *disjoint instantiations* of `Y`, or, otherwise,
- `X` must be a subtype of an instantiation `P` of `Y` formed by taking each type parameter `T` of `Y`, and constructing a type argument `C` for `T` from the type arguments `A` of `T` given in `V` and `B` of `T` given in `W`:
  - if `Y` is invariant in `T`, then `C` is the same type as `A` and `B`,
  - if `Y` is covariant in `T`, then `C` is `A&B`, or
  - if `Y` is contravariant in `T`, then `C` is `A|B`.

The following identities result from principal instantiation inheritance, for any generic type `X<T>`, and for any types `A` and `B`:

- `X<A>&X<B>` is exactly equivalent to `X<A&B>` if `X<T>` is covariant in `T`, unless either `A` or `B` involves type parameters, and
- `X<A>&X<B>` is exactly equivalent to `X<A|B>` if `X<T>` is contravariant in `T`, unless either `A` or `B` involves type parameters.

3.7.3. Principal instantiation of a supertype

If a type `X` is a subtype of some instantiation `V` of a generic type `Y`, then, as a result of the principal instantiation inheritance restriction, we can form a unique instantiation of `Y` that is a subtype of every instantiation of `Y` to which `X` is assignable. We call this type the *principal instantiation of `Y` for `X`*.

We compute principal instantiations by making use of the identities observed above in §3.2.3 Union types, §3.2.4 Intersection types, and §3.7.2 Principal instantiation inheritance. For any generic type `X`:

- The principal instantiation of the union `U|V` of two instantiations of `X`, `U` and `V`, is an instantiation `P` of `X` formed by taking each type parameter `T` of `X` and constructing a type argument `C` for `T` from the type arguments `A` of `T` given in `U` and `B` of `T` given in `V`:
  - if `X` is covariant in `T`, then `C` is `A|B`,
  - if `X` is contravariant in `T`, then `C` is `A&B`, or
  - if `X` is invariant in `T`, and `A` and `B` are exactly the same type, then `C` is this type.
- The principal instantiation of the intersection `U&V` of two instantiations of `X`, `U` and `V`, is an instantiation `P` of `X` formed by taking each type parameter `T` of `X` and constructing a type argument `C` for `T` from the type arguments `A` of `T` given in `U` and `B` of `T` given in `V`:
  - if `X` is covariant in `T`, then `C` is `A&B`,
  - if `X` is contravariant in `T`, then `C` is `A|B`, or
  - if `X` is invariant in `T`, and `A` and `B` are exactly the same type, then `C` is this type.
- Finally, the principal instantiation of a generic type `X` for a type `Y` which has one or more inherited instantiations of `X` is the principal instantiation of the intersection of all the inherited instantiations of `X`.

Note: an intersection `X<A>&X<P>` of two instantiations of an invariant type, `X<T>` where one type argument `P` is a type parameter introduces a known hole in our type system. It is impossible to form a principal instantiation of `X` for this intersection type without resorting to use-site covariance, so we don't allow references to members of the intersection type.

3.7.4. Refinement

A class or interface may declare an `actual` member with the same name as a member that it inherits from a supertype if the supertype member is declared `formal` or `default`. Then we say that the first member *refines* the second member, and it must obey restrictions defined in §4.5.6 Member class refinement, §4.7.8 Method refinement, or §4.8.7 Attribute refinement.

A declaration may not be annotated both `formal` and `default`.

If a declaration is annotated `formal`, `default`, or `actual` then it must also be annotated `shared`.

For any class or interface `X`, and for every declared or inherited member of `X` that is not refined by some other declared or inherited member of `X`, and for every other member declared or inherited by `X` that directly or indirectly refines a declaration that the first member itself directly or indirectly refines, the principal instantiation for `X` of the type that declares the first member must be a subtype of the principal instantiation for `X` of the type that declares the second member.

Note: a related restriction is defined in §5.1.1 Declaration name uniqueness.

3.7.5. Qualified types

A type declaration that directly occurs in the body of another type is called a *nested type*. If a nested type is annotated `shared`, it may be used in a type expression outside the body in which it is declared, if and only if it occurs as a *qualified type*, as specified in §3.2.7 Type expressions.

The qualified types `X.U` and `Y.V` are exactly the same types if and only if `U` is exactly the same type as `V`, and in the case that this type is a member of a generic type `Z`, then the principal instantiation of `Z` for `X` is exactly the same type as the principal instantiation of `Z` for `Y`.

A qualified type `X.U` is a subtype of a qualified type `Y.V` if `U` is a subtype of `V`, and in the case that `V` is a member of a generic type `Z`, then `X` is a subtype of the principal instantiation of `Z` for `Y`.

3.7.6. Realizations

Given a member declared by `Y`, and a declaration that refines it, we can construct a *refined realization* of the member or nested type:

- first determine the principal instantiation of `Y` for the class or interface which refines the member, and then
- substitute the type arguments in this principal instantiation into the member schema.

Given an unqualified reference, as defined in §5.1.7 Unqualified reference resolution, to a declaration, and, in the case of a generic declaration, a list of type arguments for the type parameters of the declaration, we can construct an *unqualified realization* of the declaration:

- if the declaration is a member declared by a type `Y`, first determine the principal instantiation of `Y` for the inheriting or declaring class or interface, and then
- again, only if the declaration is a member declared by a type, substitute the type arguments in this principal instantiation into the declaration schema, and, finally,
- substitute the type arguments into the declaration schema.

Given a qualified reference, as defined in §5.1.8 Qualified reference resolution, with a qualifying type `X`, to a member or nested type declared by `Y`, and, in the case of a generic member or generic nested type, a list of type arguments for the type parameters of the member, we can construct a *qualified realization* of the member or nested type:

- first determining the principal instantiation of `Y` for `X`, and then
- substituting the type arguments in this principal instantiation into the declaration schema, and, finally,
- in the case of a generic member or generic nested type, substituting the type arguments into the declaration schema.

If, for any given qualified or unqualified reference, it is impossible to form the principal instantiation of the type that declares the referenced declaration, due to the hole described above in §3.7.3 Principal instantiation of a supertype, it is impossible to form a realization, and the reference to the declaration is illegal.

Chapter 4. Declarations

Ceylon is a statically typed language. Classes, interfaces, functions, values and aliases must be declared before use. The declaration of a function or value must include an explicit type, or allow the type to be inferred. Static typing allows the compiler to detect many errors, including:

- typing errors in identifier names,
- references to types which do not exist or are not visible,
- references to type members which do not exist or are not visible,
- argument lists which do not match parameter lists,
- type argument lists which do not match type parameter lists,
- operands to which an operator cannot apply,
- incompatible assignment of an expression of one type to a program element of a different type,
- evaluation of a value before it has been explicitly specified or assigned,
- assignment to a non-`variable` value,
- failure to refine a `formal` member of a supertype,
- refinement of a non-`formal`, non-`default` member of a supertype,
- `switch` statements which do not exhaust all cases of an enumerated type.

All declarations follow a general pattern:

```
Annotations
(keyword | Type) (TypeName | MemberName) TypeParameters? Parameters*
CaseTypes? ExtendedType? SatisfiedTypes?
TypeConstraints?
(Definition | ";")
```

A type parameter does not need an explicit declaration of this form unless it has constraints. In the case that it does have constraints, the constraint declaration does follow the general pattern.

This consistent pattern for declarations, together with the strict block structure of the language, makes Ceylon a highly regular language.

4.1. Compilation unit structure

A *compilation unit* is a text file, with the filename extension `.ceylon`.

Note: it is recommended that source file names contain only characters from the ASCII character set. This minimizes problems when transferring Ceylon source between operating systems.

There are three kinds of compilation unit:

- A regular compilation unit contains a list of toplevel type, value, or function definitions.
- A *module descriptor*, defined in §9.3.12 Module descriptors, contains a `module` declaration. The file must be named `module.ceylon`.
- A *package descriptor*, defined in §9.3.11 Package descriptors, contains a `package` declaration. The file must be named `package.ceylon`.

Any compilation unit may begin with a list of imported types, values, and functions.

```
Import* (ModuleDescriptor | PackageDescriptor | Declaration*)
```

4.1.1. Toplevel and nested declarations

A *toplevel declaration* defines a type—a class or interface—or a type alias, or a function or value.

```
Declaration: FunctionValueDeclaration | TypeDeclaration | ParameterDeclaration
```

```
FunctionValueDeclaration: FunctionDeclaration | ValueDeclaration | SetterDeclaration
```

```
TypeDeclaration: ClassDeclaration | ObjectDeclaration | InterfaceDeclaration | TypeAliasDeclaration
```

A toplevel declaration is not polymorphic and so may not be annotated `formal`, `default`, or `actual`.

Note: in a future release of the language, we might relax this restriction and support package extension with toplevel member refinement. This can be viewed as a regularization of the language. The practical application is that it would make toplevel invocations and instantiations polymorphic, obviating the need for things like dependency injection.

Most toplevel declarations contain nested declarations.

Nested declarations are often mixed together with executable statements.

4.1.2. Packages

Each compilation unit belongs to exactly one *package*. Every toplevel declaration of the compilation unit also belongs directly to this package. The package is identified by the location of the text file on the file system, relative to a root *source directory*, as defined in §9.2 Source layout.

A package is a namespace. A full package name is a period-separated list of all-lowercase identifiers.

```
FullPackageName: PackageName ("." PackageName)*
```

Note: it is recommended that package names contain only characters from the ASCII character set.

There is also a *default package* whose name is empty. It is impossible to import declarations from this package.

Every package belongs to exactly one module, as specified in §9.3 Module architecture. The default package belongs to the default module.

4.2. Imports

Code in one compilation unit may refer to a toplevel declaration in another compilation unit in the same package without explicitly importing the declaration. It may refer to a declaration defined in a compilation unit in another package only if it explicitly *imports* the declaration using the `import` statement.

```
Import: "import" FullPackageName "{" ImportElements "}"
```

For a given package, there may be at most one `import` statement per compilation unit.

An `import` statement may import from a package if and only if:

- the package belongs to the same module as the compilation unit containing the `import` statement, as specified by §9.2 Source layout, or
- the package is declared `shared` in its package descriptor, and the module descriptor of the module to which the compilation unit containing the `import` statement belongs, as specified by §9.2 Source layout, explicitly or implicitly imports the module containing the package, as defined by §9.3.12 Module descriptors.

Each `import` statement imports one or more toplevel declarations from the given package, specifying a list of *import elements*.

```
ImportElements: (ImportElement ",")* (ImportElement | ImportWildcard)?
```

```
ImportElement: ImportTypeElement | ImportFunctionValueElement
```

An import element is a reference to either:

- a single toplevel type (a class, interface, or alias) of the package,
- a single toplevel function or value of the package, or
- all toplevel declarations of the package.

An import element may not refer to a declaration that is not visible to the compilation unit, as defined by §5.1.3 Visibility.

An `import` statement may not contain two import elements which refer to the same declaration.

Note that toplevel declarations in the package `ceylon.language` never need to be explicitly imported. They are implicitly imported by every compilation unit.

An imported function or value may not hide, as defined in §5.1.4 Hidden declarations, any of the modifiers declared in `ceylon.language` listed in §7.4.1 Declaration modifiers, unless the modifier itself has an alias import in the compilation unit.

Note: an unused import results in a compiler warning.

4.2.1. Type imports

An import element that specifies a type name imports the toplevel type with that name from the given package.

```
ImportTypeElement: TypeAlias? TypeName ("{" ImportElements? "}")?
```

A compilation unit may not import two types with the same name.

```
import java.util { Set, List, Map }
```

The import element may be followed by a list of nested import elements, which must specify aliases.

Note: as a special exception to the usual language rules, to support interoperation with Java, a nested import element which references a `static` member of a Java type results in a Ceylon toplevel reference to the `static` member. In this case, the import element may omit the explicit alias.

4.2.2. Function and value imports

An import element that specifies a function or value name imports the toplevel function or value with that name from the given package.

```
ImportFunctionValueElement: FunctionValueAlias? MemberName
```

A compilation unit may not import two methods or attributes with the same name.

```
import ceylon.math { sqr, sqrt, e, pi }
```

4.2.3. Alias imports

The optional alias clause in a fully-explicit import allows resolution of cross-namespace declaration name collisions.

```
TypeAlias: TypeName "="
```

```
FunctionValueAlias: MemberName "="
```

An alias assigns a different name to the imported declaration, or to a member of the imported declaration. This name is visible within the compilation unit in which the `import` statement occurs.

```
import java.util { JavaMap = Map }
```

```
import my.math { fib = fibonnacciNumber }
```

```
import java.lang {
    Math { sin, cos, ln=log },
    System { sysprops=properties },
    Char=Character { upper=toUpperCase, lower=toLowerCase, char=charValue }
}
```

4.2.4. Wildcard imports

The elipsis `...` acts as a wildcard in `import` statements. An `import` statement that specifies a wildcard imports every toplevel declaration of the imported package, except for any declaration whose name collides with the name of a toplevel declaration in the compilation unit in which the `import` statement appears.

```
ImportWildcard: "..."
```

An `import` statement may specify a list of alias imports followed by a wildcard. In this case, the alias imports are imported with the specified names, and all other toplevel declarations are imported with their declared names.

```
import ceylon.collection { ... }
```

```
import my.math { fib = fibonnacciNumber, ... }
```

Note: overuse of wildcard imports is discouraged.

4.2.5. Imported name

Inside a compilation unit which imports a declaration, the declaration may be referred to, as specified in §5.1.7 Unqualified reference resolution and §5.1.8 Qualified reference resolution, by its *imported name*:

- For an import element with an alias, the imported name is the alias.
- For an import element with no alias, or for a wildcard import, the imported name is the original name of the declaration.

An import element may not result in an imported name that is the same as the name of a toplevel declaration contained in the compilation unit in which the import element occurs.

Two import elements occurring in the same compilation unit may not result in the same imported name.

4.3. Parameters

A function or class declaration may declare *parameters*. A parameter is a value or function belonging to the declaration it parameterizes. Parameters are distinguished from other values or functions because they occur in a *parameter list*. A value or function is a parameter of a class or function if it is:

- declared inline in a parameter list of the class or function, or
- declared normally, within the body of the class or function, but named in a parameter list of the class or function.

The following class definitions are semantically identical:

```
class Person(shared String name, shared variable Integer age=0, Address* addresses) {}
```

```
class Person(name, age=0, addresses) {
    shared String name;
    shared variable Integer age;
    Address* addresses;
}
```

A parameter declaration may only occur in a parameter list, or directly, as defined by §5.1 Block structure and references, in the body of a class or function. A parameter declaration may not occur directly in the body of a getter or in a body of a control structure. Nor may a parameter declaration appear as a toplevel declaration in a compilation unit.

```
ParameterDeclaration: ValueParameter | CallableParameter | VariadicParameter
```

Every parameter declaration that occurs outside a parameter list must be named in the parameter list of the class or function in whose body it directly occurs, and its default argument, if any, must be specified in the parameter list.

4.3.1. Parameter lists

A parameter list is a list of parameter declarations and of names of parameters declared in the body of the class or function to which the parameter list belongs. A parameter list may include, optionally:

- one or more *required parameters*,
- one or more *defaulted parameters* (parameters with default values), and/or
- a *variadic parameter*.

In a parameter list, defaulted parameters, if any, must occur after required parameters, if any. The variadic parameter, if any, must occur last.

```
Parameters: "(" ( (Required ",")* ( Required | (Defaulted ",")* (Defaulted | Variadic) ) )? ")"
```

Every parameter list has a type, which captures the types of the individual parameters in the list, whether they are defaulted, and whether the last parameter is variadic. This type is always an subtype of `Anything[]`. The type of an empty parameter list with no parameters is `[]`.

A parameter may not be annotated `formal`, but it may be annotated `default`.

4.3.2. Required parameters

A required parameter is a value or callable parameter without a default argument.

A required parameter in a parameter list may be a parameter declaration, or the name of a non-variadic parameter declared in the body of the function or class.

```
Required: ValueParameter | CallableParameter | MemberName
```

Required parameters must occur before any other parameters in the parameter list.

4.3.3. Defaulted parameters

A defaulted parameter is a value or callable parameter that specifies an expression that produces a *default argument*. A defaulted parameter may be either:

- a non-variadic parameter declaration, together with a default argument expression, or
- the name of a non-variadic parameter declared in the body of the function or class, together with its default argument expression.

```
Defaulted: ValueParameter Specifier | CallableParameter LazySpecifier | MemberName Specifier
```

The `=` and `=>` specifiers are used throughout the language. In a parameter list they are used to specify a default argument.

```
Specifier: "=" Expression
```

```
LazySpecifier: "=>" Expression
```

The default argument expression may involve other parameters declared earlier in the parameter list or lists. It may not involve parameters declared later in the parameter list or lists.

The default argument expression may not involve an assignment, compound assignment, increment, or decrement operator.

Defaulted parameters must occur after required parameters in the parameter list.

```
(Product product, Integer quantity=1, Price pricing(Product p) => p.price)
```

A parameter of a method or class annotated `actual` may not specify a default argument. Instead, it inherits the default argument, if any, of the corresponding parameter of the method it refines.

If two parameter lists are almost identical, differing only in that the first parameter of one list is defaulted, and the first parameter of the second list is required, and `P` is the the type of the second parameter list, then the type of the first parameter list is `[]|P`.

Note: in Ceylon 1.0, for a function with multiple parameter lists, defaulted parameters may only occur in the first parameter list. This restriction will be removed.

TODO: Should we, purely for consistency, let you write `f(Float x) => x` in a parameter list, when the callable parameter is declared in the body of the function or class?

4.3.4. Value parameters

A *value parameter* is a reference, as specified in §4.8.1 References, that is named or defined in a parameter list. Like any other value declaration, it has a name, type, and, optionally, annotations.

```
ValueParameter: Annotations (Type | "dynamic") MemberName
```

A value parameter may be declared using the keyword `dynamic` in place of the parameter type. Such a parameter has no type.

If a value parameter `x` has type `X`, and a parameter list has type `P` with the principal instantiation `Sequential<Y>`, then the type of a new parameter list formed by prepending `x` to the first parameter list is:

- `Tuple<X|Y,X,P>`, or
- `[]|Tuple<X|Y,X,P>` if `x` is defaulted.

The default argument expression, if any, for a callable parameter is specified using an ordinary `=` specifier. The type of the default argument expression must be assignable to the declared type of the value parameter.

```
(String label, Anything() onClick)
```

```
({Value*} values, Comparison(Value,Value) by)
```

4.3.5. Callable parameters

A *callable parameter* is a function, as specified in §4.7 Functions, named or defined in a parameter list. Like any other function declaration, it has a name, type, one or more parameter lists, and, optionally, annotations.

```
CallableParameter: Annotations (Type | "void") MemberName Parameters+
```

If a callable parameter `f` has callable type `Callable<X,A>`, as specified below in §4.7.1 Callable type of a function, and a parameter list has type `P` with the principal instantiation `Sequential<Y>`, then the type of a new parameter list formed by prepending `f` to the first parameter list is:

- `Tuple<Y|Callable<X,A>,Callable<X,A>,P>`, or
- `[]|Tuple<Y|Callable<X,A>,Callable<X,A>,P>` if `f` is defaulted.

The default argument expression, if any, for a callable parameter is specified using a lazy `=>` specifier. The type of the default argument expression must be assignable to the return type of the callable parameter.

```
(String label, void onClick())
```

```
({Value*} values, Comparison by(Value x, Value y))
```

4.3.6. Variadic parameters

A *variadic parameter* is a value parameter that accepts multiple arguments:

- A variadic parameter declared `T*` accepts zero or more arguments of type `T`, and has type `[T*]`.
- A variadic parameter declared `T+` accepts one or more arguments of type `T`, and has type `[T+]`.

```
VariadicType: UnionType ("*" | "+")
```

```
VariadicParameter: Annotations VariadicType MemberName
```

A variadic parameter in a parameter list may be a variadic parameter declaration, or the name of a variadic parameter declared in the body of the function or class.

```
Variadic: VariadicParameter | MemberName
```

The variadic parameter must be the last parameter in a parameter list. A variadic parameter may not have a default argument. A variadic parameter declared `T+` may not occur in a parameter list with defaulted parameters.

```
(Name name, Organization? org=null, Address* addresses)
```

```
(Float+ floats)
```

The type of a parameter list containing just a variadic parameter of type `T*` is `[T*]` The type of a parameter list containing just a variadic parameter of type `T+` is `[T+]`.

Note: in Ceylon 1.0, for a function with multiple parameter lists, a variadic parameters may only occur in the first parameter list. This restriction will be removed.

4.4. Interfaces

An *interface* is a type schema, together with implementation details for some members of the type. Interfaces may not be directly instantiated.

```
InterfaceDeclaration: Annotations InterfaceHeader (InterfaceBody | TypeSpecifier ";")
```

An interface declaration may optionally specify a list of type parameters. An interface declaration may also have a list of interfaces is satisfies, a self type or an enumerated list of cases, and/or a list of type constraints.

```
InterfaceHeader: "interface" TypeName TypeParameters? InterfaceInheritance TypeConstraints?
```

```
InterfaceInheritance: CaseTypes? SatisfiedTypes?
```

To obtain a concrete instance of an interface, it is necessary to define and instantiate a class that satisfies the interface, or define an anonymous class that satisfies the interface.

The body of an interface contains:

- member (method, attribute, and member class) declarations, and
- nested interface, type alias, and `abstract` class declarations.

```
InterfaceBody: "{" Declaration* "}"
```

Unlike the body of a class, method, or attribute, the body of an interface is not executable, and does not directly contain procedural code.

```
shared interface Comparable<Other> {
    shared formal Comparison compare(Other other);
    shared Boolean largerThan(Other other) => compare(other)==larger;
    shared Boolean smallerThan(Other other) => compare(other)==smaller;
```

```
}
```

An interface may declare `formal` methods, attributes, and member classes, and concrete methods, getters, setters, and member classes. A reference declaration, as defined in §4.8.1 References, or anonymous class declaration, as defined in §4.5.7 Anonymous classes, may not directly occur in the body of an interface.

A non-`abstract` nested class declaration is called a *member class* of the interface. A nested interface or `abstract` class declaration is not part of the schema of the interface type, and is therefore not considered a member of the interface.

4.4.1. Interface bodies

The body of an interface consists purely of declarations. The following constructs may not occur sequentially in the body of an interface:

- a statement or control structure,
- a reference declaration,
- a forward-declared method or attribute declaration, or
- an `object` declaration.

Within an interface body, a *super reference* is any occurrence of the expression `super`, unless it also occurs in the body of a nested class or interface declaration. A statement or declaration contained in the interface body may not:

- pass a super reference as an argument of an instantiation, method invocation, or `extends` clause expression or as the value of a value assignment or specification,
- use a super reference as an operand of any operator except the member selection operator, or the `of` operator as specified in §6.3.3 super,
- return a super reference, or
- narrow the type of a super reference using the `if (is ...)` construct or `case (is ...)`.

4.4.2. Interface inheritance

An interface may satisfy any number of other interfaces.

```
shared interface List<Element>
        satisfies Collection<Element> & Correspondence<Integer,Element>
        given Element satisfies Object {
    ...
}
```

Every type listed in the `satisfies` clause must be an interface. An interface may not satisfy the same interface twice (not even with distinct type arguments).

Note: this second restriction is not strictly necessary. In fact, `satisfies List<One>&List<Two>` means the same thing as `satisfies List<One&Two>`, and the compiler already needs to be able to figure that out when it comes to multiple instantiations of the same interface inherited indirectly. Still, the restriction seems harmless enough.

The interface is a subtype of every type listed in the `satisfies` clause. The interface is also a subtype of the type `Object` defined in `ceylon.language`.

An interface inherits all members (methods, attributes and member types) of every supertype. That is, every member of every supertype of the interface is also a member of the interface. Furthermore, the interface inherits all nested types (interfaces and `abstract` classes) of every supertype.

The schema of the inherited members is formed by substituting type arguments specified in the `satisfies` clause.

An interface that satisfies a nested interface must be a member of the type that declares the nested interface or of a subtype of the type that declares the nested interface.

A user-defined interface may not satisfy the interface `Callable` defined in `ceylon.language`.

4.4.3. Interfaces with enumerated cases

An interface declaration may enumerate a list of cases of the interface.

```
shared interface Node<Element>
            of Root<Element> | Branch<Element> | Leaf<Element> { ... }
```

The cases may be interfaces, classes, or toplevel anonymous classes. A case may be an `abstract` class. Each case must be a subtype of the interface type. An interface may not be a case of itself. An interface declaration may not list the same case twice.

If an interface has an `of` clause, then every interface or class which is a subtype of the interface must be subtype of exactly one of the enumerated cases.

4.4.4. Interface aliases

An interface declaration which specifies a reference to another interface type defines an *interface alias* of the specified interface type.

```
TypeSpecifier: "=>" Type
```

The specified type must be an *interface type*, that is, a reference to an interface with no type parameters, or an instantiation of a generic interface. An interface alias simply assigns an alternative name to the original interface type. A reference to the alias may occur anywhere a reference to an interface may occur.

```
shared interface PeopleByName => Map<String,Person>;
```

```
interface Compare<Value> => Comparison(Value,Value);
```

If the aliased interface is a parameterized type, the aliased type must explicitly specify type arguments.

A class or interface may satisfy an interface alias, in which case, the class or interface inherits the aliased interface type.

Interface aliases are not reified types. The metamodel reference for an interface alias type—for example, `PeopleByName`—returns the metamodel object for the aliased interface—in this case, `Map<String,Person>`, as specified in §8.1.2 Type argument reification.

4.5. Classes

A *class* is a stateful, instantiable type. It is a type schema, together with implementation details of the members of the type.

```
ClassDeclaration: Annotations ClassHeader (ClassBody | ClassSpecifier ";")
```

An ordinary class declaration specifies a list of parameters required to instantiate the type, and, optionally a list of type parameters. A class declaration may have a superclass, a list of interfaces it satisfies, a self type or an enumerated list of cases, and/or a list of type constraints.

```
ClassHeader: "class" TypeName TypeParameters? Parameters ClassInheritance TypeConstraints?
```

```
ClassInheritance: CaseTypes? ExtendedType? SatisfiedTypes?
```

To obtain an instance of a class, it is necessary to instantiate the class, or a subclass of the class.

The body of a class contains:

- member (method, attribute, and member class) declarations,
- nested interface, type alias, and `abstract` class declarations, and

- instance initialization code.

```
ClassBody: "{" (Declaration | Statement)* "}"
```

The body of a class may contain executable code.

```
shared class Counter(Integer initialCount=0) {

    variable Integer n = initialCount;

    print("Initial count: ``n``");

    shared Integer count => n;

    shared void increment() {
        n++;
        print("Count: ``n``");
    }

}
```

A non-`abstract` nested class declaration is called a *member class* of the class. A nested interface or `abstract` class declaration is not part of the schema of the class type, and is therefore not considered a member of the class.

Ceylon classes do not have seperate nested constructor declarations. Instead, the body of the class declares *initializer parameters*. An initializer parameter may be used anywhere in the class body, including in method and attribute definitions.

```
shared class Key(Lock lock) {
    shared void lock() {
        lock.engage(this);
        print("Locked.");
    }
    shared void unlock() {
        lock.disengage(this);
        print("Unlocked.");
    }
    shared Boolean locked => lock.engaged;
}
```

An initializer parameter may be `shared`.

```
shared class Point(shared Float x, shared Float y) { ... }
```

```
shared class Counter(count=0) {
    shared variable Integer count;
    shared void increment() => count++;
}
```

4.5.1. Callable type of a class

The *callable type* of a class captures the type and parameter types of the class. The callable type is `Callable<T,P>`, where `T` is the class and `P` is the type of the initializer parameter list of the class.

An `abstract` class is not callable, except from the `extends` clause of a subclass, or the class specifier of a class alias.

4.5.2. Initializer section

The initial part of the body of a class is called the *initializer* of the class and contains a mix of declarations, statements and control structures. The initializer is executed every time the class is instantiated.

A class initializer is responsible for initializing the state of the new instance of the class, before a reference to the new instance is available to clients.

```
shared abstract class Point() {
    shared formal Float x;
    shared formal Float y;
}
```

```
shared class DiagonalPoint(Float distance)
        extends Point() {
```

```
    value d = distance / 2^0.5;
    x => d;
    y => d;

    "must have correct distance from origin"
    assert (x^2 + y^2 == distance^2);

}
```

```
shared object origin
        extends Point() {
    x => 0.0;
    y => 0.0;
}
```

Within a class initializer, a *self reference to the instance being initialized* is:

- any occurrence of the expression `this` or `super`, unless it also occurs in the body of a nested class or interface declaration, or
- any occurrence of the expression `outer` in the body of a class or interface declaration immediately contained by the class.

A statement or declaration contained in the initializer of a class may not evaluate an attribute, invoke a method, or instantiate a member class upon the instance being initialized, including upon a self reference to the instance being initialized, if the attribute, method, or member class:

- occurs later in the body of the class,
- is annotated `formal` or `default`, or
- is inherited from an interface or superclass, and is not refined by a declaration occurring earlier in the body of the class.

A member class contained in the initializer of a class may not `extend` a member or nested class of an interface or superclass of the class.

Furthermore, a statement or declaration contained in the initializer of a class may not:

- pass a self reference to the instance being initialized as an argument of an instantiation, method invocation, or `extends` clause expression or as the value of a value assignment or specification,
- use a self reference to the instance being initialized as an operand of any operator except the member selection operator, or the `of` operator,
- return a self reference to the instance being initialized, or
- attempt to narrow the type of a self reference to the instance being initialized using the `if (is ...)` construct or `case (is ...)`.

Nor may the class pass a self reference to the instance being initialized as an argument of its own `extends` clause expression, if any.

As a special exception to these rules, a statement contained in an initializer may assign a self-reference to the instance being initialized to a reference annotated `late`.

For example, the following code fragments are not legal:

```
class Graph() {
    OpenList<Node> nodes = ArrayList<Node>();
    class Node() {
        nodes.add(this);    //compiler error (this reference in initializer)
    }
}
```

```
class Graph() {
    class Node() {}
    Node createNode() {
```

```
        Node node = Node();
        nodes.add(node);    //compiler error (forward reference in initializer)
        return node;
    }
    OpenList<Node> nodes = ArrayList<Node>();
}
```

But this code fragment is legal:

```
class Graph() {
    OpenList<Node> nodes = ArrayList<Node>();
    Node createNode() {
        Node node = Node();
        nodes.add(node);
        return node;
    }
    class Node() {}
}
```

4.5.3. Declaration section

The remainder of the body of the class consists purely of declarations, similar to the body of an interface. The following constructs may not occur sequentially in the declaration section:

- a statement or control structure,
- a reference declaration,
- a forward-declared method or attribute declaration not annotated `late`,
- an `object` declaration with a non-empty initializer section, or
- an `object` declaration that directly extends a class other than `Object` or `Basic` in `ceylon.language`.

However, the declarations in this second section may freely use `this` and `super`, and may invoke any method, evaluate any attribute, or instantiate any member class of the class or its superclasses.

Within the declaration section of a class body, a *super reference* is any occurrence of the expression `super`, unless it also occurs in the body of a nested class or interface declaration. A statement or declaration contained in the declaration section of a class body may not:

- pass a super reference as an argument of an instantiation, method invocation, or `extends` clause expression or as the value of a value assignment or specification,
- use a super reference as an operand of any operator except the member selection operator, or the `of` operator as specified in §6.3.3 super,
- return a super reference, or
- narrow the type of a super reference using the `if (is ...)` construct or `case (is ...)`.

4.5.4. Class inheritance

A class may extend another class.

```
shared class Customer(Name name, Organization? org = null)
        extends Person(name, org) {
    ...
}
```

The class is a subtype of the type specified by the `extends` clause. If a class does not explicitly specify a superclass using `extends`, its superclass is the class `Basic` defined in `ceylon.language`.

A class may satisfy any number of interfaces.

```
class Token()
        extends Datetime()
```

```
        satisfies Comparable<Token> & Identifier {
    ...
}
```

The class is a subtype of every type listed in the `satisfies` clause. A class may not satisfy the same interface twice (not even with distinct type arguments).

A class inherits all members (methods, attributes, and member types) of every supertype. That is, every member of every supertype of the class is also a member of the class. Furthermore, the class inherits all nested types (interfaces and `abstract` classes) of every supertype.

Unless the class is declared `abstract` or `formal`, the class:

- must declare or inherit a member that refines each `formal` member of every interface it satisfies directly or indirectly, and
- must declare or inherit a member that refines each `formal` member of its superclass.

The schema of the inherited members is formed by substituting type arguments specified in the `extends` or `satisfies` clause.

A subclass must pass values to each superclass initialization parameter in the `extends` clause.

```
shared class SpecialKey1()
        extends Key( SpecialLock() ) {
    ...
}
```

```
shared class SpecialKey2(Lock lock)
        extends Key(lock) {
    ...
}
```

A subclass of a nested class must be a member of the type that declares the nested class or of a subtype of the type that declares the nested class. A class that satisfies a nested interface must be a member of the type that declares the nested interface or of a subtype of the type that declares the nested interface.

A user-defined class may not satisfy the interface `Callable` defined in `ceylon.language`.

4.5.5. Abstract, final, formal, and default classes

A toplevel or nested class may be annotated `abstract` and is called an `abstract` class.

A toplevel or nested class may be annotated `final` and is called an `final` class.

If a class annotated `shared` is a member of a containing class or interface, then the class may be annotated `formal` and is called a `formal` member class, or, sometimes, an *abstract member class*.

An `abstract` class or `formal` member class may have `formal` members.

An `abstract` class may not be instantiated.

A `formal` member class may be instantiated.

A class which is not annotated `formal` or `abstract` is called a *concrete* class.

A concrete class may not have `formal` members.

A class annotated `final` must be a concrete class.

A class annotated `final` may not have `default` members.

If a concrete class annotated `shared` is a member of a containing class or interface, then the class may be annotated `default` and is called a `default` member class.

A toplevel class may not be annotated `formal` or `default`.

An un-`shared` class may not be annotated `formal` or `default`.

Note: a `formal` member class would be a reasonably syntax for declaring virtual types. We think we don't need virtual types because they don't offer much that type parameters don't already provide. For example:

```
shared formal class Buffer(Character...)
        satisfies Sequence<Character>;
```

4.5.6. Member class refinement

Member class refinement is a unique feature of Ceylon, akin to the "factory method" pattern of many other languages.

- A member class annotated `formal` or `default` may be refined by any class or interface which is a subtype of the class or interface which declares the member class.
- A member class annotated `formal` *must* be refined by every concrete class which is a subtype of the class or interface that declares the member class, unless the class inherits a concrete member class from a superclass that refines the `formal` member class.

A member class of a subtype *refines* a member class of a supertype if the member class of the supertype is `shared` and the two classes have the same name. The first class is called the *refining* class, and the second class is called the *refined* class.

Then, given the refined realization of the class it refines, as defined in §3.7.6 Realizations, and, after substituting the type parameters of the refined class for the type parameters of the refining class in the schema of the refining class, the refining class must:

- have the same number of type parameters as the refined schema, and for each type parameter the intersection of its upper bounds must be a supertype of the intersection of the upper bounds of the corresponding type parameter of the realization,
- have a parameter list with the same signature as the realization, and
- directly or indirectly extend the class it refines.

Furthermore:

- the refining class must be annotated `actual`, and
- the refined class must be annotated `formal` or `default`.

If a member class is annotated `actual`, it must refine some member class of a supertype.

A member class may not, directly or indirectly, refine two different member classes not themselves annotated `actual`.

Then instantiation of the member class is polymorphic, and the actual subtype instantiated depends upon the concrete type of the containing class instance.

```
shared abstract class Reader() {
    shared formal class Buffer(Character* chars)
            satisfies Character[] {}
    ...
}
```

```
shared class FileReader(File file)
        extends Reader() {
    shared actual class Buffer(Character* chars)
            extends Reader.Buffer(chars) {
        ...
    }
    ...
}
```

All of the above rules apply equally to member classes which are aliases.

```
shared abstract class Reader() {
    shared formal class Buffer(Character* chars) => AbstractBuffer(*chars);
```

```
    ...
}
```

```
shared class FileReader(File file)
        extends Reader() {
    shared actual class Buffer(Character* chars) => FileBuffer(*chars);
    ...
}
```

4.5.7. Anonymous classes

An `object` declaration makes it possible to define a class, instantiate the class, and declare an attribute referring to the resulting class instance in a single declaration.

```
ObjectDeclaration: Annotations ObjectHeader ClassBody
```

An `object` has an initial lowercase identifier. An `object` declaration does not specify parameters or type parameters.

```
ObjectHeader: "object" MemberName ObjectInheritance
```

```
ObjectInheritance: ExtendedType? SatisfiedTypes?
```

An `object` declaration specifies the name of the attribute and the schema, supertypes, and implementation of the class. It does not specify a type name. Instead, the type has a name assigned internally by the compiler that is not available at compilation time.

An `object` class:

- is implicitly `final`,
- may not be extended by another class,
- may not be `abstract` or `formal`, and
- may not declare `default` members.

If the `object` is annotated `shared`, the class is `shared`.

This class never appears in types inferred by local declaration type inference or generic type argument inference. Instead, occurrences of the class are replaced with the intersection of the extended type with all satisfied types.

An `object` attribute:

- is non-`variable`, and
- may not be refined or declared `default`.

If the `object` is annotated `shared`, the attribute is `shared`. If the `object` is annotated `actual`, it refines an attribute of a supertype.

The following declaration:

```
shared object red extends Color('FF0000') {
     string => "Red";
}
```

Is exactly equivalent to:

```
shared final class \Ired() extends Color('FF0000') {
     string => "Red";
}

shared \Ired red = \Ired();
```

Where `\Ired` is a name generated by the compiler. The algorithm for generating this name is not specified here.

Note that a member of an anonymous class that is not annotated `actual` may only be accessed from within the body of the anonymous class or by directly invoking the `object` attribute.

```
shared object sql {
    shared String escape(String string) { ... }
}

...

String escapedSearchString = sql.escape(searchString);
```

4.5.8. Classes with enumerated cases

A class declaration may enumerate a list of cases of the class.

```
shared abstract class Boolean()
        of true | false {}

shared object true extends Boolean() { string => "true"; }
shared object false extends Boolean() { string => "false"; }
```

```
shared abstract class Node<Element>(String name)
        of Branch<Element> | Leaf<Element> { ... }

shared class Leaf<Element>(String name, Element element)
        extends Node<Element>(name) { ... }

shared class Branch<Element>(String name, Node<Element> left, Node<Element> right)
        extends Node<Element>(name) { ... }
```

The cases may be classes or toplevel anonymous classes. A case may be an `abstract` class. Each case must be a subclass of the class. A class may not be a case of itself. A class declaration may not list the same case twice.

If a class has an `of` clause, then every class that directly extends the class must be of one of the enumerated cases of the class.

A non-`abstract` class may not have an `of` clause.

Note: in a future release of the language, we will introduce an abbreviated syntax like:

```
shared abstract class Boolean(shared actual String string)
        of object true ("true") |
           object false ("false") {}
```

4.5.9. Class aliases

A class declaration which specifies a reference to another class type defines a *class alias* of the specified class type.

```
ClassSpecifier: "=>" ("super" ".")? TypeNameWithArguments PositionalArguments
```

The specified type must be a *class type*, that is, a reference to a class with no type parameters, or an instantiation of a generic class. A class alias simply assigns an alternative name to the original class type. A reference to the alias may occur anywhere a reference to a class may occur.

```
shared class People(Person* people) => ArrayList<Person>(*people);
```

```
class Named<Value>(String name, Value val) => Entry<String,Value>(name, val);
```

Arguments to the initializer parameters of the aliased class must be specified.

If the aliased class is a parameterized type, the aliased type must explicitly specify type arguments.

The type arguments may not be inferred from the initializer arguments.

Note: currently the compiler imposes a restriction that the callable type of the aliased class must be assignable to the callable type of the class alias. This restriction will be removed in future.

If a toplevel class alias or un-`shared` class alias aliases an `abstract` class, the alias must be annotated `abstract`, and it may not be directly instantiated.

If a `shared` class alias nested inside the body of a class or interface aliases an `abstract` class, the alias must be annotated `abstract` or `formal`. If it is annotated `formal`, it is considered a member class of the containing class or interface. If it is annotated `abstract`, it is considered an abstract nested class of the containing class or interface.

A class or interface may extend a class alias, in which case, the class inherits the aliased class type.

Class aliases are not reified types. The metamodel reference for a class alias type—for example, `People`—returns the metamodel object for the aliased class—in this case, `ArrayList<Person>`, as specified in §8.1.2 Type argument reification.

4.6. Type aliases

A type alias declaration assigns a name to an arbitrary type expression, usually involving a union and/or intersection of types.

```
TypeAliasDeclaration: Annotations AliasHeader TypeSpecifier
```

```
AliasHeader: "alias" TypeName TypeParameters? TypeConstraints?
```

The specified type may be any kind of type. A reference to the alias may be used anywhere a union or intersection type may be used. The alias may not appear in an `extends` or `satisfies` clause. The alias may not be instantiated.

```
shared alias Number => Integer|Float|Decimal|Whole;
```

```
alias ListLike<Value> => List<Value>|Map<Integer,Value>;
```

```
alias Numbered<Num,Value> given Num satisfies Ordinal<Num>
        => Correspondence<Num,Value>;
```

Note: class, interface, and type aliases use a "fat arrow" lazy specifier `=>` instead of `=` because the type parameters declared on the left of the specifier are in scope on the right of the specifier. An alias is in general a type constructor.

A class or interface may not extend or satisfy a type alias.

Type aliases are not reified types. The metamodel reference for a type alias type—for example, `Number`—returns the metamodel object for the aliased type—in this case, `Integer|Float|Decimal|Whole`, as specified in §8.1.2 Type argument reification.

4.7. Functions

A *function* is a callable block of code. A function may have parameters and may return a value. If a function belongs to a type, it is called a *method*.

```
FunctionDeclaration: Annotations FunctionHeader (Block | LazySpecifier? ";")
```

All function declarations specify the function name, one or more parameter lists, and, optionally, a list of type parameters. A function declaration may specify a type, called the *return type*, to which the values the method returns are assignable, or it may specify that the function is a `void` function—a function which does not return a useful value, and only useful for its effect. A generic function declaration may have a list of type constraints.

```
FunctionHeader: (Type | "function" | "dynamic" | "void") MemberName TypeParameters? Parameters+ TypeConstraints?
```

A function may be declared using the keyword `dynamic` in place of its return type. Such a function has no return type.

A function implementation may be specified using either:

- a block of code, or
- a lazy specifier.

If a function is a parameter, it must not specify any implementation.

The return type of a `void` function is considered to be `Anything` defined in `ceylon.language`.

Note: a `void` function with a concrete implementation returns the value `null`. However, since a `void` function may be a reference to a non-`void` function, or a method refined by a non-`void` function, this behavior can not be depended upon and is not implied by the semantics of `void`.

4.7.1. Callable type of a function

The *callable type* of a function captures the return type and parameter types of the function.

- The callable type of a function with a single parameter list is `Callable<R,P>` where `R` is the return type of the method, or `Anything` if the function is `void`, and `P` is the type of the parameter list.
- The callable type of a function with multiple parameter lists is `Callable<O,P>`, where `O` is the callable type of a method produced by eliminating the first parameter list, and `P` is the type of the first parameter list of the function.

Note: the identification of `void` with `Anything` instead of `Null` or some other unit type will probably be contraversial. This approach allows a non-`void` method to refine a `void` method or a non-`void` function to be assigned to a `void` functional parameter. Thus, we avoid rejecting perfectly well-typed code.

4.7.2. Functions with blocks

A function implementation may be a block.

- If the function is declared `void`, the block may not contain a `return` directive that specifies an expression.
- Otherwise, every conditional execution path of the block must end in a `return` directive that specifies an expression assignable to the return type of the function, or in a `throw` directive, as specified in §5.2.4 Definite return.

```
shared Integer add(Integer x, Integer y) {
    return x + y;
}
```

```
shared void printAll(Object* objects) {
    for (obj in objects) {
        print(obj);
    }
}
```

```
shared void addEntry(Key->Item entry) {
    map.put(entry.key,entry.item);
}
```

```
shared Set<Element> singleton<Element>(Element element)
        given Element satisfies Comparable<Element> {
    return TreeSet { element };
}
```

4.7.3. Functions with specifiers

Alternatively, a function implementation may be a lazy specifier, that is, an expression specified using `=>`. The type of the specified expression must be assignable to the return type of the function. In the case of a function declared `void`, the expression must be a legal statement.

```
shared Integer add(Integer x, Integer y) => x + y;
```

```
shared void addEntry(Key->Item entry) => map.put(entry.key,entry.item);
```

```
shared Set<Element> singleton<Element>(Element element)
           given Element satisfies Comparable<Element>
       => TreeSet { element };
```

4.7.4. Function return type inference

A non-`void`, un-`shared` function with a block or lazy specifier may be declared using the keyword `function` in place of the explicit return type declaration. Then the function return type is inferred:

- if the function implementation is a lazy specifier, then the return type of the function is the type of the specified expression,
- if the function implementation is a block, and the function contains no `return` directive, then the return type of the method is `Bottom` (this is the case where the method always terminates in a `throw` directive), or,
- otherwise, the return type of the function is the union of all returned expression types of `return` directives of the method body.

This function has inferred return type `Integer`.

```
function add(Integer x, Integer y) => x + y;
```

This function has inferred return type `Float|Integer`.

```
function unit(Boolean floating) {
    if (floating) {
        return 1.0;
    }
    else {
        return 1;
    }
}
```

This function has inferred return type `Bottom`.

```
function die() {
    throw;
}
```

4.7.5. Forward declaration of functions

The declaration of a function may be separated from the specification of its implementation. If a function declaration does not have a lazy specifier, or a block, and is not annotated `formal`, and is not a parameter, it is a *forward-declared* function.

A forward-declared function may later be specified using a specification statement, as defined in §5.2.3 Specification statements. The specification statement for a forward-declared function may be:

- a lazy specification statement with parameter lists of exactly the same types as the function, and a specified expression assignable to the declared type of the function, or
- an ordinary specification statement with a specified expression assignable to the callable type of the function.

```
Comparison order(String x, String y);
if (reverseOrder) {
    order(String x, String y) => y<=>x;
}
else {
    order(String x, String y) => x<=>y;
}
```

```
Comparison format(Integer x);
switch (base)
case (decimal) {
    format = (Integer i) => i.string;
}
case (binary) {
    format = formatBin;
}
case (hexadecimal) {
    format = formatHex;
}
```

Every forward-declared function must explicitly specify a type. It may not be declared using the keyword `function`.

A toplevel function may not be forward-declared. A method of an interface may not be forward-declared.

If a `shared` method is forward-declared, its implementation must be definitely specified by all conditional paths in the class initializer.

4.7.6. Functions with multiple parameter lists

A function may declare multiple lists of parameters. A function with more than one parameter list returns instances of `Callable` in `ceylon.language` when invoked. Every function with multiple parameter lists is exactly equivalent to a function with a single parameter list that returns an anonymous function.

This function declaration:

```
Boolean greaterThan<Element>(Element val)(Element element)
        given Element satisfies Comparable<Element> =>
                element>val;
```

is equivalent to the following:

```
Boolean(Element) greaterThan<Element>(Element val)
        given Element satisfies Comparable<Element> =>
                (Element element) => element>val;
```

For a function with `n` parameter lists, there are `n-1` inferred anonymous functions. The `i`th inferred function:

- has a callable type formed by eliminating the first `i` parameter lists of the original declared function,
- has the `i+1`th parameter list of the original declared function, and
- if `i<n`, returns the `i+1`th inferred function, or
- otherwise, if `i==n`, has the implementation of the original declared function.

Then the original function returns the first inferred anonymous function.

This method declaration:

```
function fullName(String firstName)(String middleName)(String lastName)
        => firstName + " " + middleName + " " + lastName;
```

Is equivalent to:

```
function fullName(String firstName) =>
        (String middleName) =>
                (String lastName) =>
                        firstName + " " + middleName + " " + lastName;
```

4.7.7. Formal and default methods

If a function declaration does not have a lazy specifier, or a block, and is annotated `shared`, and is a method of either:

- an interface, or
- a class annotated `abstract` or `formal`,

then the function declaration may be annotated `formal`, and is called a `formal` method, or, sometimes, an *abstract method*.

```
shared formal Item? get(Key key);
```

A method which is not annotated `formal` is called a *concrete* method.

If a concrete method is annotated `shared`, and is a member of a class or interface, then it may be annotated `default` and is

called a `default` method.

```
shared default void writeLine(String line) {
    write(line);
    write("\n");
}
```

A method annotated `formal` may not specify an implementation (a lazy specifier, or a block).

A method annotated `default` may specify an implementation (a lazy specifier, or a block), or may be forward-declared.

Every `formal` method must explicitly specify a type. It may not be declared using the keyword `function`.

A toplevel method may not be annotated `formal` or `default`.

An un-`shared` method may not be annotated `formal` or `default`.

4.7.8. Method refinement

Methods may be refined, just like in other object-oriented languages.

- A class or interface may refine any `formal` or `default` method it inherits, unless it inherits a non-`formal` non-`default` method that refines the method.
- A concrete class must refine every `formal` method it inherits, unless it inherits a non-`formal` method that refines the method.

A method of a subtype *refines* a method of a supertype if the method of the supertype is `shared` and the two methods have the same name. The first method is called the *refining* method, and the second method is called the *refined* method.

Then, given the refined realization of the method it refines, as defined in §3.7.6 Realizations, and, after substituting the type parameters of the refined method for the type parameters of the refining method in the schema of the refining method, the refining method must:

- have the same number of type parameters as the refined schema, and for each type parameter the intersection of its upper bounds must be a supertype of the intersection of the upper bounds of the corresponding type parameter of the realization,
- have the same number of parameter lists, with the same signatures, as the realization, and
- have a return type that is assignable to the return type of the realization, or
- if it has no return type, the refined method must also have no return type.

Note: in a future release of the language, we would like to support contravariant refinement of method parameter types.

Furthermore:

- the refining method must be annotated `actual`, and
- the refined method must be annotated `formal` or `default`.

If a method is annotated `actual`, it must refine some method defined by a supertype.

A method may not, directly or indirectly, refine two different methods not themselves annotated `actual`.

Then invocation of the method is polymorphic, and the actual method invoked depends upon the concrete type of the class instance.

```
shared abstract class AbstractSquareRooter() {
    shared formal Float squareRoot(Float x);
}
```

```
class ConcreteSquareRooter()
        extends AbstractSquareRooter() {
    shared actual Float squareRoot(Float x) => x^0.5;
```

```
}
```

Alternatively, a subtype may refine a method using a specification statement, as defined in §5.2.3 Specification statements. The specification statement must satisfy the requirements of §4.7.5 Forward declaration of functions above for specification of a forward-declared function.

```
class ConcreteSquareRooter()
        extends AbstractSquareRooter() {
    squareRoot(Float x) => x^0.5;
}
```

4.8. Values

There are two basic kinds of *value*:

- A *reference* defines state. It has a persistent value, determined at the moment it is specified or assigned.
- A *getter* defines how a value is evaluated. It is defined using a block or lazy specifier, which is executed every time the value is evaluated. A getter may have a matching *setter*.

If a value belongs to a type, it is called an *attribute*.

```
ValueDeclaration: Annotations ValueHeader (Block | (Specifier | LazySpecifier)? ";")
```

All value declarations specify the value name. A value declaration may specify a type.

```
ValueHeader: (Type | "value" | "dynamic") MemberName
```

A value may be declared using the keyword `dynamic` in place of its type. Such a value has no type.

Note: syntactically a value declaration looks like a function declaration with zero parameter lists. It is often helpful, in thinking about the syntax and semantics of Ceylon, to take the perspective that a value is a function with zero parameter lists, or, alternatively, that a function is a value of type `Callable`.

A value may be *variable*, in which case it may be freely assigned using the assignment and compound assignment operators defined in §6.8 Operators. This is the case for a reference annotated `variable`, or for a getter with a matching setter.

4.8.1. References

The lifecycle and scope of the persistent value of a reference depends upon where the reference declaration occurs:

- A toplevel reference represents global state associated with the lifecyle of a module, as defined by §8.2.10 Initialization of toplevel references.
- A reference declared directly inside the body of a class represents a persistent value associated with every instance of the class, as defined by §8.2.3 Current instance of a class or interface. Repeated evaluation of the attribute of a particular instance of the class produces the same result until the attribute of the instance is assigned a new value.
- A reference declared inside a block represents state associated with a frame, that is, with a particular execution of the containing block of code, as defined in §8.2.4 Current frame of a block.

The persistent value of a reference may be specified or initialized as part of the declaration of the reference, or via a later specification statement, as defined in §5.2.3 Specification statements, or assignment expression, as defined in §6.8 Operators, or, if it is a parameter, by an argument to an invocation expression, as defined in §6.6 Invocation expressions.

A reference annotated `variable` has a persistent value that can be assigned multiple times. A reference not annotated `variable` has a persistent value that can be specified exactly once and not subsequently modified.

```
variable Integer count = 0;
```

```
shared Decimal pi = calculatePi();
```

```
shared Integer[] evenDigits = [0,2,4,6,8];
```

A reference declaration may have a specifier which specifies its persistent value or, in the case of a variable reference, its initial persistent value. The type of the specified expression must be assignable to the type of the reference.

If the specified expression has no type, and the declaration occurs within a `dynamic` block, then the specification is not type-checked at compile time.

If a reference is a parameter, it must not specify a persistent value.

A reference belonging to a class may be annotated `late`, in which case the initializer of the class is not required to initialize its persistent value. Furthermore, a self-reference to an instance being initialized may be assigned to the reference. If the reference is evaluated before it is initialized, or before its value has been completely initialized, an exception is thrown.

If a class declares or inherits a `variable` reference, it must (directly or indirectly) extend the class `Basic` defined in `ceylon.language`.

4.8.2. Getters

A getter implementation may be a block.

```
shared Float total {
    variable Float sum = 0.0;
    for (li in lineItems) {
        sum += li.amount;
    }
    return sum;
}
```

Every conditional execution path of the block must end in a `return` directive that specifies an expression assignable to the type of the value, or in a `throw` directive, as specified in §5.2.4 Definite return.

Alternatively, a getter implementation may be a lazy specifier, that is, an expression specified using `=>`. The type of the specified expression must be assignable to the type of the value.

```
Name name => Name(firstName, initial, lastName);
```

4.8.3. Setters

A setter defines how the value of a getter is assigned.

```
SetterDeclaration: "assign" MemberName (Block | LazySpecifier)
```

The name specified in a setter declaration must be the name of a matching getter that directly occurs earlier in the body containing the setter declaration. If a getter has a setter, we say that the value is *variable*.

Within the body of the setter, a value reference to the getter evaluates to the value being assigned.

A setter implementation may be a block. The block may not contain a return directive that specifies an expression.

```
shared String name { return join(firstName, lastName); }
assign name { firstName=first(name); lastName=last(name); }
```

Alternatively, a setter implementation may be a lazy specifier. The specified expression must be a legal statement.

```
shared String name => join(n[0], n[1]);
assign name => n = [first(name), last(name)];
```

A setter may not be annotated `shared`, `default` or `actual`. The visibility and refinement modifiers of an attribute with a setter are specified by annotating the matching getter.

4.8.4. Value type inference

An un-`shared` value with a block, specifier, or lazy specifier may be declared using the keyword `value` in place of the ex-

plicit type declaration. Then the value's type is inferred:

- if the value is a reference with a specifier, then the type of the value is the type of the specified expression,
- if the value is a getter, and the getter implementation is a lazy specifier, then the type of the value is the type of the specified expression,
- if the value is a getter, and the getter implementation is a block, and the getter contains no `return` directive, then the type of the value is `Bottom` (this is the case where the getter always terminates in a `throw` directive), or
- otherwise, the type of the value is the union of all returned expression types of `return` directives of the getter body.

```
value names = List<String>();
```

```
variable value count = 0;
```

```
value name => Name(firstName, initial, lastName);
```

4.8.5. Forward declaration of values

The declaration of a reference may be separated from the specification or initialization of its persistent value. The declaration of a getter may be separated from the specification of its implementation. If a value declaration does not have a specifier, lazy specifier, or a block, and is not annotated `formal`, it is a *forward-declared* value.

A forward-declared value may later be specified using a specification statement, as defined in §5.2.3 Specification statements.

- The specification statement for a forward-declared getter is a lazy specification statement with no parameter list, and a specified expression assignable to the type of the value.
- The specification statement for a forward-declared reference is an ordinary specification statement with a specified expression assignable to the type of the value.

```
String greeting;
switch (language)
case (en) {
    greeting = "Hello";
}
case (es) {
    greeting = "Hola";
}
else {
    throw LanguageNotSupported();
}
print(greeting);
```

Every forward-declared value must explicitly specify a type. It may not be declared using the keyword `value`.

A toplevel value may not be forward-declared. An attribute of an interface may not be forward-declared.

A forward-declared getter may not have a setter.

If a `shared` value is forward-declared, its implementation must be definitely specified by all conditional paths in the class initializer.

4.8.6. Formal and default attributes

If a value declaration does not have a specifier, lazy specifier, or a block, and is annotated `shared`, and is a member of either:

- an interface, or
- a class annotated `abstract` or `formal`,

then the value declaration may be annotated `formal`, and is called a `formal` attribute, or, sometimes, an *abstract attribute*.

```
shared formal variable String firstName;
```

An attribute which is not annotated `formal` is called a *concrete* attribute.

If a concrete attribute is annotated `shared`, and is a member of a class or interface, then it may be annotated `default` and is called a `default` attribute.

```
shared default String greeting = "Hello";
```

An attribute annotated `formal` may not specify an implementation (a specifier, lazy specifier, or a block). Nor may there be a setter for a formal attribute.

An attribute annotated `default` may specify an implementation (a specifier, lazy specifier, or a block), or may be forward-declared.

Every `formal` attribute must explicitly specify a type. It may not be declared using the keyword `function`.

A toplevel attribute may not be annotated `formal` or `default`.

An un-`shared` attribute may not be annotated `formal` or `default`.

4.8.7. Attribute refinement

Ceylon allows attributes to be refined, just like methods. This helps eliminate the need for Java-style getter and setter methods.

- A class or interface may refine any `formal` or `default` attribute it inherits, unless it inherits a non-`formal` non-`default` attribute that refines the attribute.
- A concrete class must refine every `formal` attribute it inherits, unless it inherits a non-`formal` attribute that refines the attribute.

Any non-variable attribute may be refined by a reference or getter. A variable attribute may be refined by a `variable` refernce or by a getter and setter pair.

TODO: are you allowed to refine a getter or setter without also refining its matching setter or getter?

An attribute of a subtype *refines* an attribute of a supertype if the attribute of the supertype is `shared` and the two attributes have the same name. The first attribute is called the *refining* attribute, and the second attribute is called the *refined* attribute.

Then, given the refined realization of the attribute it refines, as defined in §3.7.6 Realizations, the refining attribute must:

- be variable, if the attribute it refines is variable, and
- have *exactly the same type* as the realization, if the attribute it refines is variable,
- have a type that is assignable to the type of the refined schema, if the attribute it refines is not variable, or
- if it has no type, the refined attribute must also have no type.

Furthermore:

- the refining attribute must be annotated `actual`, and
- the refined attribute must be annotated `formal` or `default`.

If an attribute is annotated `actual`, it must refine some attribute defined by a supertype.

An attribute may not, directly or indirectly, refine two different attributes not themselves annotated `actual`.

A non-variable attribute may be refined by a variable attribute.

TODO: Is that really allowed? It could break the superclass. Should we say that you are allowed to do it when you refine an interface attribute, but not when you refine a superclass attribute?

Then evaluation and assignment of the attribute is polymorphic, and the actual attribute evaluated or assigned depends upon the concrete type of the class instance.

```
shared abstract class AbstractPi() {
    shared formal Float pi;
}
```

```
class ConcretePi()
        extends AbstractPi() {
    shared actual Float pi = calculatePi();
}
```

Alternatively, a subtype may refine an attribute using a specification statement, as defined in §5.2.3 Specification statements. The specification statement must satisfy the requirements of §4.8.5 Forward declaration of values above for specification of a forward-declared attribute.

```
class ConcretePi()
        extends AbstractPi() {
    pi = calculatePi();
}
```

Chapter 5. Statements, blocks, and control structures

Function, value, and class bodies contain procedural code that is executed when the function is invoked, the value evaluated, or the class instantiated. The code contains expressions and control directives and is organized using blocks and control structures.

Note: the Ceylon language has a recursive block structure—statements and declarations that are syntactically valid in the body of a toplevel declaration are, in general, also syntactically valid in the body of a nested declaration or of a control structure, and vice-versa.

5.1. Block structure and references

A *body* is a block, defined in §5.2 Blocks and statements, class body, defined in §4.5 Classes, interface body, defined in §4.4 Interfaces, or comprehension clause, defined in §6.6.6 Comprehensions. Every body (except for a comprehension clause) is list of semicolon-delimited statements, control structures, and declarations, surrounded by braces. Some bodies end in a control directive. Every program element in the list is said to *directly occur* in the body. A program element *directly occurs earlier* than a second program element if both program elements directly occur in a body and the first program element occurs (lexically) earlier in the list than the second program element.

A program element *(indirectly) occurs* in a body if:

- the program element directly occurs in the body, or
- the program element indirectly occurs inside the body of a declaration or control structure that occurs directly in the body.

We sometimes say that the body *contains* the program element if the program element (indirectly) occurs in the body.

A program element *(indirectly) occurs earlier* than a second program element if:

- the two program elements both directly occur in the same body, and the second program element occurs after the first program element, or
- the second program element indirectly occurs inside the body of a declaration or control structure, and the first program element directly occurs earlier than the declaration or control structure.

Then we also say that the second program element *(indirectly) occurs later* than the first. The set of program elements that occur later than a program element is sometimes called the *lexical scope* of the program element.

A program element *sequentially occurs* in a body if:

- the program element directly occurs in the body, or
- the program element sequentially occurs inside the body of a control structure that occurs directly in the body.

A program element *sequentially occurs earlier* than a second program element if:

- the two program elements both directly occur in the same body, and the second program element occurs after the first program element, or
- the second program element sequentially occurs inside the body of a control structure, and the first program element directly occurs earlier than the declaration or control structure.

If a program element sequentially occurs earlier than a second program element, the *sequence of statements* from the first program element to the second program element comprises:

- the sequence of statements that occur directly in the body in which the first program element directly occurs, beginning from the first program element and ending with the second program element, if the second program element occurs directly in the same body as the first program element, or
- the sequence of statements that occur directly in the body in which the first program element directly occurs, beginning

from the first program element and ending with the declaration or control structure in whose body the second program element sequentially occurs, followed by the sequence of statements from the first statement of the declaration whose body contains the second program element to the second program element itself, otherwise.

5.1.1. Declaration name uniqueness

A program element is contained within the *namespace* of a declaration if either:

- the declaration is a toplevel declaration, and the program element is a toplevel declaration of the same package,
- the declaration directly occurs in a body, and the program element sequentially occurs in the same body,
- the declaration is a parameter or type parameter, and the program element sequentially occurs in the body of the parameterized declaration, or
- the program element is a control structure variable or iteration variable of a control structure that sequentially occurs in the namespace of the declaration.

The namespace of a declaration may not contain a second declaration with the same name. For example, the following is illegal:

```
function fun(Float number) {
    if (number<0.0) {
        Float number = 1.0; //error
        ...
    }
    ...
}
```

A class or interface may not inherit a declaration with the same name as a declaration it contains unless either:

- the contained declaration directly or indirectly refines the inherited declaration,
- the contained declaration is not `shared`, or
- the inherited declaration is not `shared`.

A class or interface may not inherit two declarations with the same name unless either:

- both of the inherited declarations are `formal` and directly or indirectly refine some member of a common supertype,
- the class or interface contains a declaration that directly or indirectly refines both the inherited declarations (in which case both the inherited declarations directly or indirectly refine some member of a common supertype),
- one of the inherited declarations directly or indirectly refines the other inherited declaration, or
- at least one of the inherited declarations is not `shared`.

5.1.2. Scope of a declaration

The scope of a declaration is governed by the body or package in which it occurs. A declaration is *in scope* at a program element if and only if either:

- the declaration is a parameter or type parameter of a declaration whose body contains the program element,
- the declaration is a control structure variable or iteration variable belonging to a block of a control structure that contains the program element,
- the program element belongs to or is contained in the body of the declaration itself,
- the program element belongs to or is contained in the body of a class or interface which inherits the declaration,
- the declaration directly occurs in a body containing the program element,

- the declaration is imported by the compilation unit containing the program element and is visible to the program element, or
- the declaration is a toplevel declaration in the package containing the program element.

Where:

- A control structure variable or iteration variable *belongs* to a block of a control structure if the block immediately follows the declaration of the variable.
- A program element *belongs* to a declaration if it occurs in the `extends`, `satisfies`, `of`, or `given` clause of the declaration.

Furthermore:

- A condition variable of a condition belonging to a condition list is in scope in any condition of the same condition list that occurs lexically later.
- A resource expression variable of a `try` statement is in scope in any resource expression of the same resource expression list that occurs lexically later.
- An iteration variable or condition variable of a comprehension is in scope in any clause of the comprehension that occurs lexically later, since comprehension clauses are viewed as nested bodies.

And finally, there are special rules for annotation lists, defined in §7.1.1 Annotation lists:

- An annotation argument list belongs to the annotated declaration.
- An annotation name is considered to occur directly in the compilation unit containing the program element.

Note: if no reference to an un-`shared` declaration occurs within the scope of the declaration, a compiler warning is produced.

5.1.3. Visibility

Classes, interfaces, functions, values, aliases, and type parameters have names. Occurrence of a name in code implies a hard dependency from the code in which the name occurs to the schema of the named declaration. We say that a class, interface, value, function, alias, or type parameter is *visible* to a certain program element if its name may occur in the code that belongs to that program element.

The visibility of a declaration depends upon where it occurs, and upon whether it is annotated `shared`. A toplevel or member declaration may be annotated `shared`:

- If a toplevel declaration is annotated `shared`, it is visible wherever the package that contains it is visible. Otherwise, a toplevel declaration is visible only to code in the package containing its compilation unit.
- If a member declaration is annotated `shared`, it is visible wherever the class or interface that contains it is visible. Otherwise, a declaration that occurs directly inside a class or interface body is visible only inside the class or interface declaration.

A type parameter or a declaration that occurs directly inside a block (the body of a function, getter, setter, or control structure) may not be annotated `shared`.

- A type parameter is visible only inside the declaration to which it belongs.
- A declaration that occurs directly inside a block is visible only inside the block.

TODO: Should we allow you to limit the effect of the `shared` annotation by specifying a containing program element or package?

We say that a type is *visible* to a certain program element if it is formed from references to classes, interfaces, type parameters, and type aliases whose declarations are visible to the program element. For `shared` declarations:

- The type of a value must be visible everywhere the value itself is visible.
- The return type of a function must be visible everywhere the function itself is visible.
- The satisfied interfaces of a class or interface must be visible everywhere the class or interface itself is visible.
- The superclass of a class must be visible everywhere the class itself is visible.
- The aliased type of a class alias, interface alias, or type alias must be visible everywhere the alias itself is visible.

5.1.4. Hidden declarations

If two declarations with the same name or imported name, as defined in §4.2.5 Imported name, are both in scope at a certain program element, then one declaration may *hide* the other declaration.

- If an inner body is contained (directly or indirectly) in an outer body, a declaration that is in scope in the inner body but is not in scope in the outer body hides a declaration that is in scope in the outer body. (In particular, a declaration inherited by a nested class or interface hides a declaration of the containing body.)
- An un-`shared` declaration occurring directly in the body of a class containing the program element hides a declaration inherited by the class.
- An `actual` declaration hides the declaration it refines.
- A declaration occurring in a body containing the program element hides a declaration imported by the compilation unit containing the body or implicitly imported from the module `ceylon.language`.
- A toplevel declaration of the package containing the program element hides a declaration implicitly imported from the module `ceylon.language`.
- A declaration explicitly imported by the compilation unit containing the program element hides a declaration implicitly imported from the module `ceylon.language`.
- A declaration explicitly imported by the compilation unit containing the program element hides a toplevel declaration of the package containing the compilation unit.
- A declaration explicity imported by name in the compilation unit containing the program element hides a declaration explicitly imported by wildcard in the compilation unit.

For example, the following code is legal:

```
class Person(name) {
    String name;
    shared String lowerCaseName {
        String name = this.name.lowercased;
        return name;
    }
}
```

As is this code:

```
class Point(x, y) {
    shared Float x;
    shared Float y;
}

class Complex(Float x, Float y=0.0)
        extends Point(x, y) {}
```

When a member of a class is hidden by a nested declaration, the member may be accessed via the self reference `this`, defined in §6.3.1 this, or via the outer instance reference `outer`, defined in §6.3.2 outer.

```
shared class Item(name) {
    variable String name;
    shared void changeName(String name) {
        this.name = name;
    }
}
```

```
class Catalog(name) {
    shared String name;
    class Schema(name) {
        shared String name;
        Catalog catalog => outer;
        String catalogName => outer.name;
        class Table(name) {
            shared String name;
            Schema schema => outer;
            String schemaName => outer.name;
            String catalogName => catalog.name;
        }
    }
}
```

When a toplevel declaration of a package is hidden by another declaration, the toplevel declaration may be accessed via the containing package reference `package`, defined in §6.3.4 package.

```
Integer n => 0;
Integer f(Integer n) => n+package.n;
```

5.1.5. References and block structure

A declaration may be in scope at a program element, but not *referenceable* at the program element. A declaration is referenceable at a program element if the declaration is in scope at the program element and either:

- the program element occurs within the lexical scope of the declaration, or
- the declaration does not directly occur in a block or in the initializer section of a class body.

Note that these rules have very different consequences for:

- a declaration that occurs in a block, as specified in §5.2 Blocks and statements, or in an class initializer section, as specified in §4.5.2 Initializer section, and
- a toplevel declaration, as specified in §4.1.1 Toplevel and nested declarations, or a declaration that occurs in a class declaration sectiony, as specified in §4.5.3 Declaration section, or interface body, as specified in §4.4.1 Interface bodies.

Declarations that occurs in a block or class initializer section are interspersed with procedural code that initializes references. Therefore, a program element in a block or initializer may not refer to a declaration that occurs later in the block or class body. This restriction does not apply to declarations that occur in an interface body or class declaration section. Nor does it apply to toplevel declarations, which are not considered to have a well-defined order.

The following toplevel function declarations, belonging to the same package, are legal:

```
Float x => y;
```

```
Float y => x;
```

This code is not legal, since the body of a function is an ordinary block:

```
Float->Float xy() {
    Float x => y;  //compiler error: y is not referenceable
    Float y => x;
    return x->y;
}
```

This code is not legal, since all three statements occur in the initializer section of the class body:

```
class Point() {
    Float x => y;  //compiler error: y is not referenceable
    Float y => x;
    Float->Float xy = x->y;
}
```

However, this code *is* legal, since the statements occur in the declaration section of the class body:

```
class Point() {
```

```
    Float x => y;
    Float y => x;
}
```

Likewise, this code is legal, since the statements occur in an interface body:

```
interface Point {
    Float x => y;
    Float y => x;
}
```

5.1.6. Type inference and block structure

A value declared using the keyword `value` or a function declared using the keyword `function` may be in scope at a program element, but its type may not be *inferrable*, as defined by §3.2.9 Type inference, from the point of view of that program element.

The type of a value or function declared using the keyword `value` or `function` is inferrable to a program element if the declaration is in scope at the program element and the program element occurs within the lexical scope of the declaration.

Note: the type of a value or function declared using the keyword `value` or `function` is not inferrable within the body of the value or function itself.

For any other declaration, including any declaration which explicitly specifies its type, the type is considered inferrable to a program element if the declaration is in scope at the program element.

The following code is not legal:

```
interface Point {
    value x => y;  //compiler error: type of y is not inferrable
    value y => x;
}
```

However, this code is legal:

```
interface Point {
    value x => y;
    Float y => x;
}
```

5.1.7. Unqualified reference resolution

An *unqualified reference* is:

- the type name in an unqualified type declaration or type argument, as defined by §3.2.7 Type expressions, for example `String` and `Sequence` in `Sequence<String>`,
- the value, function, or type name in a base expression, as defined by §6.5.1 Base expressions, for example `counter` in `counter.count`, `entries` and `people` in `entries(people*.name)`, or `Entry`, `name`, and `item` in `Entry(name,item)`, or
- the type name in an unqualified type in a static expression, as defined by §6.5.5 Static expressions, for example `Sequence` in `Sequence.iterator`.

If a program element contains an unqualified reference:

- there must be at least one declaration with the given name or imported name, as defined in §4.2.5 Imported name, in scope at the program element, and
- if multiple declarations with the given name or imported name are in scope at the program element where the given name occurs, then it is guaranteed by the type system and §5.1.1 Declaration name uniqueness that there is exactly one such declaration which is not hidden by any other declaration.

Then the reference is to this unique unhidden declaration, and:

- the declaration must be referenceable at the program element,
- the type of the declaration must be inferrable to the program element, and
- if the declaration is forward-declared, it must be definitely initialized at the program element.

As a special exception to the above, if there is no declaration with the given name or imported name in scope at the program element and the program element occurs inside a `dynamic` block, then the unqualified reference does not refer to any statically typed declaration.

If an unqualified reference refers to a member declaration of a type, then there is a unique *inheriting or declaring class or interface* for the unqualified reference, that is, the unique class or interface in whose body the unqualified reference occurs, and which declares or inherits the member declaration, and for which the member is not hidden at the program element where the unqualified reference occurs.

5.1.8. Qualified reference resolution

A *qualified reference* is:

- the type name in a qualified type declaration or type argument, as defined by §3.2.7 Type expressions, for example `Buffer` in `BufferedReader.Buffer`,
- the value, function, or type name in a member expression, as defined by §6.5.2 Member expressions, for example `count` in `counter.count`, `split` in `text.split()`, or `Buffer` in `br.Buffer()`,
- the type name in a qualified type in a static expression, as defined by §6.5.5 Static expressions, for example `Buffer` in `BufferedReader.Buffer.size`, or the member name in a static expression, for example `iterator` in `Sequence.iterator`, or `size` in `BufferedReader.Buffer.size`.

Every qualified reference has a qualifying type:

- For a type declaration, the qualifying type is the full qualified type the qualifies the type name.
- For a value reference or callable reference, the qualifying type is the type of the receiver expression.
- For a static reference, the qualifying type is the full qualified type the qualifies the type or member name.

A qualified reference may not have `Nothing` as the qualifying type.

If a program element contains a qualified reference:

- the qualifying type must have or inherit at least one member or nested type with the given name or imported name, as defined in §4.2.5 Imported name, which is visible at the program element, and
- if there are multiple visible members with the given name or imported name, then it is guaranteed by the type system and §5.1.1 Declaration name uniqueness that there is exactly one such member which is not refined by another member, except
- if the qualifying type inherits a class or interface that contains the program element, and an un-`shared` declaration contained directly in the body of this class or interface has the same name as a `shared` member of the qualifying type, in which case the un-`shared` declaration hides the `shared` member, or
- if the qualifying type is an intersection type, in which case there may be multiple members which are not refined by another member, but where there is exactly one such member that is refined by each of these members, but is not refined by another member that is refined by all of these members, except
- in the case of certain pathological intersection types, where two of the intersected types declare distinct members with the same name, that do not refine any member of a common supertype (in which case what we actually have are disjoint types that are nevertheless not considered provably disjoint within the rules of the typesystem), and in this case the qualified reference is considered illegal.

Then the reference is to the unique member or nested class. If the program element is contained in the body of a class or interface, and the member declaration directly occurs in the body of the class or interface, and the qualified reference is a

value reference or callable reference, and the receiver expression is a self reference to the instance being initialized, then:

- the member declaration must be referenceable at the program element,
- the type of the member must be inferrable to the program element, and
- if the member declaration is forward-declared, it must be definitely initialized at the program element.

As a special exception to the above, if the program element occurs inside a `dynamic` block, and the the receiver expression has no type, then the qualified reference does not refer to any statically typed declaration.

5.2. Blocks and statements

A *block* is list of semicolon-delimited statements, control structures, and declarations, surrounded by braces.

```
Block: "{" (Declaration | Statement)* "}"
```

A *statement* is an assignment or specification, an invocation of a method, an instantiation of a class, a control structure, a control directive, or an assertion.

```
Statement: ExpressionStatement | Specification | Assertion | DirectiveStatement | ControlStructure
```

A statement or declaration contained in a block may not evaluate a value, invoke a function, instantiate a class, or extend a class whose declaration occurs later in the block.

5.2.1. Expression statements

Only certain expressions are valid statements:

- assignment,
- prefix or postfix increment or decrement,
- invocation of a method,
- instantiation of a class.

```
ExpressionStatement: ( Assignment | IncrementOrDecrement | Invocation ) ";"
```

For example:

```
x += 1;
```

```
x++;
```

```
print("Hello");
```

```
Main(process.arguments);
```

5.2.2. Control directives

A *control directive* statement ends execution of the current block and forces the flow of execution to resume in some outer scope. They may only occur as the lexically last statement of a block.

```
DirectiveStatement: Directive ";"
```

There are four control directives:

- the `return` directive—to return a value from a getter or non-`void` function or terminate execution of a setter, class initializer, or `void` method,

- the `break` directive—to terminate a loop,
- the `continue` directive—to jump to the next iteration of a loop, and
- the `throw` directive—to raise an exception.

```
Directive: Return | Throw | Break | Continue
```

For example:

```
throw Exception();
```

```
return x+y;
```

```
break;
```

```
continue;
```

The `return` directive must sequentially occur in the body of a function, getter, setter, or class initializer. In the case of a setter, class initializer, or `void` function, no expression may be specified. In the case of a getter or non-`void` function, an expression must be specified. The expression type must be assignable to the return type of the function or the type of the value. When the directive is executed, the expression is evaluated to determine the return value of the function or getter.

```
Return: "return" Expression?
```

If the specified expression has no type, or if the function or getter has no type, and the directive occurs within a `dynamic` block, then the directive is not type-checked at compile time.

Note: a `return` statement returns only from the innermost function, getter, setter, or class initializer, even in the case of a nested or anonymous function. There are no "non-local returns" in the language.

The `break` directive must sequentially occur in the body of a loop.

```
Break: "break"
```

The `continue` directive must sequentially occur in the body of a loop.

```
Continue: "continue"
```

A `throw` directive may appear anywhere and may specify an expression, whose type must be a subtype of type `Exception` defined in `ceylon.language`. When the directive is executed, the expression is evaluated and the resulting exception is thrown. If no expression is specified, the directive is equivalent to `throw Exception()`.

```
Throw: "throw" Expression?
```

If the specified expression has no type, and the directive occurs within a `dynamic` block, then the directive is not type-checked at compile time.

5.2.3. Specification statements

A *specification* statement may specify or initialize the persistent value of a forward-declared reference, or specify the implementation of a forward-declared getter or function.

```
Specification: ValueSpecification | LazySpecification
```

The persistent value of a forward-declared reference or the implementation of a forward-declared function may be specified by a *value specification statement*. The value specification statement consists of an unqualified value reference and an ordinary `=` specifier. The value reference must refer to a declaration which sequentially occurs earlier in the body in which the specification statement occurs.

```
ValueSpecification: MemberName Specifier ";"
```

The type of the specified expression must be assignable to the type of the reference, or to the callable type of the function.

If the specified expression has no type, or if the reference or function has no type, and the specification occurs within a `dynamic` block, then the specification is not type-checked at compile time.

```
String greeting;
if (exists name) {
    greeting = "hello ``name``";
}
else {
    greeting = "hello world";
}
```

```
String process(String input);
if (normalize) {
    process = String.normalized;
}
else {
    process = (String s) => s;
}
```

Note: there is an apparent ambiguity here. Is the statement `x=1;` a value specification statement, or an assignment expression statement? The language resolves this ambiguity by favoring the interpretation as a specification statement whenever that interpretation is viable. This is a transparent solution, since it accepts strictly more code than the alternative interpretation, and for ambiguous cases the actual semantics are identical between the two interpretations.

The implementation of forward-declared getter or function may be specified using a *lazy specification statement*. The specification statement consists of either:

- an unqualified value reference and a lazy `=>` specifier, or
- a unqualified callable reference, one or more parameter lists, and a lazy specifier.

The value reference or callable reference must refer to a declaration which sequentially occurs earlier in the body in which the specification statement occurs.

A callable reference followed by a parameter list is itself considered a callable reference, called a *parameterized reference*. If the parameter list has type `P` then the callable reference must have the exact type `Callable<R,P>` for some type `R`. Then the type of the parameterized reference is `R`.

```
ParameterizedReference: MemberName Parameters+
```

Thus, the specification statement consists of a parameterized reference followed by a lazy specifier.

```
LazySpecification: (MemberName | ParameterizedReference) LazySpecifier ";"
```

The type of the specified expression must be assignable to the type of the parameterized reference, or to the type of the value reference.

```
String greeting;
if (exists name) {
    greeting => "hello ``name``";
}
else {
    greeting => "hello world";
}
```

```
String process(String input);
if (normalize) {
    process(String input) => input.normalized;
}
else {
    process(String s) => s;
}
```

5.2.4. Definite return

A sequence of statements may *definitely return.*

- A sequence of statements definitely returns if it ends in a `return` or `throw` directive, or in a control structure that definitely returns, or contains an assertion with a condition list that is never satisfied.
- A body definitely returns if it contains a list of statements that definitely returns.
- An `if` conditional definitely returns if it has an `else` block and both the `if` and `else` blocks definitely return, or if its condition list is always satisfied and the `if` block definitely returns, or if its condition list is never satisfied and it has an `else` block that definitely returns.
- A `switch` conditional definitely returns if all `case` blocks definitely return and the `else` block, if any, definitely returns.
- A `for` loop definitely returns if it has an `else` block that definitely returns, and there is no `break` directive in the `for` block, or if the iterated expression type is a nonempty type, and the `for` block definitely returns.
- A `while` loop definitely returns if its condition list is always satisfied and the `while` block definitely returns.
- A `try/catch` exception manager definitely returns if the `try` block definitely returns and all `catch` blocks definitely return or if the `finally` block definitely returns.

The body of a non-`void` method or getter must definitely return.

A body may not contain an additional statement, control structure, or declaration following a sequence of statements that definitely returns. Such a statement, control structure, or declaration is considered *unreachable*.

5.2.5. Definite initialization

A sequence of statements may *definitely initialize* a forward-declared declaration.

- A sequence of statements definitely initializes a declaration if one of the statements is a specification statement or assigment expression for the declaration or a control structure that definitely initializes the declaration, or if the sequence of statements ends in a `return` or `throw` directive, or contains an assertion with a condition list that is never satisfied.
- An `if` conditional definitely initializes a declaration if it has an `else` block and both the `if` and `else` blocks definitely initialize the declaration, of if its condition list is always satisfied and the `if` block definitely initializes the declaration, of if its condition list is never satisfied and it has an `else` block that definitely initializes the declaration.
- A `switch` conditional definitely initializes a declaration if all `case` blocks definitely initialize the declaration and the `else` block, if any, definitely initializes the declaration.
- A `for` loop definitely initializes a declaration if it has an `else` block that definitely initializes the declaration, and there is no `break` directive in the `for` block, or if the iterated expression type is a nonempty type, and the `for` block definitely initializes the declaration.
- A `while` loop definitely initializes a declaration if its condition list is always satisfied and the `while` block definitely initializes the declaration.
- A `try/catch` exception manager definitely initializes a declaration if the `try` block definitely initializes the declaration and all `catch` blocks definitely initialize the declaration or if the `finally` block definitely initializes the declaration.

A forward-declared declaration is considered *definitely initialized* at a certain statement or declaration if its declaration has a specifier, if it is referenced by a parameter, or if it is definitely initialized by the sequence of statements from its declaration to the given statement or declaration.

A forward-declared declaration must be definitely initialized wherever any value reference or callable reference to it occurs as an expression within the body in which it is declared.

A `shared` forward-declared declaration belonging to a class and not annotated `late` must be definitely initialized:

- at every `return` statement of the initializer of the containing class, and
- at the very last expression statement, directive statement or specification statement of the initializer of the containing class.

A specification statement for a method or non-`variable` reference, getter, or function may not (indirectly) occur in a `for` or `while` block unless the declaration itself occurs within the same `for` or `while` block.

TODO: Furthermore, the typechecker does some tricky analysis to determine that code like the following can be accepted:

```
Boolean minors;
for (p in people) {
    if (p.age<18) {
        minors = true;
        break;
    }
}
else {
    minors = false;
}
```

5.2.6. Definite uninitialization

A sequence of statements may *possibly initialize* a forward-declared declaration.

- A sequence of statements possibly initializes a declaration if one of the statements is a specification statement for the declaration or a control structure that possibly initializes the declaration.
- An `if` conditional possibly initializes a declaration if either the `if` block possibly initializes the declaration and the condition list is not never satisfied, or if the `else` block, if any, possibly initializes the declaration and the condition list is not always satisfied.
- A `switch` conditional possibly initializes a declaration if one of the `case` blocks possibly initializes the declaration or the `else` block, if any, possibly initializes the declaration.
- A `for` loop possibly initializes a declaration if the `for` block possibly initializes the declaration or if it has an `else` block that possibly initializes the declaration.
- A `while` loop possibly initializes a declaration if the `while` block possibly initializes the declaration and the condition list is not never satisfied.
- A `try/catch` exception manager possibly initializes a declaration if the `try` block possibly initializes the declaration, if one of the `catch` blocks possibly initializes the declaration, or if the `finally` block possibly initializes the declaration.

A forward-declared declaration is considered *definitely uninitialized* at a certain statement or declaration if:

- it is not possibly initialized by the sequence of statements from its declaration to the given statement or declaration,
- the statement does not (indirectly) occur in the `for` block or `else` block of a `for` loop with a `for` block that possibly initializes it,
- the statement does not (indirectly) occur in the `while` block of a `while` loop with a `while` block that possibly initializes it,
- the statement does not (indirectly) occur in a `catch` block of a `try/catch` exception manager with a `try` block that possibly initializes it, and
- the statement does not (indirectly) occur in the `finally` block of a `try/catch` exception manager with a `try` block or `catch` block that possibly initializes it.

A method or non-`variable` local or simple attribute must be definitely uninitialized wherever any value reference or callable reference to it occurs as a specification statement within the body in which it is declared.

5.3. Control structures and assertions

Control of execution flow may be achieved using control directives and *control structures*. Control structures include conditionals, loops, and exception management.

Ceylon provides the following control structures:

- the `if/else` conditional—for controlling execution based on a boolean condition, type condition, or check for a non-null or non-empty value,
- the `switch/case/else` conditional—for controlling execution using an enumerated list of values or types,
- the `while` loop—for loops which terminate based on a boolean condition, type condition, or check for a non-null or non-empty value,
- the `for/else` loop—for looping over elements of an iterable object, and
- the `try/catch/finally` exception manager—for managing exceptions and controlling the lifecycle of objects which require explicit destruction.

```
ControlStructure: IfElse | SwitchCaseElse | While | ForFail | TryCatchFinally | Dynamic
```

Control structures are not considered to be expressions, and therefore do not evaluate to a value. However, comprehensions—and conditional expressions, planned for a future release of the language—are part of the expression syntax and share much of the syntax and semantics of the control structures they resemble.

Assertions are runtime checks upon program invariants, or function preconditions and postconditions. An assertion failure represents a bug in the program, and is not considered recoverable. Therefore, assertions should not be used to control "normal" execution flow.

Note: of course, in certain circumstances, it is appropriate to handle the exception that results from an assertion failure, for example, to display a message to the user, or in a testing framework to aggregate and report the failures that occurred in test assertions. A test failure may be considered "normal" occurrence from the point of view of a testing framework, but it's not "normal" in the sense intended above.

5.3.1. Control structure variables

Assertions and some control structures allow inline declaration of a *variable*. A variable is a reference, as defined by §4.8.1 References.

```
TypedVariable: Type MemberName
```

In most cases, the explicit type be omitted.

```
Variable: Type? MemberName
```

If the type is missing from the declaration, the type of the variable is inferred, according to rules that depend upon the control structure to which the variable belongs.

A variable declared by an assertion is a reference scoped to the body in which the `assert` statement occurs.

A variable declared by a control structure is a reference scoped to the block that immediately follows the variable declaration:

- For a variable in an `if` condition, the scope of the variable is the `if` block.
- For a variable in a `while` condition, the scope of the variable is the `while` block.
- For a variable in a `for` iterator, the scope of the variable is the `for` block.
- For a variable in a `try` clause, the scope of the variable is the `try` block.
- For a variable in a `catch` clause, the scope of the variable is the `catch` block.
- For a variable in an `assert` statement, the scope of the variable is the body containing the `assert` statement.

5.3.2. Iteration variables

A `for` loop requires an *iteration variable* declaration. An iteration variable is a reference scoped to the body of the loop.

```
IteratorVariable: Variable | EntryVariablePair
```

An iteration variable of type `Entry` may be specified in destructured form.

```
EntryVariablePair: Variable "->" Variable
```

If the type is missing from the declaration, the type of the iteration variable is inferred:

- given an iterated expression which has the principal instantiation `Iterable<X>`, the inferred type of the variable is `X`, unless
- the destructured form is used for an iterated expression which is has the principal instantiation `Iterable<Entry<X,Y>>`, in which case the inferred type of the first variable is `X`, and the inferred type of the second variable is `Y`.

TODO: Should we, purely for consistency, let you write `for (f(Float x) in functions)`, even though it's not very useful?

5.3.3. Control structure conditions

Some control structures expect conditions. There are four kinds of condition:

- a *boolean condition* is satisfied when a boolean expression evaluates to `true`,
- an *assignabilty condition* is satisfied when an expression evaluates to an instance of a specified type,
- an *existence condition* is satisfied when an expression evaluates to a non-null value, and
- a *nonemptiness condition* is satisfied when an expression evaluates to a non-null, non-empty value.

```
Condition: BooleanCondition | IsCondition | ExistsOrNonemptyCondition
```

A condition list has one or more conditions.

```
ConditionList: "(" Condition ("," Condition) ")"
```

A condition in the list may refer to a condition variable defined earlier in the list.

A condition list is considered to be *always satisfied* if every condition in the list is always satisfied. A condition list is considered to be *never satisfied* if some condition in the list is never satisfied.

TODO: are we going to support `satisfies` conditions on type parameters, for example, `if (Element satisfies Object)`, to allow refinement of its upper bounds?

5.3.4. Boolean conditions

A boolean condition is just an expression.

```
BooleanCondition: Expression
```

The expression must be of type `Boolean`.

A boolean condition is considered to be *always satisfied* if it is a value reference to `true`. A boolean condition is considered to be *never satisfied* if it is a value reference to `false`.

TODO: Should we do some more sophisticated static analysis to determine if a condition is always/never satisfied?

5.3.5. Assignability, existence, and nonemptiness conditions

An assignability, existence, or nonemptiness condition may contain either:

- an unqualified value reference to a non-`variable`, non-`default` reference, or

- an inline variable declaration together with an expression.

In the case of an assignability or existence condition, the type of the variable may be inferred.

```
IsCondition: "!"? "is" (TypedVariable Specifier | Type MemberName)
```

```
ExistsOrNonemptyCondition: ("exists" | "nonempty") (Variable Specifier | MemberName)
```

TODO: are we going to allow `is Type this` and `is Type outer` to narrow the type of a self reference?

The type of the value reference or expression must be:

- in the case of an assignability condition, a type which is not a subtype of the specified type, but whose intersection with the specified type is not exactly `Nothing`, except
- in the case of a *negated assignability condition* with `!is`, a type whose intersection with the specified type is not exactly `Nothing`, and which is not a supertype of the specified type, or
- in the case of an exists condition, a type whose intersection with `Null` is not exactly `Nothing` and whose intersection with `Object` is not exactly `Nothing`, or
- in the case of a nonemptiness condition, a subtype of `Anything[]?` whose intersection with `[]` is not exactly `Nothing`, and whose intersection with `[Nothing+]` is not exactly `Nothing`.

Note: an assignability condition may narrow to an intersection or union type.

```
if (is Printable&Identifiable obj) { ... }
```

```
if (is Integer|Float num) { ... }
```

Every existence or nonemptiness condition is equivalent to—and may be considered an abbreviation of—an assignability condition:

- `exists x` is equivalent to `is Object x`, and
- `nonempty x` is equivalent to `is [E+] x` where `x` is an expression whose type has the principal instantiation `E[]?`.

For an `is` assignability condition:

- if the condition contains a value reference, the value will be treated by the compiler as having type `T&X` where the conditional expression is of type `T` and `X` is the specified type, inside the block that immediately follows the condition, unless
- it is a *negated assignability condition* with `!is`, in which case the value will be treated by the compiler as having type `T~X`.

Where, for any given types `T` and `X`, the type `T~X` is determined as follows:

- if `X` covers `T`, as defined by §3.4.1 Coverage, then `T~X` is `Nothing`,
- if `T` is an intersection type, then `T~X` is the intersection of all `U~X` for every type `U` in the intersection,
- if `T` is a union type, then `T~X` is the union of all `U~X` for every type `U` in the union,
- if `T` is an enumerated type or an instantiation of a generic enumerated type, then `T~X` is the union of all `C~X` for every case `C` of `T`, or,
- otherwise, `T~X` is `T`.

If you prefer, you can think of the following:

```
Transaction tx = ...
if (is Usable tx) { ... }
```

As an abbreviation of:

```
if (is Transaction&Usable tx = tx) { ... }
```

Where the `tx` declared by the condition hides the outer declaration of `tx` inside the block that follows.

For an `exists` existence condition:

- if the condition declares a variable, the declared type of the variable must be a supertype of `T&Object`, where the specifier expression is of type `T`, or
- if the condition contains a value reference, the value will be treated by the compiler as having type `T&Object` inside the block that immediately follows the condition, where the conditional expression is of type `T`.

For a `nonempty` nonemptiness condition:

- if the condition declares a variable, the declared type of the variable must be a supertype of `T&[E+]`, where the specifier expression is of type `T` and `T` has the principal instantiation `E[]?`, or
- if the condition contains a value reference, the value will be treated by the compiler as having type `T&[E+]` inside the block that immediately follows the condition, where the conditional expression is of type `T` and `T` has the principal instantiation `E[]?`.

If you prefer, you can think of the following:

```
if (exists name) { ... }
```

As an abbreviation of:

```
if (exists String name = name) { ... }
```

Where the `name` declared by the condition hides the outer declaration of `name` inside the block that follows.

As a special exception to the above, if a condition occurs in a `dynamic` block, and conditional expression has no type, and the condition contains a value reference, then:

- the value will be treated by the compiler as having type `X` where `X` is the specified type, inside the block that immediately follows the condition, unless
- it is a negated assignability condition `!is`, an existence condition `exists`, or a nonempty condition `nonempty`, in which case the value will be treated by the compiler as having no type.

5.3.6. `if/else`

The `if/else` conditional has the following form:

```
IfElse: If Else?
```

```
If: "if" ConditionList Block
```

```
Else: "else" (Block | IfElse)
```

The construct may include a chain of an arbitrary number of child `else if` clauses.

```
if (payment.amount <= account.balance) {
    account.balance -= payment.amount;
    payment.paid = true;
}
else {
    throw NotEnoughMoneyException();
}
```

```
shared void welcome(User? user) {
    if (exists user) {
```

```
        print("Welcome back, ``user.name``!");
    }
    else {
        print("Welcome to Ceylon!");
    }
}
```

```
if (is CardPayment p = order.payment,
        !p.paid) {
    p.card.charge(total);
}
```

5.3.7. `switch/case/else`

The `switch/case/else` conditional has the following form:

```
SwitchCaseElse: Switch Cases
```

```
Switch: "switch" "(" Expression ")"
```

```
Cases: CaseItem+ DefaultCaseItem?
```

```
CaseItem: "case" "(" Case ")" Block
```

```
DefaultCaseItem: "else" Block
```

Each `case` is either:

- a *value case*—a list of string literals, character literals, integer literals, negated integer literals, and/or value references to anonymous classes, or
- a *type case*—an assignability condition of form `is V` for some type `V`.

```
Case: CaseValue ("," CaseValue)* | "is" Type
```

```
CaseValue: LiteralCase | BaseExpression
```

```
LiteralCase: "-"? IntegerLiteral | CharacterLiteral | StringLiteral | VerbatimStringLiteral
```

Every `case` has a type:

- for a value case, the type is the union of the types of the values, and
- for a type case, the type is the specified type.

The type of a case must be a subtype of the `switch` expression type.

For a value case, each value reference must be to an anonymous class that is a subtype of `Identifiable|Null`.

For a type case of type `V`, the intersection type `V&U` must not be exactly `Nothing`.

Two `cases` are said to be *disjoint* if the intersection of their types is exactly `Nothing`, as defined by §3.4.4 Disjoint types, or if they are both value cases with distinct literal values. In every `switch` statement, all `cases` must be mutually disjoint.

A `switch` is *exhaustive* if there are no literal values in its `cases`, and the union type formed by the types of the `cases` of the `switch` covers the `switch` expression type, as defined by §3.4.1 Coverage.

If no `else` block is specified, the `switch` must be exhaustive.

Note: On the other hand, even if the `switch` is exhaustive, an `else` block may be specified, in order to allow a `switch` that accommodates additional cases without resulting in a compilation error.

As a special exception to the above, if a `switch` occurs in a `dynamic` block, and the `switch` expression has no type, the

cases are not statically type-checked for exhaustion.

Note: an assignability condition `case` may narrow to an intersection or union type.

```
case (is Persistent & Serializable) { ... }
```

```
case (is Integer | Float) { ... }
```

If a `switch` has an assignability condition `case`, then the `switch` expression must be an unqualified value reference to a non-`variable`, non-`default` reference.

For an assignability condition `case`, the value referred by the `switch` expression will be treated by the compiler as having the intersection type of its declared type with the specified type inside the `case` block. This intersection type must not be exactly `Nothing`.

As a special exception to the above, if a `switch` occurs in a `dynamic` block, and the `switch` expression has no type, the value referred by the `switch` expression will be treated by the compiler as having the the specified type inside the `case` block.

```
Boolean? maybe = ... ;
switch (maybe)
case (null, false) {
    return false;
}
case (true) {
    return true;
}
```

```
Integer|Float number = ... ;
switch (number)
case (is Integer) {
    return sqrt(number.float);
}
case (is Float) {
    return sqrt(number);
}
```

A Java-style overloaded method may be emulated as follows:

```
shared void print<Printable>(Printable printable)
        given Value of String | Integer | Float {
    switch (printable)
    case (is String) {
        print("\"``printable``\"");
    }
    case (is Integer) {
        print(printable + ".00");
    }
    case (is Float) {
        print(formatFloat(printable, 2));
    }
}
```

5.3.8. `for/else`

The `for/else` loop has the following form:

```
ForFail: For Fail?
```

```
For: "for" ForIterator Block
```

```
Fail: "else" Block
```

The `for` iterator consists of an iteration variable declaration and an iterated expression that contains the range of values to be iterated.

```
ForIterator: "(" IteratorVariable "in" Expression ")"
```

The type of the iterated expression depends upon the iteration variable declarations:

- The iterated expression must be an expression of type assignable to `Iterable<X>` where `X` is the declared type of the iteration variable.
- If two iteration variables are defined, the iterated expression type must be assignable to `Iterable<Entry<U,V>>` where `U` and `V` are the declared types of the iteration variables.

As a special exception to the above, if a `for` occurs in a `dynamic` block, and the iterated expression has no type, the iterator is not statically type-checked. If the iteration variable does not declare an explicit type, the iteration variable has no type.

```
for (p in people) {
    print(p.name);
}
```

```
variable Float sum = 0.0;
for (i in -10..10) {
    sum += x[i] else 0.0;
}
```

```
for (word -> freq in wordFrequencyMap) {
    print("The frequency of ``word`` is ``freq``.");
}
```

```
for (p in group) {
    if (p.age >= 18) {
        log.info("Found an adult: ``p.name``.");
        break;
    }
}
else {
    log.info("No adult in group.");
}
```

5.3.9. `while`

The `while` loop has the form:

```
While: LoopCondition Block
```

The loop condition list determines when the loop terminates.

```
LoopCondition: "while" ConditionList
```

TODO: does `while` need an `else` block? Python has it, but what is the real usecase?

```
variable Integer n=0;
variable [Integer*] seq = [];
while (n<=max) {
    seq=seq.withTrailing(n);
    n+=step(n);
}
```

5.3.10. `try/catch/finally`

The `try/catch/finally` exception manager has the form:

```
TryCatchFinally: Try Catch* Finally?
```

```
Try: "try" ("(" Resource ("," Resource) ")")? Block
```

```
Catch: "catch" "(" Variable ")" Block
```

```
Finally: "finally" Block
```

Each `catch` block defines a variable. The type of the variable must be assignable to `Exception` in `ceylon.language`. If no type is explicitly specified, the type is inferred to be `Exception`.

Note: a `catch` block type may be a union or intersection type:

```
catch (NotFoundException|DeletedException e) { ... }
```

If there are multiple `catch` blocks in a certain control structure, then:

- The type of a `catch` variable may not be a subtype of any catch variable of an earlier `catch` block belonging to the same control structure.
- If the type of a `catch` variable is a union type `E1|E2|...|En` then no member `Ei` of the union may be a subtype of any catch variable of an earlier `catch` block belonging to the same control structure.

The `try` block may have a list of *resource expressions*, each of which may be either:

- an instantiation expression, or
- an inline variable declaration together with an instantiation expression.

A resource expresson produces a heavyweight object that must be released when execution of the `try` terminates. Each resource expression must be of type assignable to `Closeable` in `ceylon.language`.

```
Resource: Invocation | Variable Specifier
```

If no type is explicitly specified for a resource variable, the type of the variable is inferred to be the type of the instantiation expression.

```
try (file = File(path)) {
    file.open(readOnly);
    ...
}
catch (FileNotFoundException fnfe) {
    print("file not found: ``path``");
}
catch (FileReadException fre) {
    print("could not read from file: ``path``");
}
finally {
    assert (file.closed);
}
```

```
try (Transaction()) {
    try (s = Session()) {
        return s.get(Person, id);
    }
    catch (NotFoundException|DeletedException e) {
        return null;
    }
}
```

5.3.11. Assertions

An assertion has an asserted condition list and, optionally, an annotation list.

```
Assertion: Annotations "assert" ConditionList ";"
```

The message carried by the assertion failure may be specified using a `doc` annotation.

```
"total must be less than well-defined bound"
assert (exists bound, total<bound);
```

If the assertion contains an assignability, existence, or nonemptiness condition containing a value reference then the compiler treats the referenced value as having a narrowed type at program elements that occur in the lexical scope of the assertion.

```
{Element*} elements = ... ;
assert (nonempty elements);
Element first = elements.first;
```

TODO: how can we support interpolation in the assertion failure message?

```
assert (total<bound)
else "total must be less than ``bound``";
```

5.3.12. Dynamic blocks

A `dynamic` block allows interoperation with dynamically typed native code.

```
Dynamic: "dynamic" Block
```

Inside a `dynamic` block an expression may have no type, as specified in Chapter 6, *Expressions*.

An expression with no type:

- may be specified or assigned to a typed value, as defined in §5.2.3 Specification statements,
- may be passed as the argument of a typed parameter in an invocation expression, as defined in §6.6.1 Direct invocations,
- may be the invoked expression of an invocation, as defined in §6.6 Invocation expressions,
- may be returned by a typed function or getter, or thrown as an exception, as defined in §5.2.2 Control directives,
- may be the operand of an operator expression, as defined in §6.5 Compound expressions, or
- may be the subject of a control structure condition, as defined in §5.3.5 Assignability, existence, and nonemptiness conditions, a `switch`, as defined in §5.3.7 switch/case/else, or a `for` iterator, as defined in §5.3.8 for/else.

Furthermore:

- a qualified or unqualified reference may not refer to a statically typed declaration, as defined by §5.1.7 Unqualified reference resolution and §5.1.8 Qualified reference resolution.

These situations result in *dynamic type checking*, as defined in §8.3.6 Dynamic type checking, since the usual static type checks are impossible.

Note: within a `dynamic` block, Ceylon behaves like a language with optional static typing, performing static type checks where possible, and dynamic type checking where necessary.

Chapter 6. Expressions

An *expression* produces a value when executed. An algorithm expressed using functions and expressions, rather than sequences of statements is often easier to understand and refactor. Therefore, Ceylon has a highly flexible expressions syntax. Expressions are formed from:

- literal values, string templates, and self references,
- evaluation and assignment of values,
- invocation of functions and instantiation of classes,
- callable references, static references, and anonymous functions,
- comprehensions,
- metamodel references,
- enumeration of iterables and tuples, and
- operators.

Ceylon expressions are validated for typesafety at compile time. To determine whether an expression is assignable to a program element such as a value or parameter, Ceylon considers the *type* of the expression (the type of the objects that are produced when the expression is evaluated). An expression is assignable to a program element if the type of the expression is assignable to the declared type of the program element.

Within a `dynamic` block, an expression may have no type, in the sense that its type can not be determined using static analysis of the code.

6.1. Literal values

Ceylon supports literal values of the following types:

- `Integer` and `Float`,
- `Character`, and
- `String`.

The types `Integer`, `Float`, `Character`, and `String` are defined in the module `ceylon.language`.

Note: Ceylon does not need a special syntax for `Boolean` literal values, since `Boolean` is just a class with the cases `true` and `false`. Likewise, `null` is just the singleton value of an anonymous class.

```
Literal: IntegerLiteral | FloatLiteral | CharacterLiteral | StringLiteral | VerbatimStringLiteral
```

All literal values are instances of immutable types. The value of a literal expression is an instance of the type. How this instance is produced is not specified here.

6.1.1. Integer number literals

An integer literal, as defined in §2.4.1 Numeric literals, is an expression of type `Integer`, representing a numeric integer.

```
Integer five = 5;
```

```
Integer mask = $1111_0000;
```

```
Integer white = #FFFF;
```

6.1.2. Floating point number literals

A floating point literal, as defined in §2.4.1 Numeric literals, is an expression of type `Float`, a floating-point representation of a numeric value.

```
shared Float pi = 3.14159;
```

6.1.3. Character literals

A single character literal, as defined in §2.4.2 Character literals, is an expression of type `Character`, representing a single 32-bit Unicode character.

```
if (exists ch=string[i], ch == '+') { ... }
```

6.1.4. Character string literals

A character string literal or verbatim string, as defined in §2.4.3 String literals, is an expression of type `String`, representing a sequence of Unicode characters.

```
person.name = "Gavin King";
```

```
print("Melbourne\tVic\tAustralia\nAtlanta\tGA\tUSA\nGuanajuato\tGto\tMexico\n");
```

```
String verbatim = """A verbatim string can have \ or a " in it.""";
```

6.2. String templates

A character *string template* contains interpolated expressions, surrounded by character string fragments.

```
StringTemplate: StringStart (ValueExpression StringMid)* ValueExpression StringEnd
```

Each interpolated expression contained in the string template must have a type assignable to `Object` defined in `ceylon.language`.

```
print("Hello, ``person.firstName`` ``person.lastName``, the time is ``Time()``.");
```

```
print("1 + 1 = ``1 + 1``");
```

A string template is an expression of type `String`.

6.3. Self references and the current package reference

The type of the following expressions depends upon the context in which they appear.

```
SelfReference: "this" | "super" | "outer" | "package"
```

A self reference expression may not occur outside of a class or interface body.

The *immediately containing class or interface* for a program element is the class or interface in which the program element occurs, and which contains no other class or interface in which the program element occurs. If there is no such class or interface, the program element has no immediately containing class or interface.

A `this`, `outer`, or `super` self reference must have an immediately containing class or interface. An `outer` self reference must have an immediately containing class or interface for its immediately containing class or interface.

6.3.1. `this`

The keyword `this` refers to the current instance, as defined in §8.2.3 Current instance of a class or interface, of the immediately containing class or interface (the class or interface in which the expression appears). Its type is the type of the immediately containing class or interface.

6.3.2. `outer`

The keyword `outer` refers to the current instance, as defined in §8.2.3 Current instance of a class or interface, of the class or interface which immediately contains the immediately containing class or interface. Its type is assignable to the type of this class or interface.

6.3.3. `super`

The keyword `super` refers to the current instance of the immediately containing class or interface. Its type is the intersection of the immediate superclass and all immediate superinterfaces of the class. A member reference such as `super.x` may not resolve to a `formal` declaration, nor to any member inherited from more than one supertype of the intersection type.

The keyword `super` may occur as the first operand of an `of` operator, in which case the second operand is any supertype of the class. The expression `(super of Type)` has type `Type`. A member reference such as `(super of Type).x` may not resolve to a `formal` member, nor to any member inherited from more than one supertype of `Type`, nor to any member that is refined by the class or any intermediate supertype of the class.

6.3.4. `package`

The keyword `package` is not an expression, and does not have a well-defined type. However, it may be used to qualify and disambiguate a value reference or callable reference. A value reference or callable reference qualified by the keyword `package` always refers to a toplevel member of the containing package, never to an imported declaration or nested declaration.

6.4. Anonymous functions

An anonymous function is a function, as specified in §4.7 Functions, with no name, defined within an expression. It comprises one or more parameter lists, followed by an expression.

```
FunctionExpression: ("function" | "void")? Parameters+ (LazySpecifier | Block)
```

The parameters are the parameters of the function. The lazy specifier or block of code is the implementation of the function. If the `void` keyword is specified, the function is a `void` function. Otherwise, it is a non-`void` function, and its return type is inferred.

The type of an anonymous function expression is the callable type of the function, as specified in §4.7.1 Callable type of a function.

```
(Value x, Value y) => x<=>y
```

```
void (String name) => print(name)
```

```
(String string) {
    value mid = string.size % 2;
    return [string[0..mid],string[mid+1...]];
}
```

An anonymous function occurring in an `extends` clause may not contain a reference to a variable value.

Note: evaluation of an anonymous function expression, as defined in §8.4.5 Evaluation of anonymous functions results in instantiation of an object of type `Callable`. However, the members of this object are never in scope, do not hide other declarations, and are not referenceable from within the anonymous function.

Note: there is almost no semantic difference between the following function declarations:

```
Float f(Float x)(Float y) => x*y;
```

```
Float(Float) f(Float x) => (Float y) => x*y;
```

The first form is strongly preferred.

6.5. Compound expressions

An *atom* is a literal or self reference, a string template, an iterable or tuple enumeration, or a parenthesized expression.

```
Atom: Literal | StringTemplate | SelfReference | GroupedExpression | Iterable | Tuple | DynamicValue
```

A *primary* is formed by recursively forming member expressions and invocation expressions from an atom, base expression, or static expression.

```
Primary: Atom | BaseExpression | MemberExpression | StaticExpression | Invocation | Meta | Dec
```

More complex expressions are formed by combining expressions using operators, including assignment operators, and anonymous functions.

```
ValueExpression: Primary | OperatorExpression
```

```
Expression: ValueExpression | FunctionExpression | OperatorInvocation | OperatorMemberExpression
```

Parentheses are used for grouping:

```
GroupedExpression: "(" Expression ")"
```

A compound expression occurring in a `dynamic` block, and involving a qualified or unqualified reference with no type, or a reference to a declaration with no type, may also have no type.

In particular, if an operand expression has no type, and the type of the operator expression depends upon the type of the operand, and the operator expression occurs within a `dynamic` block, then the whole operator expression has no type.

6.5.1. Base expressions

A *base expression* is an unqualified identifier, with an optional list of type arguments:

```
BaseExpression: (MemberName | TypeName) TypeArguments?
```

A base expression is either:

- a reference to a toplevel function, toplevel value, or toplevel class,
- a reference within the lexical scope of the referenced function, value, or class, or
- a reference within the body of the referenced function, value, or class.

The referenced declaration is determined by resolving the unqualified reference as defined by §5.1.7 Unqualified reference resolution. The unqualified realization for the unqualified reference is determined according to §3.7.6 Realizations.

The type argument list, if any, must conform, as defined by §3.6.1 Type arguments and type constraints, to the type parameter list of the unqualified realization.

If a base expression is a reference to an attribute, method, or member class of a class, the receiving instance is the current instance of that class, as defined by §8.2.3 Current instance of a class or interface. Otherwise, there is no receiving instance.

6.5.2. Member expressions

A *member expression* is a *receiver expression*, followed by an identifier, with an optional list of type arguments.

```
MemberExpression: (Primary ".") (MemberName | TypeName) TypeArguments?
```

A member expression is a reference to a member of a type: an attribute, method, or member class.

The referenced member is determined by resolving the qualified reference as defined by §5.1.8 Qualified reference resolu-

tion. The qualified realization for the qualified reference is determined according to §3.7.6 Realizations.

The type argument list, if any, must conform, as defined by §3.6.1 Type arguments and type constraints, to the type parameter list of the qualified realization.

The receiver expression produces the instance upon which the member is invoked or evaluated. When a member expression is executed, the receiver expresson is evaluated to produce the receiving instance which is held until the member is invoked or evaluated, as defined in §8.4 Evaluation, invocation, and assignment.

6.5.3. Value references

A *value reference* is a base expression or member expression that references a value declaration.

The type of a value reference expression is the type of the realization of the referenced value.

A value declaration is never generic, so a value reference never has a type argument list.

A value reference that does not occur within any `dynamic` block may not refer to a value declaration or value parameter with no type.

A value reference which occurs within a `dynamic` block and which does not reference any statically typed declaration, or which references a value declaration or value parameter with no type, has no type.

If a base expression or member expression does not reference any statically typed declaration, and occurs within a `dynamic` block, then it is considered a value reference.

6.5.4. Callable references

A *callable reference* is a base expression or member expression that references something—a function or class—that can be *invoked* or *instantiated* by specifying a list of arguments.

A callable reference may be invoked immediately, or it may be passed to other code which may invoke the reference. A callable reference captures the return type and parameter list types of the function or class it refers to, allowing compile-time validation of argument types when the callable reference is invoked.

The type of a callable reference expression is the callable type of the realization of the referenced function or class.

If a callable reference expression refers to a generic declaration, either:

- it must be immediately followed by an argument list, allowing the compiler to infer the type arguments, or
- it must have an explicit type argument list.

A callable reference may not appear as the receiver expression of a member expression.

Note: this restriction exists to eliminate an ambiguity in the interpretation of static expressions such as `Person.string` and `Person.equals`.

A callable reference that does not occur within any `dynamic` block may not refer to a function declaration with no return type.

A callable reference which occurs within a `dynamic` block and which references a function declaration with no return type, has no type.

Note: in a future release of the language, we would like to add a syntax for obtaining a callable reference to an attribute, something like `person.@name`, to allow attributes to be passed by reference. This would also allow static references like `Person.@name`.

6.5.5. Static expressions

A *static expression* is a type, followed by an identifier, with an optional list of type arguments.

```
StaticExpression: (TypeName TypeArguments? ".")+ (MemberName | TypeName) TypeArguments?
```

A static expression is a reference to a member of a type: an attribute, method, or member class.

The referenced member is determined by resolving the qualified reference as defined by §5.1.8 Qualified reference resolution. The qualified realization for the qualified reference is determined according to §3.7.6 Realizations.

The type argument list, if any, must conform, as defined by §3.6.1 Type arguments and type constraints, to the type parameter list of the qualified realization.

Unlike member expressions, a static expression does not have a receiver expression. All static expressions are callable expressions which accept an argument of the specified type.

A static expression must reference a statically typed declaration with no missing types, even within a `dynamic` block.

6.5.6. Static value references

A *static value reference* is a static expression that references an attribute declaration.

```
List<Anything>.size
```

The type of a static value reference expression for an attribute whose realization is of type `X`, and with qualifying type `T`, is `X(T)`.

A value declaration is never generic, so a static value reference never ends in a type argument list.

6.5.7. Static callable references

A *static callable reference* is a static expression that references something—a method or member class—that can be *invoked* or *instantiated*.

```
List<String>.filter
```

```
Iterable<Integer>.map<String>
```

The type of a static callable reference expression for a method or member class whose realization has callable type `C`, and with qualifying type `T`, is `C(T)`.

If a callable reference expression refers to a generic declaration, it must end in an explicit type argument list.

6.6. Invocation expressions

A callable expression—any expression of type `Callable`—is *invokable*. An *invocation* consists of an *invoked expression*, together with an argument list and, optionally, an explicit type argument list.

```
Invocation: Primary Arguments
```

The invoked expression must be of type `Callable<R,P>` for some types `R` and `P`. Then the type of the invocation expression is simply `R`.

If the invoked expression has no type, and occurs within a `dynamic` block, then the whole invocation expression has no type, and the argument list is not type-checked at compile time, unless it is a direct invocation expression.

An invocation expression must specify arguments for parameters of the callable object, either as a positional argument list, or as a named argument list.

```
Arguments: PositionalArguments | NamedArguments
```

Every argument list has a type, as specified below in §6.6.7 Positional argument lists and §6.6.8 Named argument lists. If an invocation is formed from a callable expression of type exactly `Callable<R,P>` and an argument list of type `A`, then `A` must be a subtype of `P`.

TODO: should we support an infix-operator-style syntax for method invocation like `string split ",;".contains`? This

is especially nice for conceptually symmetric operations like `a xor b`*, or when the argument is an anonymous function like* `people map (Person p)=>p.firstName+p.lastName`*.*

6.6.1. Direct invocations

Any invocation expression where the invoked expression is a callable reference expression is called a *direct invocation expression* of the function or class to which the callable reference refers.

TODO: Should we consider `x{y=1;}{z=2;}` *a legal direct invocation if* `x` *has multiple parameter lists?*

In a direct invocation expression:

- the compiler has one item of additional information about the schema of the method or class that is not reified by the `Callable` interface: the names of the parameters of the function or class, and therefore named arguments may be used, and
- type argument inference is possible, as defined in §3.6.3 Type argument inference, since the compiler has access to the type parameters and constraints of the function or class.

If an invocation expression has a named argument list, it must be a direct invocation.

The type of a direct invocation expression is the return type of the realization of the function, or the type of the realization of the class, as defined in §3.7.6 Realizations.

If the function has no return type, and occurs within a `dynamic` block, then the whole direct invocation expression has no type.

In a direct invocation expression of a function or class, the restriction above on the argument list type is equivalent to the following requirements. Given the parameter list of the realization of the function or class, and the arguments of the direct invocation:

- for each required parameter, an argument must be given,
- for each defaulted parameter, an argument may optionally be given,
- if the parameter list has a variadic parameter of type `T+`, one or more arguments must be given,
- if the parameter list has a variadic parameter of type `T*`, one or more arguments may optionally be given,
- no additional arguments may be given,
- for a required or defaulted parameter of type `T`, the type of the corresponding argument expression must be assignable to `T`, and
- for a variadic parameter of type `T*` or `T+`, the type of every corresponding argument expression must be assignable to `T`.

Furthermore, if type argument are inferred, then the inferred type arguments must conform, as defined by §3.6.1 Type arguments and type constraints, to the type parameter list of the realization of the function or class.

If an argument expression has no type, or if its parameter has no type, and the invocation occurs within a `dynamic` block, then the argument is not type-checked at compile time.

An invocation expression that does not occur within any `dynamic` block may not assign an argument to a value parameter with no type.

6.6.2. Default arguments

When no argument is assigned to a defaulted parameter by the caller, the default argument defined by the parameter declaration of the realization, as defined by §3.6.1 Type arguments and type constraints, of the function or class is used. The default argument expression is evaluated every time the method is invoked with no argument specified for the defaulted parameter.

This class:

```
shared class Counter(Integer initialCount=0) { ... }
```

May be instantiated using any of the following invocations:

```
Counter()
```

```
Counter(1)
```

```
Counter {}
```

```
Counter { initialCount=10; }
```

6.6.3. The type of a list of arguments

A list of arguments may be formed from:

- any number of *listed arguments*, optionally followed by either
- a *spread argument*, or
- a *comprehension*.

```
ArgumentList: ((ListedArgument ",")* (ListedArgument | SpreadArgument | Comprehension))?
```

Every such list of arguments has a type, which captures the types of the individual arguments in the list. This type is always a subtype of `Anything[]`. The type of an empty list of arguments is `[]`.

6.6.4. Listed arguments

A listed argument is an expression.

```
ListedArgument: Expression
```

If a listed argument is an expression of type `T`, and a list of arguments has type `P` with principal instantiation `Sequential<Y>`, then the type of a new argument list formed by prepending the expression to the first parameter list is `Tuple<T|Y,T,P>`.

6.6.5. Spread arguments

A spread argument is an expression prefixed by the *spread operator* `*`.

```
SpreadArgument: "* "Expression
```

The expression type `T` must have the principal instantiation `{X*}` for some type `X`. We form the *sequential type of a spread argument* as follows:

- if the expression type `T` is an invariant subtype of `X[]`, for some type `X` then the sequential type of the spread argument is `T`, or, if not,
- if the expression type `T` is an invariant subtype of `{X+}`, for some type `X` then the sequential type of the spread argument is `[X+]`, or, otherwise,
- the expression type `T` is an invariant subtype of `{X*}`, for some type `X` and the sequential type of the spread argument is `X[]`,

When a spread argument with an expression type not assignable to `Anything[]` is evaluated, the elements of the iterable automatically are packaged into a sequence.

Note: the spread "operator" is not truly an operator in the sense of §6.8 Operators, and so a spread argument is not an expresson. An expression, when evaluated, produces a single value. The spread operator produces multiple values. It is

therefore more correct to view the spread operator as simply part of the syntax of an argument list.

The type of a list of arguments containing only a spread argument of sequential type `S` is simply `S`.

6.6.6. Comprehensions

A *comprehension* accepts one or more streams of values and produces a new stream of values. Any instance of `Iterable` is considered a stream of values. The comprehension has two or more *clauses*:

- A `for` clause specifies a source stream and an iteration variable, as defined in §5.3.2 Iteration variables, representing the values produced by the stream.
- An `if` clause specifies a condition list, as defined in §5.3.3 Control structure conditions, used to filter the values produced by the source stream or streams.
- An expression clause produces the values of the resulting stream.

Every comprehension begins with a `for` clause, and ends with an expression clause. There may be any number of intervening `for` or `if` clauses. Each clause in the comprehension is considered a child of the clause that immediately precedes it.

```
Comprehension: ForComprehensionClause
```

```
ForComprehensionClause: "for" ForIterator ComprehensionClause
```

```
IfComprehensionClause: "if" ConditionList ComprehensionClause
```

```
ComprehensionClause: ForComprehensionClause | IfComprehensionClause | Expression
```

An expression that occurs in a child clause may refer to iteration variables and condition variables declared by parent clauses. The types of such variables are specified in §5.3 Control structures and assertions.

Note: each child clause can be viewed as a body nested inside the parent clause. The scoping rules for variables declared by comprehension clauses reflects this model.

The type of a list of arguments containing only a comprehension is `[T*]` where `T` is the type of the expression which terminates the comprehension, or `[T+]` if there are no `if` clauses, and if every `for` clause has an iterated expression of nonempty type.

An comprehension occurring in an `extends` clause may not contain a reference to a variable value.

Note: a comprehension, like a spread argument, is not considered an expression. An expression, when evaluated, produces a single value. A comprehension, produces multiple values, like a spread argument, or like a series of listed arguments. Therefore, a comprehension may only appear in an argument list or an enumeration expression. This is however, no limitation; we can simply wrap the comprehension in braces in order to get an expression of type `{T*}`, or in brackets to get an expression of type `[T*]`.

TODO: properly define how expressions with no type occurring in a `dynamic` block affect comprehensions.

6.6.7. Positional argument lists

When invocation arguments are listed positionally, the argument list is enclosed in parentheses.

```
PositionalArguments: "(" ArgumentList ")"
```

The type of the positional argument list is the type of the list of arguments it contains.

6.6.8. Named argument lists

When invocation arguments are listed by name, the argument list is enclosed in braces.

```
NamedArguments: "{" NamedArgument* ArgumentList "}"
```

Named arguments may be listed in a different order to the corresponding parameters.

Each named argument in a named argument list is either:

- an *anonymous argument*—an expression, with no parameter name explicitly specified,
- a *specified argument*—a specification statement where name of the value of function being specified is interpreted as the name of a parameter, or
- an inline getter, function, or anonymous class declaration, whose name is interpreted as the name of a parameter.

```
NamedArgument: AnonymousArgument | SpecifiedArgument | InlineDeclarationArgument
```

Additionally, a named argument list has an ordinary list of arguments, which may be empty. This argument list is interpreted as a single argument to a parameter of type `Iterable`.

```
{ initialCapacity=2; "hello", "world" }
```

```
{ initialCapacity=people.size; loadFactor=0.8; for (p in people) p.name->p }
```

Note: in a future release of the language, we would like to be able to assign a local name to an anonymous argument or listed argument, allowing it to be referenced later in the argument list. We might consider this a kind of "let" expression, perhaps.

Given a parameter list, and a named argument list, we may attempt to construct an *equivalent positional argument list* as follows:

- Taking each argument in the named argument list in turn, on the order they occur lexically:
  - if the argument is anonymous, assign it to the first unassigned parameter of the parameter list, or
  - if the argument is named, assign it to the parameter with that name in the parameter list.

  If for any argument, there is no unassigned parameter, no parameter with the given name, or the parameter with the given name has already been assigned an argument, construction of the positional argument list fails, and the invocation is not well-typed.
- Next, if the parameter list has an unassigned parameter of type exactly `Iterable<T,N>` for some types `T` and `N`, then an iterable enumeration expression, as defined in §6.6.12 Iterable and tuple enumeration, is formed from the ordinary list of arguments, and assigned to that parameter.

  If there is no such parameter, and the the ordinary list of arguments is nonempty, then construction of the positional argument list fails, and the invocation is not well-typed.
- Finally, we assign each unassigned defaulted parameter its default argument.

The resulting equivalent positional argument list is formed by ordering the arguments according to the position of their corresponding parameters in the parameter list, and then replacing any inline value, function, or object declarations with a reference to the declaration.

The type of a named argument list is the type of the equivalent positional argument list.

6.6.9. Anonymous arguments

An anonymous argument is just an expression followed by a semicolon.

```
AnonymousArgument: Expression ";"
```

The type of the argument is the type of the expression.

```
{
    Head { title="Hello"; };
    Body {
```

```
        Div { "Hello " name "!" };
    };
}
```

6.6.10. Specified arguments

A specified argument is a value specification statement or lazy specification statement, as defined in §5.2.3 Specification statements, where the value reference or callable reference is treated as the name of a parameter of the invoked function or class instead of using the usual rules for resolving unqualified names.

```
SpecifiedArgument: Specification
```

- If a specified argument is a value specification statement, its type is the type of the specified expression.
- If a specified argument is a lazy specification statement with no parameter lists, its type is the type of the specified expression.
- Otherwise, if it is a lazy specification statement with a parameter list, its type is the callable type formed from the type of the expression, interpreted as a function return type, and the types of its parameter lists, according to §4.7.1 Callable type of a function.

Note: there is an ambiguity here between assignment expressions and specified arguments. This ambiguity is resolved in favor of interpreting the argument as a specified argument. Therefore an anonymous argument in a named argument list may not be an assignment expression.

```
{
    product = getProduct(id);
    quantity = 1;
}
```

```
{
    by(Value x, Value y) => x<=>y;
}
```

6.6.11. Inline declaration arguments

An *inline declaration argument* defines a getter, function, or anonymous class, and assigns it to a parameter.

```
ValueArgument | FunctionArgument | ObjectArgument
```

An inline getter argument is a streamlined getter declaration, as defined in §4.8.2 Getters. The type of the argument is the declared or inferred type of the getter.

```
ValueArgument: ValueHeader (Block | (Specifier | LazySpecifier) ";")
```

An inline function argument is a streamlined function declaration, as defined in §4.7 Functions. The type of the argument is the callable type of the function, as defined by §4.7.1 Callable type of a function.

```
FunctionArgument: FunctionHeader (Block | LazySpecifier ";")
```

An inline anonymous class argument is a streamlined anonymous class declaration, as defined in §4.5.7 Anonymous classes. The type of the argument is the anonymous class type.

```
ObjectArgument: ObjectHeader ClassBody
```

A named argument may not have type parameters or annotations.

```
{
    description = "Total";
    value amount {
        variable Float total = 0.0;
        for (Item item in items) {
            sum += item.amount;
        }
        return total;
```

```
    }
}
```

```
{
    label = "Say Hello";
    void onClick() {
        say("Hello!");
    }
}
```

```
{
    function by(Value x, Value y) => x<=>y;
}
```

```
{
    object iterator
            satisfies Iterator<Order> {
        variable value done = false;
        shared actual Order|Finished next() {
            if (done) {
                return finished;
            }
            else {
                done=true;
                return order;
            }
        }
    }
}
```

6.6.12. Iterable and tuple enumeration

An *enumeration expression* is an abbreviation for tuple and iterable object instantiation. Iterable enumerations are delimited using braces. Tuple enumerations are delimited by brackets.

```
Iterable: "{" ArgumentList "}"
```

```
Tuple: "[" ArgumentList "]"
```

The type of an iterable enumeration expression is:

- `Empty` if there are no argument expressions, or
- `Iterable<U,Nothing>` where `U`, the argument expression list is an invariant suptype of `U[]`.

The type of a tuple enumeration expression is the type of the list of arguments it contains.

```
{String+} = { "hello", "world" };
```

```
[] none = [];
```

```
[Float,Float] xy = [x, y];
```

```
[Float,Float, String*] xy = [x, y, *labels];
```

Every argument expression must have a type, even if the enumeration expression occurs in a `dynamic` block.

6.6.13. Dynamic enumerations

A *dynamic enumeration expression* creates a new object with no class by enumerating its members, allowing interoperation with dynamically typed native code.

```
DynamicValue: "value" NamedArguments
```

A dynamic enumeration expression has no type.

Any argument names may be specified in the named argument list.

A dynamic enumeration expression must occur inside a `dynamic` block.

The semantics of this construct are platform-dependent and beyond the scope of this specification.

6.7. Conditional expressions and anonymous class expressions

A *conditional expression* resembles a control structure but is part of the expression syntax, and evaluates to a value.

An *inline class* is an anonymous class defined within an expression.

6.7.1. Inline conditional expressions

We plan to support inline `if/then/else` conditional expressions, for example:

```
Integer port = if (exists setting = process.propertyValue("port"))
                        then parseInteger(setting) else 8080;
```

Note that this is more powerful than the `then` and `else` operators because it allows all kinds of conditions, not only boolean conditions.

Should we also support:

- *inline `switch/case/else` conditional expressions, or even*
- *inline `try/catch` exceptional conditions?*

For example:

```
Float evaluated => switch (expr)
        case (is Literal) expr.integer
        case (is Plus) expr.left.evaluated + expr.right.evaluated
        case (is Times) expr.left.evaluated * expr.right.evaluated;
```

6.7.2. Let expressions

Should we support let expressions, possibly reusing the keyword `given`?

```
given (dist = sqrt(x^2+y^2)) [x/dist,y/dist]
```

6.7.3. Inline anonymous class expressions

Should we support inline `object` declarations, for example:

```
iterator => object satisfies Iterable<Nothing> { next() => finished; }
```

Or, alternatively, a kind of named argument ""instantiation" syntax for interfaces and abstract classes:

```
iterator => Iterable { next() => finished; }
```

The first option is more flexible, but also more verbose. The second is streamlined for the common case and might even be able to do type argument inference as shown here.

If we go with the first option, should we support inline `class` declarations? This would be like an inline `object` and an anonymous function rolled into one. We could even support `class` arguments in named argument lists.

6.8. Operators

Operators are syntactic shorthand for more complex expressions involving invocation, evaluation, or instantiation. There is

no support for user-defined *operator overloading*:

- new operator symbols may not be defined outside of the operators specified below, and
- the definition of the operators specified below may not be changed or overloaded.

However, many of the operators below are defined in terms of `default` or `formal` methods or attributes. So, within well-defined limits a concrete implementation may customize the behavior of an operator. This approach is called *operator polymorphism*.

Some examples:

```
Float z = x * y + 1.0;
```

```
even = n % 2 == 0;
```

```
++count;
```

```
Integer j = i++;
```

```
if ( x > 100 || x < 0 ) { ... }
```

```
User user = users[userId] else guest;
```

```
List<Item> firstPage = results[0..20];
```

```
for (n in 0:length) { ... }
```

```
if (char in 'A'..'Z') { ... }
```

```
String[] names = people*.name;
```

```
this.total += item.price * item.quantity;
```

```
Float vol = length^3;
```

```
Vector scaled = scale ** vector;
```

```
map.define(person.name->person);
```

```
if (!document.internal || user is Employee) { ... }
```

6.8.1. Operator precedence

There are 18 distinct operator precedence levels, but these levels are arranged into layers in order to make them easier to predict.

- Operators in layer 1 produce, transform, and combine values.
- Operators in layer 2 compare or predicate values, producing a `Boolean` result.
- Operators in layer 3 are logical operators that operate upon `Boolean` arguments to produce a `Boolean` value.
- Operators in layer 4 perform assignment and conditional evaluation.

Within each layer, postfix operators have a higher precedence than prefix operators, and prefix operators have a higher precedence than binary operators.

There is a single exception to this principal: the binary exponentiation operator `^` has a higher precedence than the prefix operators `+` and `-`. The reason for this is that the following expressions should be equivalent:

```
-x^2      //means -(x^2)
```

```
0 - x^2    //means 0 - (x^2)
```

This table defines the relative precedence of the various operators, from highest to lowest, along with associativity rules:

Table 6.1.

Operations	Operators	Type	Associativity
Layer 1			
Member invocation and selection, index, span, postfix increment, postfix decrement:	`.`, `*.`, `?.`, `()`, `{}`, `[]`, `[:]`, `[..]`, `[...]`, `++`, `--`	Binary / ternary / N-ary / unary postfix	Left
Prefix increment, prefix decrement:	`++`, `--`	Unary prefix	Right
Exponentiation:	`^`	Binary	Right
Negation:	`+`, `-`	Unary prefix	Right
Set intersection:	`&`	Binary	Left
Set union and complement:	`|`, `~`	Binary	Left
Multiplication, division, remainder:	`*`, `/`, `%`	Binary	Left
Scale:	`**`	Binary	Right
Addition, subtraction:	`+`, `-`	Binary	Left
Range and entry construction:	`..`, `:`, `->`	Binary	None
Layer 2			
Existence, emptiness:	`exists`, `nonempty`	Unary postfix	None
Comparison, containment, assignability, inheritance:	`<=>`, `<`, `>`, `<=`, `>=`, `in`, `is`, `of`, `satisfies`	Binary (and ternary)	None
Equality, identity:	`==`, `!=`, `===`	Binary	None
Layer 3			
Logical not:	`!`	Unary prefix	Right
Logical and:	`&&`	Binary	Left
Logical or:	`||`	Binary	Left
Layer 4			
Conditionals:	`then`, `else`	Binary	Left
Assignment:	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `~=`, `&&=`, `||=`	Binary	Right

It's important to be aware that in Ceylon, compared to other C-like languages, the logical not operator `!` has a very low precedence. The following expressions are equivalent:

```
!x.y == 0.0  //means !(x.y == 0.0)
```

```
x.y != 0.0
```

6.8.2. Operator definition

The following tables define the semantics of the Ceylon operators. There are six basic operators which do not have a definition in terms of other operators or invocations:

- the *member selection* operator `.` separates the receiver expression and member name in a member expression, as defined above in §6.5.2 Member expressions,
- the *argument specification* operators `()` and `{}` specify the argument list of an invocation, as defined in §6.6 Invocation expressions and §8.4.4 Invocation,
- the *assignment* operator `=` assigns a new value to a variable and returns the new value after assignment, as defined in §8.4.3 Assignment,
- the *identity* operator `===` evaluates to `true` if its argument expressions evaluate to references to the same object, as defined in §8.1 Object instances, identity, and reference passing, or to `false` otherwise,
- the *assignability* operator `is` evaluates to `true` if its argument expression evaluates to an instance of a class, as defined in §8.1 Object instances, identity, and reference passing, that is a subtype of the specified type, or to `false` otherwise, and
- the *coverage* operator `of` narrows or widens the type of an expression to any specified type that covers the expression type, as defined by §3.4.1 Coverage, without affecting the value of the expression.

All other operators are defined below in terms of other operators and/or invocations.

In the tables, the following pseudo-code is used, which is not legal Ceylon syntax:

First,

```
if (b) then x else y   //pseudocode
```

means the value of `result` after execution of the following:

```
X result; if (b) { result=x; } else { result=y; }
```

Second,

```
let t=x in y   //pseudocode
```

means the value of `result` after execution of the following:

```
X t = x; Y result=y;
```

6.8.3. Basic invocation and assignment operators

These operators support method invocation and attribute evaluation and assignment.

Table 6.2.

Example	Name	Definition	LHS type	RHS type	Return type
Invocation					
`lhs.member`	member		`X`	a member of `X`, of type `T`	`T`
`lhs(x,y,z)` or `lhs{a=x;b=y;}`	invoke		`Callable <T,P>`	argument list of type `P`	`T`
Assignment					
`lhs = rhs`	assign		variable of type `X`	`X`	`X`

Example	Name	Definition	LHS type	RHS type	Return type
Coverage					
`lhs of Type`	of		`X`	a literal type `T` that covers `X`	`T`

6.8.4. Equality and comparison operators

These operators compare values for equality, order, magnitude, or membership, producing boolean values.

Table 6.3.

Example	Name	Definition	LHS type	RHS type	Return type
Equality and identity					
`lhs === rhs`	identical	`identical(lhs,rhs)`	`X given X satisfies Identifiable`	`Y given Y satisfies Identifiable` where `X&Y` is not `Nothing`	`Boolean`
`lhs == rhs`	equal	`lhs.equals(rhs)`	`Object`	`Object`	`Boolean`
`lhs != rhs`	not equal	`!lhs.equals(rhs)`	`Object`	`Object`	`Boolean`
Comparison					
`lhs <=> rhs`	compare	`lhs.compare(rhs)`	`Comparable <T>`	`T`	`Comparison`
`lhs < rhs`	smaller	`lhs.compare(rhs)==smaller`	`Comparable <T>`	`T`	`Boolean`
`lhs > rhs`	larger	`lhs.compare(rhs)==larger`	`Comparable <T>`	`T`	`Boolean`
`lhs <= rhs`	small as	`lhs.compare(rhs)!=larger`	`Comparable <T>`	`T`	`Boolean`
`lhs >= rhs`	large as	`lhs.compare(rhs)!=smaller`	`Comparable <T>`	`T`	`Boolean`
Containment					
`lhs in rhs`	in	`let x=lhs in rhs.contains(x)`	`Object`	`Category`	`Boolean`
Assignability					
`rhs is Type`	is		any type which is not a subtype of `T`, whose intersection with `T` is not `Nothing`	any literal type `T`	`Boolean`

TODO: Should we have allow the operators `<=` and `>=` to handle partial orders? A particular usecase is `Set` comparison.

A *bounded comparison* is an abbreviation for two binary comparisons:

- `l<x<u` means `x>l && x<u`,
- `l<=x<u` means `x>=l && x<u`,

- `l<x<=u` means `x>l && x<=u`, and
- `l<=x<=u` means `x>=l && x<=u`

for expressions `l`, `u`, and `x`.

These abbreviations have the same precedence as the binary `<` and `<=` operators, and, like the binary forms, are not associative.

6.8.5. Logical operators

These are the usual logical operations for boolean values.

Table 6.4.

Example	Name	Definition	LHS type	RHS type	Return type
Logical operators					
`!rhs`	not	`if (rhs) false else true`		`Boolean`	`Boolean`
`lhs || rhs`	conditional or	`if (lhs) true else rhs`	`Boolean`	`Boolean`	`Boolean`
`lhs && rhs`	conditional and	`if (lhs) rhs else false`	`Boolean`	`Boolean`	`Boolean`
Logical assignment					
`lhs ||= rhs`	conditional or	`if (lhs) true else lhs=rhs`	variable of type `Boolean`	`Boolean`	`Boolean`
`lhs &&= rhs`	conditional and	`if (lhs) lhs=rhs else false`	variable of type `Boolean`	`Boolean`	`Boolean`

6.8.6. Operators for handling null values

These operators make it easy to work with optional expressions.

Table 6.5.

Example	Name	Definition	LHS type	RHS type	Return type
Existence					
`lhs exists`	exists	`if (exists lhs) true else false`	any type whose intersections with `Object` and `Null` are not `Nothing`		`Boolean`
`lhs nonempty`	nonempty	`if (nonempty lhs) true else false`	any subtype of `Anything[]?` whose intersections with `[]` and `[Nothing+]` are not `Nothing`		`Boolean`
Nullsafe invocation					
`lhs?.member`	nullsafe attribute	`if (exists lhs) lhs.member else null`	`X?`	an attribute of type `T` of `X`	`T?`
`lhs?.member`	nullsafe		`X?`	a method of	`Callable`

Example	Name	Definition	LHS type	RHS type	Return type
	method			callable type `Callable<T,P>` of `X`	`<T?,P>`

6.8.7. Correspondence and sequence operators

These operators provide a simplified syntax for accessing values of a `Correspondence`, and for joining and obtaining subranges of `Sequences`.

Table 6.6.

Example	Name	Definition	LHS type	RHS type	Return type
Keyed item access					
`lhs[index]`	lookup	`lhs.item(index)`	`Correspondence<X,Y>`	`X`	`Y?`
Spans and segments					
`lhs[from:length]`	segment	`lhs.segment(from,length)`	`Ranged<X,Y>`	`X`, `Integer`	`Y`
`lhs[from..to]`	span	`lhs.span(from,to)`	`Ranged<X,Y>`	`X`, `X`	`Y`
`lhs[from...]`	upper span	`lhs.spanFrom(from)`	`Ranged<X,Y>`	`X`	`Y`
`lhs[...to]`	lower span	`lhs.spanTo(to)`	`Ranged<X,Y>`	`X`	`Y`
Spread invocation					
`lhs*.member`	spread attribute	`[ for (X x in lhs) x.member ]`	`X[]`	attribite of `X` of type `T`	`T[]`
`lhs*.member`	spread method		`X[]`	method of `X` of callable type `Callable<T,P>`	`Callable<T[],P>`
Spread multiplication					
`lhs ** rhs`	scale	`rhs.scale(lhs)`	`Y`	`Scalable<X,Y>`	`X`

There are two special cases related to sequences. A type `X` is a *sequence type* if `X` is a subtype of `Sequential<Anything>`.

For any sequence type `X` and integer `n`, we can form the *nth tail type, Xn, of X* as follows:

- for every `i<=0`, `Xi` is `X`, and
- for every `i>0`, if `Xi` has the principal instantiation `Tuple<Ui,Fi,Yi>` then `X(i+1)` is `Yi`, or, otherwise, `X(i+1)` is `Xi`.

For any sequence type `X` and integer `n`, we can form the *nth element type, En, of X* as follows:

- if `Xn` has the principal instantiation `Tuple<Un,Fn,Yn>` then `En` is `Fn`, or, otherwise, `Xn` has the principal instantiation `Sequential<Fn>` and `En` is `Fn?`.

Then the two special cases are:

- The type of an expression of form `x[i...]` where `x` is of tuple type `X` and `n` is an integer literal is `Xn`.
- The type of an expression of form `x[i]` where `x` is of tuple type `X` and `n` is an integer literal is `En`.

6.8.8. Operators for creating objects

These operators simplify the syntax for instantiating certain commonly used built-in types.

Table 6.7.

Example	Name	Definition	LHS type	RHS type	Return type
Range and entry constructors					
`lhs..rhs`	spanned range	`Range(lhs, rhs)`	`T given T satisfies Ordinal & Comparable<T>`	`T`	`Range<T>`
`lhs:rhs`	segmented range	`if (lhs<=0) [] else TODO`	`T given T satisfies Ordinal & Comparable<T>`	`Integer`	`T[]`
`lhs->rhs`	entry	`Entry(lhs, rhs)`	`U given U satisfies Object`	`V given V satisfies Object`	`Entry<U,V>`

6.8.9. Conditional operators

Two special operators allow emulation of the famous ternary operator of C-like languages.

Table 6.8.

Example	Name	Definition	LHS type	RHS type	Return type
Conditionals					
`lhs then rhs`	then	`if (lhs) then rhs else null`	`Boolean`	`T given T satisfies Object`	`T?`
`lhs else rhs`	else	`if (exists lhs) then lhs else rhs`	`U` such that `null is U`	`V`	`U&Object|V`

6.8.10. Arithmetic operators

These are the usual mathematical operations for all kinds of numeric values.

Table 6.9.

Example	Name	Definition	LHS type	RHS type	Return type
Increment, decrement					
`++rhs`	successor	`rhs=rhs.successor`		variable of type `Ordinal<T>`	`T`
`--rhs`	predecessor	`rhs=rhs.predecessor`		variable of type `Ordinal<T>`	`T`
`lhs++`	increment	`(++lhs).predecessor`	variable of type `Ordinal<T>`		`T`
`lhs--`	decrement	`(--lhs).successor`	variable of type `Ordinal<T>`		`T`

Example	Name	Definition	LHS type	RHS type	Return type
Numeric operators					
`+rhs`		`rhs.positiveValue`		`Invertable<I>`	`I`
`-rhs`	negation	`rhs.negativeValue`		`Invertable<I>`	`I`
`lhs + rhs`	sum	`lhs.plus(rhs)`	`Summable<X>`	`X`	`X`
`lhs - rhs`	difference	`lhs.minus(rhs)`	`Subtractable<X>`	`X`	`X`
`lhs * rhs`	product	`lhs.times(rhs)`	`Numeric<X>`	`X`	`X`
`lhs / rhs`	quotient	`lhs.divided(rhs)`	`Numeric<X>`	`X`	`X`
`lhs % rhs`	remainder	`lhs.remainder(rhs)`	`Integral<X>`	`X`	`X`
`lhs ^ rhs`	power	`lhs.power(rhs)`	`Exponentiable<X,Y>`	`Y`	`X`
Numeric assignment					
`lhs += rhs`	add	`lhs=lhs.plus(rhs)`	variable of type `Summable<N>`	`N`	`N`
`lhs -= rhs`	subtract	`lhs=lhs.minus(rhs)`	variable of type `Subtractable<N>`	`N`	`N`
`lhs *= rhs`	multiply	`lhs=lhs.times(rhs)`	variable of type `Numeric<N>`	`N`	`N`
`lhs /= rhs`	divide	`lhs=lhs.divided(rhs)`	variable of type `Numeric<N>`	`N`	`N`
`lhs %= rhs`	remainder	`lhs=lhs.remainder(rhs)`	variable of type `Integral<N>`	`N`	`N`

Arithmetic operators automatically widen from `Integer` to `Float` when necessary. If one operand expression is of static type `Integer`, and the other is of type `Float`, the operand of type `Integer` is widened to a `Float` in order to make the operator expression well-typed. Widening is performed by evaluating the attribute `float` defined by `Integer`.

Note: this is the only circumstance in the language where implicit type conversion occurs. In fact, it is more correct to view this behavior as an instance of operator overloading than as an implicit type conversion. Implicit widening does not occur when an expression of type `Integer` is merely assigned to the type `Float`, since such behavior would result in ambiguities when generics come into play.

6.8.11. Set operators

These operators provide traditional mathematical operations for sets.

Table 6.10.

Example	Name	Definition	LHS type	RHS type	Return type
Set operators					
`lhs | rhs`	union	`lhs.union(rhs)`	`Set<X>`	`Set<Y>`	`Set<X|Y>`
`lhs & rhs`	intersection	`lhs.intersection(rhs)`	`Set<X>`	`Set<Y>`	`Set<X&Y>`
`lhs ~ rhs`	complement	`lhs.complement(rhs)`	`Set<X>`	`Set<Object>`	`Set<X>`

Example	Name	Definition	LHS type	RHS type	Return type
Set assignment					
`lhs |= rhs`	union	`lhs=lhs|rhs`	variable of type `Set<X>`	`Set<X>`	`Set<X>`
`lhs &= rhs`	intersection	`lhs=lhs&rhs`	variable of type `Set<X>`	`Set<Object>`	`Set<X>`
`lhs ~= rhs`	comple-ment	`lhs=lhs~rhs`	variable of type `Set<X>`	`Set<Object>`	`Set<X>`

6.9. Operator-style member and invocation expressions

An member expression or a method invocation with a single positional argument may be written using an *operator-style* syntax. This syntax has an extremely low precedence, just above the precedence of the assignment operator, is not associative, and does not form a legal expression statement.

```
0..max by step
```

In an *operator-style invocation expression*, the invoked method name and optional type arguments occur in an infix location between two expressions. The first expression is interpreted as the receiver expression, and the second expression is interpreted as a positional argument to the first parameter of the method.

```
OperatorInvocation: ValueExpression MemberName TypeArguments? Expression
```

The semantics of this syntax are identical to ordinary invocation expressions, as defined in §6.6 Invocation expressions.

In an *operator-style member expression*, the member name and optional type arguments occur in a postfix location after an expression. The expression is interpreted as the receiver expression.

```
OperatorMemberExpression: ValueExpression MemberName TypeArguments?
```

The semantics of this syntax are identical to ordinary member expression, as defined in §6.5.2 Member expressions.

TODO: Should we rather use this syntax as a sugar for invocation of a toplevel function?

6.10. Metamodel expressions

A *metamodel expression* is a reference to a type, a class, a function, or a value. It evaluates to a metamodel object whose static type captures the type itself, the callable type of the class, the callable type of the function, or the type of the value, respectively.

```
Meta: TypeMeta | BaseMeta | MemberMeta
```

A *type metamodel expression* is a type, as defined by §3.2 Types, surrounded by backticks.

```
TypeMeta: "`" Type "`"
```

The type may or may not be a reference to a class or interface.

```
Class<Person,[Name]> personClass = `Person`;
```

```
Interface<List<String>> stringListInterface = `List<String>`;
```

```
UnionType<Integer|Float> numberType = `Number`;
```

```
Type<Element> elementType = `Element`;
```

A *base metamodel expression* is a member name, with an optional list of type arguments, surrounded by backticks.

```
BaseMeta: "`" MemberName TypeArguments? "`"
```

A base metamodel expression is a reference to a value or function. The referenced declaration is determined according to §5.1.7 Unqualified reference resolution.

A *member metamodel expression* is a type, as defined by §3.2 Types, followed by a member name, with an optional list of type arguments, surrounded by backticks.

```
MemberMeta: "`" (QualifiedType|GroupedType) "." MemberName TypeArguments? "`"
```

A member metamodel expression is a reference to an attribute or method of the type. The member is resolved as a member of the type according to §5.1.8 Qualified reference resolution.

```
Function<Float,[{Float+}]> sumFunction = `sum<Float>`;
```

```
Attribute<Person,String> personNameAttribute = `Person.name`;
```

```
Method<Person,Anything,[String]> personSayMethod = `Person.say`;
```

Type argument inference is impossible in a metamodel expression, so type arguments must be explicitly provided for every generic declaration.

6.10.1. Type of a metamodel expression

The type of a metamodel expression depends upon the kind of declaration referenced:

- for a toplevel value of type `R`, the type is `Value<R>`,
- for a toplevel function of callable type `Callable<R,P>`, the type is `Function<R,P>`,
- for a toplevel class of callable type `Callable<R,P>`, the type is `Class<R,P>`,
- for a class nested in a block of callable type `Callable<R,P>`, the type is `Class<R,Nothing>`, and
- for a toplevel interface or interface nested in a block of type `R`, the type is `Interface<R>`.

Furthermore, given a member of a type `T`:

- for an attribute of type `R`, the type is `Attribute<T,R>`,
- for a method of callable type `Callable<R,P>`, the type is `Method<T,R,P>`,
- for a member class of callable type `Callable<R,P>`, the type is `MemberClass<T,R,P>`, and
- for a nested interface of type `R`, the type is `MemberInterface<T,R>`.

Finally:

- for a union type `T`, the type is `UnionType<T>`,
- for an intersection type `T`, the type is `IntersectionType<T>`,
- for the type `Nothing`, the type is `Type<Nothing>`, and
- for a type parameter `T`, the type is `Type<T>`.

If a type alias occurs inside a typed metamodel expression, it is replaced by its definition, after substituting type arguments, before determining the type of the metamodel expression.

6.11. Reference expressions

A *reference expression* is a reference to a program element and evaluates to a detyped metamodel of the program element. Reference expressions are used primarily in annotations, especially the documentation annotations listed in §7.4.2 Documentation. A reference expression may refer to:

- a class, interface, type alias, or type parameter,
- a function or value, or
- a package or module.

```
Dec: TypeDec | MemberDec | PackageDec | ModuleDec
```

6.11.1. Declaration references

A *class reference expression*, *interface reference expression*, *alias reference expression*, or *type parameter reference expression* is a series of initial uppercase identifiers, with the keyword `class`, `interface`, `alias`, or `given`, respectively, surrounded by backticks.

```
TypeDec: "`" ("class" | "interface" | "alias" | "given") (TypeName ".")* TypeName "`"
```

A *value reference expression* or *function reference expression* is an initial lowercase identifier, qualified by a list of initial uppercase identifiers, with the keyword `value` or `function`, surrounded by backticks.

```
MemberDec: "`" ("value" | "function") (TypeName ".")* MemberName "`"
```

A reference expression is a reference to a declaration. The referenced declaration is determined according to §5.1.7 Unqualified reference resolution and §5.1.8 Qualified reference resolution. The kind of the referenced declaration must match the kind of reference indicated by the keyword.

```
ClassDeclaration personClass = `class Person`;
```

```
InterfaceDeclaration stringListInterface = `interface List`;
```

```
AliasDeclaration numberAlias = `alias Number`;
```

```
TypeParameter elementTypeParameter = `given Element`;
```

```
ValueDeclaration personNameAttribute = `value Person.name`;
```

```
FunctionDeclaration personSayMethod = `function Person.say`;
```

6.11.2. Package and module references

A *package reference expression* is a package name, as defined by §4.1.2 Packages, with the keyword `package`, surrounded by backticks.

```
PackageDec: "`" "package" FullPackageName "`"
```

The package name must refer to a package from which an `import` statement in the same compilation unit may import declarations, as defined by §4.2 Imports.

```
Package modelPackage = `package ceylon.language.meta.model`;
```

A *module reference expression* is a module name, as defined by §9.3.1 Module names and version identifiers, with the keyword `module`, surrounded by backticks.

```
ModuleDec: "`" "module" FullPackageName "`"
```

The module name must refer to the module to which the compilation unit belongs, as specified by §9.2 Source layout, or to a module imported by the module to which the compilation unit belongs, as defined by §9.3.12 Module descriptors.

```
Module languageModule = `module ceylon.language`;
```

6.11.3. Type of a reference expression

The type of a reference expression depends upon the kind of program element referenced:

- for a module, the type is `Module`,
- for a package, the type is `Package`,
- for a value, the type is `ValueDeclaration`,
- for a function, the type is `FunctionDeclaration`,
- for a type parameter, the type is `TypeParameter`,
- for a type alias declared using the keyword `alias`, the type is `AliasDeclaration`,
- for a class or class alias, the type is `ClassDeclaration`, and
- for an interface or interface alias, the type is `InterfaceDeclaration`.

Chapter 7. Annotations

Annotations allow information to be attached to a declaration or assertion, and recovered at runtime via the use of the Ceylon metamodel. Annotations are used to specify:

- information used by the compiler while typechecking the program,
- API documentation for the documentation compiler,
- serialization of a class, and
- information needed by generic frameworks and libraries.

7.1. Annotations of program elements

Annotations occur at the very beginning of a declaration or assertion, in an *annotation list*.

```
"The user login action"
by ("Gavin King",
    "Andrew Haley")
throws (`class DatabaseException`,
        "if database access fails")
see (`function LogoutAction.logout`)
scope (session)
action { description="Log In"; url="/login"; }
shared deprecated
```

7.1.1. Annotation lists

An annotation is an initial lowercase identifier, optionally followed by an argument list.

```
Annotation: MemberName Arguments?
```

The annotation name is a reference to an annotation constructor, resolved according to §5.1.7 Unqualified reference resolution.

A list of annotations does not require punctuation between the individual annotations in the list. An annotation list may begin with a string literal, in which case it is interpreted as the argument of a `doc` annotation.

```
Annotations: StringLiteral? Annotation*
```

Every annotation is an invocation expression, as defined by §6.6 Invocation expressions, of an annotation constructor. The annotation name is interpreted as a base expression, as defined in §6.5.1 Base expressions.

7.1.2. Annotation arguments

For an annotation with no arguments, the argument list may be omitted, in which case the annotation is interpreted as having an empty positional argument list. Otherwise, the annotation argument list may be specified using one of two forms:

- Using a positional argument list, as defined in §6.6.7 Positional argument lists:

```
doc ("the name") String name;
```

- Using a named argument list, as defined in §6.6.8 Named argument lists:

```
doc { description="the name"; } String name;
```

As a special case, the name of the `doc` annotation and the parenthesis around its argument may be ommitted if it is the first annotation in an annotation list.

```
"the name" String name;
```

Operator expressions, member expressions, self references, anonymous functions, comprehensions, and string templates are not permitted in an annotation argument. Every base expression in an annotation argument must be a value reference to an anonyous class instance of an enumerated type, or must occur in a direct instantiation expression for an annotation type.

A named argument to an annotation may not be an inline function, value, or anonymous class.

7.2. Annotation definition

Annotations are typesafe.

- An *annotation constructor* defines the schema of an annotation as it appears at a program element.
- An *annotation type* defines constraints upon which program elements can bear the annotation, and an API for accessing the information carried by an annotation.

7.2.1. Annotation constructors

An *annotation constructor* is a toplevel function that defines an annotation schema. An annotation constructor must be annotated `annotation`. An annotation constructor may not declare type parameters.

Each parameter of an annotation constructor must have one of the following types:

- `Integer`, `Float`, `Character`, or `String`,
- an enumerated type whose cases are all anonymous classes, such as `Boolean`,
- a subtype of `Declaration` in `ceylon.language.meta.declaration`,
- an annotation type,
- `{T*}` or `[T*]` where `T` is a legal annotation constructor parameter type, or
- any tuple type whose element types are legal annotation constructor parameter types.

A parameter of an annotation constructor may be variadic.

An annotation constructor must simply instantiate and return an instance of an annotation type. The body of an annotation constructor may not contain multiple statements. Operator expressions, member expressions, self references, anonymous functions, comprehensions, and string templates are not permitted in the definition of an annotation constructor. Every base expression in the body of an annotation constructor must be a reference to a parameter of the annotation constructor or to an anonyous class instance of an enumerated type, or must occur in a direct instantiation expression for an annotation type.

A named argument appearing in the definition of an annotation constructor may not be an inline function, value, or anonymous class.

```
shared annotation Scope scope(ScopeType s) => Scope(s);
```

```
shared annotation Todo todo(String text) => Todo(text);
```

An annotation constructor parameter may have a default argument, which must be a legal annotation argument.

The return type of an annotation constructor must be a constrained annotation type, as defined below in §7.2.3 Constrained annotation types.

A user-defined annotation constructor may not return the same annotation type as one of the modifiers listed below in §7.4.1 Declaration modifiers.

Note: in future releases of the language we will let an annotation constructor return a sequence or tuple of annotation type instances.

7.2.2. Annotation types

Annotation constructors produce instances of *annotation types*. An annotation type is a class annotated `annotation`. An annotation type may not be a generic type with type parameters. An annotation type must have an empty initializer section.

Note: currently every annotation type must be a `final` class which directly extends `Basic` in `ceylon.language`.

Each initializer parameter of an annotation type must have one of the following types:

- `Integer`, `Float`, `Character`, or `String`,
- an enumerated type whose cases are all anonymous classes, such as `Boolean`,
- a subtype of `Declaration` in `ceylon.language.meta.declaration`,
- an annotation type,
- `{T*}` or `[T*]` where `T` is a legal annotation parameter type, or
- any tuple type whose element types are legal annotation parameter types.

An initializer parameter of an annotation type may be variadic.

An initializer parameter of an annotation type may have a default argument, which must be a legal annotation argument.

7.2.3. Constrained annotation types

A *constrained annotation type* is an annotation type that is a subtype of `OptionalAnnotation` or `SequencedAnnotation` defined in the package `ceylon.language`.

- If `A` is a subtype of `OptionalAnnotation`, at most one annotation of annotation type `A` may occur at a given program element.
- If `A` is a subtype of `SequencedAnnotation`, multiple annotations of annotation type `A` may occur at a given program element.
- If `A` is a subtype of `OptionalAnnotation<A,P>`, or `SequencedAnnotation<A,P>` then an annotation of annotation type `A` may not occur at a program element whose reference expression type, as defined in §6.11.3 Type of a reference expression, is not assignable to `P`.

```
shared final annotation class Scope(shared ScopeType scope)
        satisfies OptionalAnnotation<Scope,ClassOrInterfaceDeclaration> {
    string => (scope==request then "request")
         else (scope==session then "session")
         else (scope==application then "application")
         else nothing;
}
```

```
shared final annotation class Todo(String text)
        satisfies SequencedAnnotation<Todo> {
    string => text;
}
```

Note: it is perfectly acceptable for multiple annotation constructors to return the same annotation type.

7.3. Annotation values

An *annotation value* is the value returned when an annotation constructor is invoked. We may obtain the annotation values of all annotations of a given annotation type that occur at a given program element by passing the annotation type and program element metamodel reference to the method `annotations()` defined in the package `ceylon.language.model`.

```
Scope scope = annotations(`Scope`, `class Person`) else Scope(request);
```

```
Todo[] todos = annotations(`Todo`, `function method`);
```

7.4. Language annotations

Certain important annotations are predefined in the module `ceylon.language`.

7.4.1. Declaration modifiers

The following annotations, called *modifiers*, are compiler instructions that affect the compilation process:

- `shared` specifies that a declaration is visible outside of the package or body in which it occurs, or that a package is visible outside the module it belongs to.
- `abstract` specifies that a class cannot be instantiated.
- `formal` specifies that a member does not specify an implementation and must therefore be refined by every concrete subclass.
- `default` specifies that a method, attribute, or member class may be refined by subtypes.
- `actual` indicates that a method, attribute, or member type refines a method, attribute, or member type defined by a supertype.
- `variable` specifies that a value may be assigned multiple times.
- `late` disables definite initialization checking for a reference, allowing the reference to be initialized after the initializer of the class to which it belongs has already completed.
- `native` specifies that a program element is actually implemented in a different language, and that the program element should be ignored by the Ceylon compiler backend.
- `deprecated` indicates that a value, function or type is deprecated. It accepts an optional `String` argument. The compiler produces a warning when compiling code that depends upon a deprecated program element.
- `final` specifies that a class may not be extended.
- `annotation` specifies that a class is an annotation type, or that a toplevel function is an annotation constructor.

The following annotation is a hint to the compiler that lets the compiler optimize compiled bytecode for non-64 bit architectures:

- `small` specifies that a value of type `Integer`, `Integer` or `Float` contains 32-bit values.

By default, `Integer` and `Float` are assumed to represent 64-bit values, as specified in §8.5.2 Numeric operations.

Note that `small` is not yet supported in Ceylon 1.0.

7.4.2. Documentation

The following annotations are instructions to the documentation compiler:

- `doc` specifies the description of a program element, in Markdown format text.
- `by` specifies the authors of a program element.
- `license` specifies the URL of the license under which a module or package is distributed.
- `see` specifies a related member or type.
- `throws` specifies a thrown exception type.
- `tagged` specifies classifying named tags.

The `String` arguments to the `deprecated`, `doc`, `throws` and `by` annotations are parsed by the documentation compiler as Markdown-format content.

These annotations are all defined in the package `ceylon.language`.

7.5. Serialization

TODO: Define how serialization works.

Chapter 8. Execution

A Ceylon program executes in a virtual machine environment, either:

- a Java Virtual Machine (JVM), or
- a JavaScript virtual machine.

In future, other virtual machine architectures may be supported.

Despite the obvious differences between the respective languages that these virtual machines were designed for, they share very much in common in terms of runtime semantics, including common notions such as object identity, primitive value types, exceptions, garbage collection, dynamic dispatch, and pass by reference.

Ceylon abstracts away many of the differences between these platforms, and reuses what is common between them. Inevitably there are some differences that can't reasonably be hidden from the Ceylon program, and the programmer must take these differences into consideration.

In Ceylon, every value is a reference to an instance of a class, except within a `dynamic` block, where a value with no type may be a reference to an object which is not an instance of a class.

Note: the semantics of objects without classes is platform-dependent and outside the scope of this specification.

8.1. Object instances, identity, and reference passing

An *object* is a unique identifier, together with a reference to a class, its type arguments, and a persistent value for each reference declared by the class (including inherited references). The object is said to be an *instance* of the class.

A *value* is a reference to an object (a copy of its unique identifier). At a particular point in the execution of the program, every reference of every object that exists, and every initialized reference of every function, getter, setter, or initializer that is currently executing has a value. Furthermore, every time an expression is executed, it produces a value.

Two values are said to be *identical* if they are references to the same object—if they hold the same unique identifier. The program may determine if two values of type `Identifiable` are identical using the `===` operator defined in §6.8.2 Operator definition. It may not directly obtain the unique identifier (which is a purely abstract construct). The program has no way of determining the identity of a value which is not of type `Identifiable`.

Given a value, the program may determine if the referenced object is *assignable to a certain type* using the `is` operator. The object is assignable to the given type if the applied type formed by its class and type arguments is a subtype of the given type according to the type system defined in Chapter 3, *Type system*. (Therefore, the Ceylon runtime must be capable of reasoning about subtyping.)

Invocation of a function or instantiation of a class results in execution of the function body or class initializer with parameter values that are copies of the value produced by executing the argument expressions of the invocation, and a reference to the receiving instance that is a copy of the value produced by executing the receiver expression. The value produced by the invocation expression is a copy of the value produced by execution of the `return` directive expression.

```
Person myself(Person me) { return me; }
Person p = ...;
assert (myself(p)===p); //assertion never fails
```

```
Semaphore s = Semaphore();
this.semaphore = s;
assert (semaphore===s); //assertion never fails
```

A new object is produced by execution of a class instantiation expression. The Ceylon compiler guarantees that if execution of a class initializer terminates with no uncaught exception, then every reference of the object has been initialized with a well-defined persistent value. The value of a reference is initialized for the first time by execution of a specifier or assignment expression. Every class instantiation expression results in an object with a new unique identifier shared by no other existing object. The object exists from the point at which execution of its initializer terminates. *Conceptually*, the object exists until execution of the program terminates.

In practice, the object exists at least until the point at which it is not reachable by recursively following references from

any function, getter, setter, or initializer currently being executed, or from an expression in a statement currently being executed. At this point, its persistent values are no longer accessible to expressions which subsequently execute and the object may be destroyed by the virtual machine. There is no way for the program to determine that an object has been destroyed by the virtual machine (Ceylon does not support finalizers).

8.1.1. Value type optimizations

As a special exception to the rules defined above, the compiler is permitted to emit bytecode or compiled JavaScript that produces a new instance of certain types in the module `ceylon.language` without execution of the initializer of the class, whenever any expression is evaluated. These types are: `Integer`, `Float`, `Character`, `Range`, `Entry`, `String`, `Array`, and `Tuple`. Furthermore, it is permitted to use such a newly-produced instance as the value of the expression, as long as the newly-produced instance is equal to the value expected according to the rules above, as determined using the == operator.

Therefore, the types listed above directly extend `Object` instead of `Basic`, and are not `Identifiable`.

Note: this does no justice at all to our compiler. Actually the compiler infrastructure already supports value type optimization for user-defined types, though we have not yet exposed this functionality as part of the language.

8.1.2. Type argument reification

Type arguments, as defined in §3.6 Generic type arguments, are *reified* in Ceylon. An instance of a generic type holds a reference to each of its type arguments. Therefore, the following are possible in Ceylon:

- testing the runtime value of a type argument of an instance, for example, `objectList is List<Person>` or `case (is List<Person>)`,
- filtering exceptions based on type arguments, for example, `catch (NotFoundException<Person> pnfe)`, and
- testing the runtime value of an instance against a type parameter, for example `x is Key`, or against a type with a type parameter as an argument, for example, `objectList is List<Element>`.
- obtaining a `Type` object representing a type with type arguments, for example, `` `List<Person>` ``,
- obtaining a `Type` object representing the runtime value of a type parameter, for example, `` `Element` ``, or of a type with a type parameter as an argument, for example, `` `List<Element>` ``, and
- obtaining a `Type` object representing the runtime value of a type argument of an instance using reflection, for example, `type(objectList).typeArguments.first`.

At runtime, all types are *concrete types* formed by:

- recursively replacing all type aliases, class aliases, and interface aliases with their definitions, which is always possible according to §3.2.10 Type alias elimination, and
- recursively replacing all type parameters with their type arguments

in any type that appears in an expression or condition.

Therefore, every type parameter refers, at runtime, to a concrete type that involves no type aliases or type parameters. In particular, the type arguments held by an instance of a generic class are concrete types.

This program prints `String[]`.

```
class Generic<out T>(T t) { string=>`T`.string; }
Generic<{S*}> gen<S>(S* ss) => Generic(ss);
void run() {
    print(gen("hello", "world"));
}
```

The runtime is generally permitted, as an optimization, to return a more precise type in place of a less precise type when a type parameter is evaluated. This program may print `String` instead of `Object`, even though `Object` is the type argument inferred at compile time.

```
class Generic<out T>(T t) { string=>`T`.string; }
```

```
Generic<Object> gen(Object o) => Generic(o);
void run() {
    print(gen("hello"));
}
```

8.2. Sequential execution and closure

Ceylon programs are organized into bodies, as defined in §5.1 Block structure and references, containing statements which are executed sequentially and have access to declarations which occur in the surrounding lexical context and to persistent values held by references, as defined in §4.8.1 References, declared in the surrounding lexical context.

Note: for the purposes of this section, an interface body is, strictly speaking, a trivial case of a body which contains no statements or persistent values, but we're primarily concerned with blocks and class bodies.

The statements and non-lazy specifiers that directly occur in a body are executed sequentially in the lexical order in which they occcur. Execution of a body begins at the first statement or non-lazy specifier. Execution of a block terminates when the last statement or non-lazy specifier of the body finishes executing, or when a control directive that terminates the block is executed, or when an exception is thrown by an evaluation, assignment, invocation, or instantiation.

8.2.1. Frames

When execution of a body begins, a *frame* is created. For each reference whose declaration directly occurs in the body, the frame has a value, which may or may not be initialized. The value may be initialized or assigned during execution of the body.

While a body is executing, all values held in the frame are considered accessible. An evaluation, assignment, invocation, or instantiation may result in a pause in execution of the body while the called getter, setter, function, or class is executed or instantiated. However, the frame associated with the calling body is retained and values held in the frame are still considered accessible. When execution of the body resumes, the frame is restored.

When execution of a body terminates, the frame may or may not become inaccessible. In the case of a class body, if the initializer terminates with no thrown exception, the frame and its values become a new instance of the class, are associated with the newly created unique identifier, and remain accessible while this object is itself accessible. In the case of any other kind of body, or in the case that an initializer throws an exception, the frame and its values may remain accessible if:

- a reference to a function or class declared within the body is accessible,
- an instance of a class declared within the body is accessible, or
- an instance of a comprehension declared within the body is accessible.

Otherwise, the frame becomes inaccessible and may be destroyed.

The principle of *closure* states that a nested body always has access to a frame for every containing body. The set of *current instances* of containing classes and *current frames* of containing blocks forms the closure of a nested declaration.

8.2.2. Current instances and current frames

A frame may be the *current frame* for a body. When the body is executing, the created frame is the current frame. When execution of the body terminates, the created frame is no longer the current frame. Invocation or evaluation of a member of a class or interface, invocation of a callable reference or anonymous function, or evaluation of the values produced by a comprehension may result in the frame being restored as the current frame.

A class instance, callable reference, anonymous function reference, or comprehension instance packages a reference to a frame for each body containing the program element, as specified below. When a member of the class instance is invoked or evaluated, when the callable reference or anonymous function is invoked, or when the comprehension instance produces a value, these frames are restored as the current frames of the associated bodies. When the invocation or evaluation terminates, or when the comprehension value has been produced, these frames are no longer current frames.

The value associated with a value reference in the current frame of the body to which the value reference belongs is called the *current value* of the value reference.

If a frame is the current frame for a class or interface body, we call it the *current instance* of the class or interface.

TODO: in the following two sections, account for callable references, anonymous function references, and comprehension instances.

8.2.3. Current instance of a class or interface

If a statement is occurs directly or indirectly inside a class or interface body, then there is always a current instance of the class or interface when the statement is executed. The current instance is determined as follows:

- For a statement that occurs sequentially, as defined by §5.1 Block structure and references, in the body of the class, the current instance is the new instance being initialized.
- For a statement that occurs sequentially in the body of a member of the class or interface, the current instance is the receiving instance of the base or member expression that resulted in a reference to the member.
- For a statement that occurs sequentially in the body of a nested class or interface that occurs in the body of the class or interface, the current instance is the same object that was the current instance when the initializer of the current instance of the nested class or interface was executed.
- Otherwise, for any other statement that occurs sequentially in the body of a declaration that occurs in the body of the class or interface, the current instance is the same object that was the current instance when the base member expression that resulted in a reference to the declaration was executed.

Here, `innerObject` is the current instance of `Inner` when `member()` is executed, and `outerObject` is the current instance of `Outer`:

```
Outer outerObject = Outer();
Inner innerObject = outerObject.Inner();
innerObject.member();
```

8.2.4. Current frame of a block

If a statement occurs directly or indirectly inside a block, then there is always a current frame of the block when the statement is executed. The current frame is determined as follows:

- If the statement occurs sequentially, as defined by §5.1 Block structure and references, in the block, the current frame is the frame associated with the current execution of the block.
- For a statement that occurs sequentially in the body of a nested class or interface that occurs in the block, the current frame is the same frame that was the current frame when the initializer of the current instance of the nested class or interface was executed.
- Otherwise, for any other statement that occurs sequentially inside the body of a declaration that occurs in the block, and the current frame is the frame that was the current frame when the base member expression that resulted in a reference to the declaration was executed.

In each of the following code fragments, `result` refers to the value `"hello"`:

```
String()() outerMethod(String s) {
    String() middleMethod() {
        String innerMethod() => s;
        return innerMethod;
    }
    return middleMethod;
}

String middleMethod()() => outerMethod("hello");
String innerMethod() => middleMethod();
String result = innerMethod();
```

```
Object outerMethod(String s) {
    object middleObject {
        shared actual String string => s;
    }
    return middleObject;
```

```
}

Object middleObject = outerMethod("hello");
String result = middleObject.string;
```

8.2.5. Initialization

When an instance is instantiated, its initializer is executed, and the initializer for every class it inherits is executed. For a class `C`:

- First, the initializer of `Object` defined in `ceylon.language` is executed. (This initializer is empty and does no work.)
- For each superclass `X` of `C`, there is exactly one other superclass `Y` of `C` that directly extends `X`. When execution of the initializer of `X` terminates without a thrown exception, execution of the initializer of `Y` begins.
- Finally, when execution of the initializer of `C` terminates without a thrown exception, the new instance of `C` is fully-initialized and made accessible to the calling code.

If any initializer in the class heirarchy terminates with a thrown exception, initialization terminates and the incompletely-initialized instance never becomes accessible.

Each initializer produces a frame containing values for each reference declared be the corresponding class. These frames are aggregated together to form the new instance of the class `C`.

Note: since interfaces don't have initializers, the issue of "linearization" of supertypes simply never arises in Ceylon. There is a natural, well-defined initialization ordering.

8.2.6. Class instance optimization

As an exception to the above, the compiler is permitted to destroy a persistent value associated with a class instance when the class initializer terminates, potentially rendering inaccessible the instance identified by the value, if it can determine that the persistent value will never be subsequently accessed by the program.

This optimization is the only source of a distinction between a "field" of a class and a "local variable" of its initializer. There is no way for a program to observe this distinction.

8.2.7. Execution of expression and specification statements

When an expression statement is executed, the expression is evaluated.

When a non-lazy specification statement is executed, the specified expression is evaluated, and the resulting value assigned to the specified reference within the current frame or current instance associated with the body to which the specified reference belongs.

When a lazy specification statement is executed, the specified expression is associated with the specified reference within the current frame or current instance associated with the body to which the specified reference belongs. Subsequent evaluation or invocation of the reference for this current frame or current instance may result in evaluation of the specified expression, in which case the expression is evaluated within this current frame or current instance.

8.2.8. Execution of control directives

Execution of a control directive, as specified in §5.2.2 Control directives, terminates execution of the body in which it occurs, and possibly of other containing bodies.

- A `return` directive that occurs sequentially in the body of a function, getter, setter, or class initializer terminates execution of the body of the function, getter, setter, or class initializer and of all intervening bodies. Optionally, it determines the return value of the function or getter.
- A `break` directive terminates execution of the body of the most nested containing loop in which it occurs sequentially, and of all intervening bodies. Additionally, it terminates execution of the loop.
- A `continue` directive terminates execution of the body of the most nested containing loop in which it occurs sequen-

tially, and of all intervening bodies. It does not terminate execution of the loop.

- A `throw` directive that occurs sequentially in the body of a function, getter, setter, or class initializer terminates execution of the body of the function, getter, setter, or class initializer and of all intervening bodies, and, furthermore, the exception propagates to the caller, as defined below, unless there is an intervening `try` with a `catch` clause matching the thrown exception, in which case it terminates execution of the body of the `try` statement and all intervening bodies, and execution continues from the body of the `catch` clause.

8.2.9. Exception propagation

If execution of an evaluation, assignment, invocation, or instantiation terminates with an exception thrown, the exception propagates to the calling code, and terminates execution of the body of the function, getter, setter, or class initializer in which the expression involving the evaluation, assignment, invocation, or instantiation sequentially occurs, and of all intervening bodies, and, furthermore, the exception propagates to the caller unless there is an intervening `try` with a `catch` clause matching the thrown exception, in which case it terminates execution of the body of the `try` statement and all intervening bodies, and execution continues from the body of the `catch` clause.

8.2.10. Initialization of toplevel references

A toplevel reference has no associated frame. Instead, the lifecycle of its persistent value is associated with the loading and unloading of a module by the module runtime. The first time a toplevel reference is accessed following the loading of its containing module, its initializer expression is evaluated, and the resulting value is associated with the reference. This association survives until the toplevel reference is reassigned, or until the module is unloaded by the module runtime.

Initialization of a toplevel reference may result in recursive initialization of other toplevel references. Therefore, it is possible that a cycle could occur where evaluation of a toplevel reference occurs while evaluating its initializer expression. When this occurs, an `InitializationException` is thrown.

8.3. Execution of control structures and assertions

Control structures, as specified in §5.3 Control structures and assertions, are used to organize conditional and repetitive code within a body. Assertions are essentially a sophisticated sort of control directive, but for convenience are categorized together with control structures.

8.3.1. Evaluation of condition lists

Execution of an `if`, `while`, or `assert` requires evaluation of a condition list, as defined in §5.3.3 Control structure conditions.

To determine if a condition list is satisfied, its constituent conditions are evaluated in the lexical order in which they occur in the condition list. If any condition is not satisfied, none of the subsequent conditions in the list are evaluated.

- A boolean condition is satisfied if its expression evaluates to `true` when the condition is evaluated.

For any other kind of condition, the condition is satisfied if its value reference or expression evaluates to an instance of the required type when the condition is evaluated:

- for an assignability condition, the condition is satisfied if the expression evaluates to an instance of the specified type when the control structure is executed,
- for an existence condition, the condition is satisfied unless the expression evaluates to `null` when the control structure is executed, or
- for a nonemptiness expression, the condition is satisfied unless the expression evaluates to an instance of `[]|Null` when the control structure is executed.

A condition list is satisfied if and only if all of its constituent conditions are satisfied.

8.3.2. Validation of assertions

When an assertion, as specified in §5.3.11 Assertions, is executed, its condition list is evaluated. If the condition list is not satisfied, an exception of type `AssertionFailure` in `ceylon.language` is thrown.

The information carried by the `AssertionFailure` includes:

- the text of the Ceylon code of the condition that failed,
- the message specified by the `doc` annotation of the assertion, if any.

8.3.3. Execution of conditionals

The `if/else` and `switch/case/else` constructs control conditional execution.

When the `if/else` construct, specified in §5.3.6 if/else, is executed, its condition list is evaluated. If the condition list is satisfied, the `if` block is executed. Otherwise, the `else` block, if any, is executed, or, if the construct has an `else if`, the child `if` construct is executed.

When a `switch/case/else` construct, specified in §5.3.7 switch/case/else, is executed, its `switch` expression is evaluated to produce a value. The value is guaranteed to *match* at most one `case` of the `switch`. If it matches a certain case, then that `case` block is executed. Otherwise, `switch` is guaranteed to have an `else`, and so the `else` block is executed.

The value produced by the `switch` expression matches a case if either:

- the case is a list of literal values and value references the value is identical to one of the value references in the list or equal to one of the literal values in the list, or if
- the case is an assignability condition of form `case (is V)` and the value is an instance of `V`.

8.3.4. Execution of loops

The `for/else` and `while` loops control repeated execution.

When a `while` construct, specified in §5.3.9 while, is executed, the loop condition list is evaluated repeatedly until the first time the condition list is not satisfied, or until a `break`, `return`, or `throw` directive that terminates the loop is executed. Each time the condition is satisfied, the `while` block is executed.

When a `for/else` construct, specified in §5.3.8 for/else, is executed:

- the iterated expression is evaluated to produce an an instance of `Iterable`,
- an `Iterator` is obtained by calling `iterator()` on the iterable object, and then
- the `for` block is executed once for each value of produced by repeatedly invoking the `next()` method of the iterator, until the iterator produces the value `finished`, or until a `break`, `return`, or `throw` directive that terminates the loop is executed.

Note that:

- if the iterated expression is also of type `X[]`, the compiler is permitted to optimize away the use of `Iterator`, instead using indexed element access.
- if the iterated expression is a range constructor expression, the compiler is permitted to optimize away creation of the `Range`, and generate the indices using the `successor` operation.

We say that the loop *exits early* if it ends via execution of a `break`, `return`, or `throw` directive. Otherwise, we say that the loop *completes* normally.

If the loop completes normally, the `else` block is executed. Otherwise, if the loop exists early, the `else` block is not executed.

8.3.5. Exception handling

When a `try/catch/finally` construct, specified in §5.3.10 try/catch/finally, is executed:

- the resource expressions, if any, are evaluated in the order they occur, and then `open()` is called on each resulting resource instance, in the same order, then
- the `try` block is executed, then
- `close()` is called on the resource instances, if any, in the reverse order that the resource expressions occur, with the exception that propagated out of the `try` block, if any, then
- if an exception did propagate out of the `try` block, the first `catch` block with a variable to which the exception is assignable, if any, is executed, and then
- the `finally` block, if any, is executed, even in the case where an exception propagates out of the whole construct.

TODO: Specify what happens if `close()` throws an exception? Same semantics as Java with "suppressed" exceptions.

8.3.6. Dynamic type checking

Inside a `dynamic` block, a situation might occur that requires dynamic type checking, as specified in §5.3.12 Dynamic blocks. It is possible that:

- the value to which an expression with no type evaluates at execution time might not be an instance of the type required where the expression occurs,
- in particular, the value to which a `switch` expression with no type evaluates at execution time might be an instance of a type not covered by the `cases` of a `switch` with no `else`, or
- a qualified or unqualified reference which does not refer to a statically typed declaration might not resolve to any declaration at all.

Whenever such a condition is encountered at runtime, an `AssertionException` is immediately thrown.

Note: in Ceylon 1.0, dynamic type checking is only supported on JavaScript virtual machines.

8.4. Evaluation, invocation, and assignment

Evaluation of an expression may result in:

- invocation of a function or instantiation of a class,
- evaluation of a value,
- instantiation of an instance of `Callable` that packages a callable reference, or
- assignment to a variable value.

8.4.1. Dynamic dispatch

Dynamic dispatch is the process of determing at runtime a member declaration based upon the runtime type of an object, which, as a result of subtype polymorphism, may be different to its static type known at compile time.

Any concrete class is guaranteed to have exactly one declaration of a member, either declared or inherited by the class, which refines all other declarations of the member declared or inherited by the class. At runtime, this member is selected.

There is one exception to this rule: member expressions where the receiver expression is of form `super` or `(super of Type)`, as defined in §6.3.3 super, are dispatched based on the static type of the receiver expression:

- Any invocation of a member of `super` is processed by the member defined or inherited by the supertype, bypassing any member declaration that refines this member declaration.

- Any invocation of a member of an expression of form `(super of Type)` is processed by the member defined or inherited by `Type`, bypassing any member declaration that refines this member declaration.

8.4.2. Evaluation

Evaluation of a value reference, as defined in §6.5.3 Value references, produces its current value. Evaluation of a callable reference, as defined in §6.5.4 Callable references, that does not occur as the primary of a direct invocation results in a new instance of `Callable` that packages the callable reference.

```
person.name
```

```
'/'.equals
```

When a value reference expression is executed:

- first, the receiver expression, if any, is evaluated to obtain a receiving instance for the evaluation, then
- the actual declaration to be invoked is determined by considering the runtime type of the receiving instance, if any, and then
- if the declaration is a reference, its persistent value is retrieved from the receiving instance, or
- otherwise, execution of the calling body pauses while the body of its getter is executed by the receiving instance, then,
- finally, when execution of the getter ends, execution of the calling body resumes.

The resulting value is the persistent value retrieved, or the return value of the getter, as specified by the `return` directive.

When a callable reference expression that does not occur as the primary of a direct invocation expression is executed:

- first, the receiver expression, if any, is evaluated to obtain a receiving instance for the evaluation, then
- the receiving instance, a reference to the declaration to be invoked, or a reference to the current frame or instance of every body that contains the referenced declaration are packaged together into an instance of `Callable`.

The resulting value is the instance of `Callable`. The concrete class of this instance is not specified here.

8.4.3. Assignment

Given a value reference, as defined in §6.5.3 Value references, to a variable, the assignment operator `=` assigns it a new value.

```
person.name = "Gavin"
```

When an assignment expression is executed:

- first, the receiver expression of the value reference expression is executed to obtain the receiving instance, then
- the actual declaration to be assigned is determined by considering the runtime type of the receiving instance, and then
- if the member is a reference, its persistent value is updated in the receiving instance, or
- otherwise, execution of the calling body pauses while the body of its setter is executed by the receiving instance with the assigned value, then,
- finally, when execution of the setter ends, execution of the calling body resumes.

8.4.4. Invocation

Evaluation of an invocation expression, as defined in §6.6 Invocation expressions, results in *invocation* of a function, or *instantiation* of a class. Every invocation has a callable expression:

- in a direct invocation, the callable expression is a callable reference, and
- in an indirect invocation, the callable expression is an instance of `Callable` that packages an underlying callable reference.

In either case, the callable expression determines the instance and member to be invoked.

```
print("Hello world!")
```

```
Entry(person.name, person)
```

When an invocation expression is executed:

- first, the callable expression is evaluated to obtain the receiving instance, then
- each listed argument or spread argument is evaluated in turn in the calling body, and
- if the argument list has a comprehension, a comprehension instance, as defined in §8.6 Evaluation of comprehensions, is obtained, and then
- the actual declaration to be invoked is determined by considering the runtime type of the receiving instance, if any, and then
- execution of the calling body pauses while the body of the function or initializer is executed by the receiving instance with the argument values, then
- finally, when execution of the function or initializer ends, execution of the calling body resumes.

A function invocation evaluates to the return value of the function, as specified by the `return` directive. The argument values are passed to the parameters of the method, and the body of the method is executed.

A class instantiation evaluates to a new instance of the class. The argument values are passed to the initializer parameters of the class, and the initializer is executed.

8.4.5. Evaluation of anonymous functions

When an anonymous function expression, as defined in §6.4 Anonymous functions, is evaluated, a reference to the function and a reference to the current frame or instance of every containing body are packaged into an instance of `Callable`. The instance of `Callable` is the resulting value of the expression. The concrete class of this instance is not specified here.

8.4.6. Evaluation of enumerations

Evaluation of an enumeration expression, as defined in §6.6.12 Iterable and tuple enumeration, results in creation of an iterable object or tuple.

```
{ "hello", "world" }
```

```
[ new, *elements ]
```

When an enumeration expression is executed:

- first, each listed argument or spread argument is evaluated in turn in the calling body, and
- if the argument list has a comprehension, a comprehension instance, as defined in §8.6 Evaluation of comprehensions, is obtained, and then
- the resulting argument values are packaged into an instance of `Iterable` or `Sequence`, and this object is the resulting value of the enumeration expression, unless
- there are no arguments, and no comprehension, in which case the resulting value of the enumeration expression is the object `empty`.

In the case of an iterable enumeration, the concrete class of the resulting value is not specified here. In the case of a tuple enumeration it is always `Tuple`, `Empty`, or `Sequence`.

8.4.7. Evaluation of spread arguments and comprehensions

A spread argument, as defined in §6.6.5 Spread arguments, produces multiple values by iterating the iterable object to which the spread operator is applied.

When a spread argument expression type is a subtype of `Sequential`, the behavior does not depend upon where the spread argument occurs:

- If it occurs as an argument, the sequence produced by evaluating the expression is passed directly to the parameter.
- If it occurs in an enumeration expression, the sequence produced by evaluating the expression is appended directly to the resulting iterable object or tuple.

On the other hand, when a spread argument expression type is not a subtype of `Sequential`, the behavior depends upon where the spread argument occurs:

- If it occurs as an argument to a variadic parameter in a positional argument list, the values produced by a spread argument are evaluated immediately and packaged into an instance of `Sequence` and passed to the variadic parameter, unless there are no values, in which case the object `empty` is passed to the variadic parameter.
- If it occurs as an argument to a parameter of type `Iterable` at the end of a named argument list, the iterable object produced by evaluating the expression is passed directly to the parameter.
- If it occurs in a tuple enumeration, the values produced by a spread argument are evaluated immediately and packaged into an instance of `Sequence` and appended to the resulting tuple.
- If it occurs in an iterable enumeration, the iterable object produced by evaluating the expression is chained directly to the resulting iterable object.

Likewise, a comprehension, as defined in §6.6.6 Comprehensions, produces multiple values, as specified by §8.6 Evaluation of comprehensions. The behavior depends upon where the comprehension occurs:

- If it occurs as an argument to a variadic parameter in a positional argument list, the values produced by the comprehension instance are evaluated immediately, packaged into an instance of `Sequence`, and passed to the variadic parameter, unless there are no values, in which case the object `empty` is passed to the variadic parameter.
- If it occurs as an argument to a parameter of type `Iterable` at the end of a named argument list, the comprehension instance is packaged into an iterable object that produces the values of the comprehension on demand, and this iterable object is passed directly to the parameter. The concrete class of this object is not specified here.
- If it occurs in a tuple enumeration, the values produced by the comprehension instance are evaluated immediately, packaged into an instance of `Sequence`, and appended to the resulting tuple.
- If it occurs in an iterable enumeration, the comprehension instance is packaged into an iterable object that produces the values of the comprehension on demand, and this iterable object is chained directly to the resulting iterable object. The concrete class of this object is not specified here.

8.5. Operator expressions

Most operator expression are defined in terms of function invocation, value evaluation, or a combination of invocations and evaluations, as specified in §6.8 Operators. The semantics of evaluation of an operator expression therefore follows from the above definitions of evaluation and invocation and from its definition in terms of evaluation and invocation.

However, this specification allows the compiler to take advantage of the optimized support for primitive value types provided by the virtual machine environment.

8.5.1. Operator expression optimization

As a special exception to the rules, the compiler is permitted to optimize certain operations upon certain types in the module `ceylon.language`. These types are: `Integer`, `Float`, `Character`, `Range`, `Entry`, `String`, `Array`, and `Tuple`.

Thus, the tables in the previous chapter define semantics only. The compiler may emit bytecode or compiled JavaScript that produces the same value at runtime as the pseudo-code that defines the operator, without actually executing any invocation, for the following operators:

- all arithmetic operators,
- the comparison and equality operators `==`, `!=`, `<=>`, `<`, `>`, `<=`, `>=` when the argument expression types are built-in numeric types, and
- the `Range` and `Entry` construction operators `..` and `->`.

In all operator expressions, the arguments of the operator must be evaluated from left to right when the expression is executed. In certain cases, depending upon the definition of the operator, evaluation of the leftmost argument expression results in a value that causes the final value of the operator expression to be produced immediately without evaluation of the remaining argument expressions. Optimizations performed by the Ceylon compiler must not alter these behaviours.

Note: this restriction exists to ensure that any effects are not changed by the optimizations.

8.5.2. Numeric operations

The arithmetic operations defined in §6.8.10 Arithmetic operators for values of type `Integer` and `Float` are defined in terms of methods of the interface `Numeric`. However, these methods themselves make use of the native operations of the underlying virtual machine. Likewise, values of type `Integer` and `Float` are actually represented in terms of a format native to the virtual machine.

It follows that the precise behavior of numeric operations depends upon the virtual machine upon which the program executes. However, certain behaviours are common to supported virtual machines:

- Values of type `Float` are represented according to the IEEE 754 specification, *IEEE Standard for Binary Floating-Point Arithmetic*, and floating point numeric operations conform to this specification. Where possible, a double-precision 64-bit representation is used. Note that even though `Float` has a 64-bit representation on both Java and JavaScript virtual machines, the actual range of representable values differs.
- Where possible, values of type `Integer` are represented in two's complement form using a fixed bit length. Where possible, a 64-bit representation is used. Overflow and underflow wrap silently. This is the case for the Java Virtual Machine.
- Otherwise, values of type `Integer` are represented according to the IEEE 754 specification. This is the case for JavaScript virtual machines.

Platform-dependent behavior of numeric operations is defined in the Java Language Specification, and the ECMAScript Language Specification.

It might be argued that having platform-dependent behavior for numeric operations opens up the same portability concerns that affected languages like C in the past. However, the cross-platform virtual machines supported by Ceylon already provide a layer of indirection that substantially eases portability concerns. Of course, numeric code is not guaranteed to be completely portable between the Java and JavaScript virtual machines, but it's difficult to imagine how such a level of portability could reasonably be achieved.

8.6. Evaluation of comprehensions

When a comprehension, as specified in §6.6.6 Comprehensions, is evaluated, a reference to the comprehension, together with a reference to the current frame or instance of every containing body, are packaged together into a *comprehension instance*. A comprehension instance is not considered a value in the sense of §8.1 Object instances, identity, and reference passing. Instead, it is a stream of values, each produced by evaluating the expression clause of the comprehension.

A comprehension consists of a series of clauses. Each clause of a comprehension, except for the expression clause that terminates the list of clauses, produces a stream of *frames*. A frame is a set of values for iteration variables and condition variables declared by the clause and its parent clauses.

Note: each child clause can be viewed as a body nested inside the parent clause. The lifecycle of comprehension frames reflects this model.

Evaluation of an expression occurring in a comprehension clause occurs in the context of the packaged frames associated with the comprehension instance together with a comprehension frame associated with the clause.

8.6.1. `for` clause

The expression which produces the source stream for a child `for` clause may refer to an iteration variable of a parent `for` clause. In this case the child clause is considered *correlated*. Otherwise it is considered *uncorrelated*.

In either case, the child clause produces a stream of frames. For each frame produced by the parent clause, and for each value produced by the source stream of the child clause, the child clause produces a frame consisting of the parent clause frame extended by the iteration variable value defined by the child clause.

This comprehension has a correlated `for` clause. For each character `c` in each string `w` in `words`, the child `for` clause produces the frame `{ String word=w; Character char=c; }`.

```
for (word in words) for (char in word) char
```

This comprehension has an uncorrelated `for` clause. For each string `n` in `nouns`, and each string `a` in `adjectives`, the child `for` clause produces the frame `{ String noun=n; String adj=a; }`.

```
for (noun in nouns) for (adj in adjectives) adj + " " + noun
```

8.6.2. `if` clause

A child `if` clause filters its parent clause frames. For every frame produced by the parent clause which satisfies the condition list of the child clause, the child clause produces that frame, extended by any condition variable defined by the child clause.

This comprehension has an `if` clause. For each object `o` in `objects` that is a nonempty `String`, the `if` clause produces the frame `{ Object obj=o; String str=o; }`.

```
for (obj in objects) if (is String str=obj, !str.empty) str
```

8.6.3. Expression clause

As specified in §6.6.6 Comprehensions, every comprehension ends in an expression clause. An expression clause produces a single value for each frame produced by its parent clause, by evaluating the expression in the frame. These resulting values are the values returned by the whole comprehension.

8.7. Concurrency

Neither this specification nor the module `ceylon.language` provide any facility to initiate or control concurrent execution of a program written in Ceylon. However, a Ceylon program executing on the Java Virtual Machine may interact with Java libraries (and other Ceyon modules) that make use of concurrency.

In this scenario, the execution of a Ceylon program is governed by the rules laid out by the Java programming language's execution model (Chapter 17 of the Java Language Specification). Ceylon references belonging to a class or interface are considered *fields* in the sense of the JLS. Any such refence not explicitly declared `variable` is considered a *final field*. Evaluation of a reference is considered a *use* operation, and assignment to or specification of a variable reference is considered an *assign* operation, again in terms of the JLS.

Chapter 9. Module system

The Ceylon module architecture enables a toolset which relieves developers of many mundane tasks. The module system specifies:

- the format of packaged deployable module archives (for the Java platform), module scripts (for the JavaScript platform), and source archives,
- the layout of a module repository
- the format of the package descriptor files which contain information about the packages contained in a module, including whether a package is visible to other modules, and
- the format of the module descriptor file which contains information about a module, along with a list of its versioned dependencies.

Thus, developers are never exposed to individual `.class` files, and are not required to manually manage module archives using the operating system file manager. Instead, the toolset helps automate the management of modules within module repositories.

Circular dependencies between modules are not supported. The Ceylon compiler detects such dependencies and produces an error.

9.1. The module runtime and module isolation

At any time, there may be multiple versions of a certain module available in the virtual machine. Modules execute under the control of the *module runtime*. The module runtime:

- obtains modules from module repositories,
- reads module metadata and recursively loads dependencies, and
- isolates modules that belong to different assemblies.

Execution of a module begins with a specified toplevel method or class, or with an entry point specified in the module descriptor, and imported modules are loaded lazily as classes they contain are needed. The name and version id of the imported module containing the needed class are determined from the imported package name specified by the compilation unit and the imported module version specified by the module descriptor.

The mechanism behind this is platform-dependent.

9.1.1. Module isolation for the Java platform

In the JVM environment, Each version of each module is loaded using a different class loader. Classes inside a module have access to other classes in the same module and to classes belonging to modules that are explicitly imported in the module descriptor. Classes in other modules are not accessible.

Ceylon supports a simplified class loader architecture:

- The *bootstrap* class loader owns classes required to bootstrap the module runtime. It is the direct parent of all module class loaders, and its classes are visible to all module class loaders.
- A *module* class loader owns classes belonging to a given version of a certain module. Its classes are visible only to classes belonging to the module class loader of a module which declares an explicit dependency on the given version of the first module.

The Ceylon module runtime for the JVM is implemented using JBoss Modules. It is included in the Ceylon SDK.

9.1.2. Module isolation for the JavaScript platform

In the JavaScript environment, modules are loaded using the `require()` function defined by CommonJS Modules.

There are various implementations of the CommonJS-style `require()` function, and Ceylon module scripts should work with any of them.

9.1.3. Assemblies

A future release of the language will add support for assemblies, that is, the ability to:

- *package together several interdependent versioned modules into a single archive for deployment as a single well-defined application or service,*
- *specify the name and version of the application or service, and*
- *override the versions of imported modules declared in `modules.ceylon`, as defined in §9.3.12 Module descriptors, with assembly-specific module versions.*

An assembly archive will probably just be an archived module repository with an assembly descriptor.

9.2. Source layout

A *source directory* contains Ceylon source code in files with the extension `.ceylon` and Java source code in files with the extension `.java`. The module and package to which a compilation unit belongs is determined by the subdirectory in which the source file is found.

The name of the package to which a compilation unit belongs is formed by replacing every path directory separator character with a period in the relative path from the root source directory to the subdirectory containing the source file. In the case of a Java source file, the subdirectory must agree with the package specified by the Java `package` declaration.

The name of the module to which a compilation unit belongs is determined by searching all containing directories for a module descriptor. The name of the module is formed by replacing every path directory separator character with a period in the relative path from the source directory to the subdirectory containing the module descriptor. If no module descriptor is found, the code belongs to the *default module*.

Note: the default module is intended only as a convenience for experimental code.

A package or compilation unit may belong to only one module. No more than one module descriptor may occur in the containing directories of a compilation unit.

Thus, the structure of the source directory containing the module `org.hello` might be the following:

```
source/
    org/
        hello/
            module.ceylon      //the module descriptor
            main/
                hello.ceylon
            default/
                DefaultHello.ceylon
            personalized/
                PersonalizedHello.ceylon
```

The source code for multiple modules may be contained in a single source directory.

9.3. Module architecture

Compiled code is automatically packaged into *module archives* and *module scripts* by the Ceylon compiler. A *module repository* is a repository containing module archives, module scripts, and other miscellaneous artifacts. A module archive or module script is automatically obtained from a module repository when code belonging to the module is needed by the compiler or module runtime.

Modules that form part of the Ceylon SDK are found in the module repository in the `modules` directory of the Ceylon distribution.

Red Hat maintains a central module repository at `http://modules.ceylon-lang.org`. Read access to this site is free of re-

gistration and free of charge. Ceylon projects may apply for a user account which provides write access to the central module repository.

A module belonging to the central module repository must satisfy the following regulations:

- the first element of the module name must be a top-level internet domain name, and the second element of the module name must be a second-level domain of the given top-level domain owned by the organization distributing the module, and.
- the module must be made available under a royalty-free license.

For example, a module developed by Red Hat might be named `org.jboss.server`.

TODO: should we require that module archives be signed using the Java `jarsigner` tool?

9.3.1. Module names and version identifiers

A module *name* is a period-separated list of lowercase identifiers, for example:

```
ceylon.language
```

```
org.hibernate
```

It is recommended that module names follow the Java package naming convention embedding the organization's domain name (in this case, `hibernate.org`). The namespace `ceylon` is reserved for Ceylon SDK modules. The namespace `java` is reserved for modules belonging to the Java SDK. The namespace `default` is reserved for the default module.

It is highly recommended, but not required, that every user-written module have at least three identifiers in its name. Therefore, `org.hibernate.orm` is strongly preferred to `org.hibernate`.

Modules may not be "nested". That is, the list of identifiers forming the name of a module may not be a prefix of the list of identifiers forming the name of another module.

A package belongs to a module if the list of identifiers forming the name of the module is a prefix of the list of identifiers forming the name of the package. For example, the packages:

```
ceylon.language
```

```
ceylon.language.assertion
```

```
ceylon.language.meta
```

```
ceylon.language.meta.declaration
```

belong to the module `ceylon.language`. The packages:

```
org.hibernate
```

```
org.hibernate.impl
```

```
org.hibernate.cache
```

belong to the module `org.hibernate`.

TODO: This might not work out all that well in practice, unless we introduce some additional convention for "extras" modules, for example, modules containing examples. It could be `org.hibernate` vs `org.hibernate_example` or `org.hibernate.core` vs `org.hibernate.example`.

The name of the default module is `default`. The default module has no version and cannot be published to a remote repository nor to the local repository cache under `~/.ceylon/repo`.

A module *version identifier* is a character string containing digits, periods, and lowercase letters, for example:

```
1.0.1
```

```
3.0.0.beta
```

TODO: at some stage we will probably need to add a format for specifying version ranges.

9.3.2. Module archive names for the Java platform

A *module archive name* is constructed from the module name and version identifier. A module archive name is of the following standard form:

```
<module>-<version>.car
```

where <*module*> is the full name of the module, and <*version*> is the module version identifier. For example:

```
ceylon.language-1.0.1.car
```

```
org.hibernate-3.0.0.beta.car
```

The default module has no version, its module archive name is `default.car`

9.3.3. Module script names for the JavaScript platform

A *module script name* is likewise constructed from the module name and version identifier. A module script name is of the following standard form:

```
<module>-<version>.js
```

where <*module*> is the full name of the module, and <*version*> is the module version identifier. For example:

```
ceylon.language-1.0.1.js
```

```
org.hibernate-3.0.0.beta.js
```

The default module has no version, its module archive name is `default.js`

9.3.4. Source archive names

A *source archive name* is of the following standard form:

```
<module>-<version>.src
```

For example:

```
ceylon.language-1.0.1.src
```

```
org.hibernate-3.0.0.beta.src
```

The default module has no version, its source archive name is `default.src`

9.3.5. Documentation archive names

A *documentation archive name* is of the following standard form:

```
<module>-<version>.doc.zip
```

For example:

```
ceylon.language-1.0.1.doc.zip
```

```
org.hibernate-3.0.0.beta.doc.zip
```

The default module has no version, its documentation archive name is `default.doc.zip`

9.3.6. Module archives

A Ceylon module archive is a Java `jar` archive which:

- contains a Ceylon module descriptor in the *module directory*,
- contains the compiled `.class` files for all compilation units belonging to the module, and
- has a filename which adheres to the standard for module archive names.

The *module directory* of the module archive is formed by replacing each period in the fully qualified package name with the directory separator character. For example, the module directory for the module `ceylon.language` is:

```
/ceylon/language
```

The module directory for the module `org.hibernate` is:

```
/org/hibernate
```

The *package directory* for a package belonging to the module archive is formed by replacing each period in the fully qualified package name with the directory separator character. For example, the package directory for the package `org.hibernate.impl` is:

```
/org/hibernate/impl
```

Inside a module archive, a `.class` file is found in the package directory of the package to which it belongs.

Thus, the structure of the module archive for the module `org.hello` might be the following:

```
org.hello-1.0.0.car
    META-INF/
        MANIFEST.MF
    org/
        hello/
            module.class       //the module descriptor
            main/
                package.class  //a package descriptor
                hello.class
            default/
                DefaultHello.class
            personalized/
                PersonalizedHello.class
```

A module archive may not contain multiple modules.

9.3.7. Module scripts

A Ceylon module script is a JavaScript source file which:

- complies with the CommonJS Modules specification, and
- has a filename which adheres to the standard for module script names.

9.3.8. Source archives

A *source archive* is a `zip` archive which:

- contains the source code (`.ceylon` and `.java` files) for all compilation units belonging to the module, and
- has a filename which adheres to the standard for source archive names.

Inside a source archive, a Ceylon or Java source file is located in the *package directory* of the package to which the compilation unit belongs. The package directory for a package belonging to the source archive is formed by replacing each period in the fully qualified package name with the directory separator character.

Thus, the structure of the source archive for the module `org.hello` might be the following:

```
org.hello-1.0.0.src
    org/
        hello/
            module.ceylon       //the module descriptor
            main/
                package.ceylon  //a package descriptor
                hello.ceylon
            default/
                DefaultHello.ceylon
            personalized/
                PersonalizedHello.ceylon
```

A source archive may not contain the source of multiple modules.

9.3.9. Documentation archives

A *documentation archive* is a `zip` archive which:

- contains the module documentation generated by the documentation compiler (`.html` and resources files), and
- has a filename which adheres to the standard for documentation archive names.

Inside a documentation archive, HTML source is located in the `module-doc` directory.

Thus, the structure of the documentation archive for the module `org.hello` might be the following:

```
org.hello-1.0.0.doc.zip
    module-doc/
        .resources/
            ...
        index.html
        search.html
        module.ceylon.html
        main/
            index.html
            package.ceylon.html
            hello.ceylon.html
        default/
            index.html
            DefaultHello.html
            DefaultHello.ceylon.html
        personalized/
            index.html
            PersonalizedHello.html
            PersonalizedHello.ceylon.html
```

A documentation archive may not contain the documentation of multiple modules.

9.3.10. Module repositories

A module repository is a directory structure on the local filesystem or a remote HTTP server.

- A *local* module repository is identified by a filesystem path.
- A *remote* module repository is identified by a URL with protocol `http:` or `https:`.

A *publishable* module repository is a local module repository, or a WebDAV-enabled remote module repository.

For example:

```
modules
```

```
/usr/bin/ceylon/modules
```

```
http://jboss.org/ceylon/modules
```

```
https://gavin:secret@modules.ceylon-lang.org
```

A module repository contains module archives, module scripts, source archives, and documentation archives. The address of an artifact belonging to the repository adheres to the following standard form:

```
<repository>/<module-path>/<version>/<artifact>
```

where *<repository>* is the filesystem path or URL of the repository, *<artifact>* is the name of the artifact, *<version>* is the module version, and *<module-path>* is formed by replacing every period with a slash in the module name.

The default module having no version, its access path does not contain the version.

```
<repository>/default/<archive>
```

For example, the module archive `ceylon.language-1.0.1.car`, module script, source archive `ceylon.language-1.0.1.src`, and documentation archive belonging to the repository included in the Ceylon SDK are obtained from the following addresses:

```
modules/ceylon/language/1.0.1/ceylon.language-1.0.1.car
```

```
modules/ceylon/language/1.0.1/ceylon.language-1.0.1.js
```

```
modules/ceylon/language/1.0.1/ceylon.language-1.0.1.src
```

```
modules/ceylon/language/1.0.1/ceylon.language-1.0.1.doc.zip
```

The module archive `org.hibernate-3.0.0.beta.car`, source archive `org.hibernate-3.0.0.beta.src`, and documentation archive belonging to the repository `http://jboss.org/ceylon/modules` are obtained from the following addresses:

```
http://jboss.org/ceylon/modules/org/hibernate/3.0.0.beta/org.hibernate-3.0.0.beta.car
```

```
http://jboss.org/ceylon/modules/org/hibernate/3.0.0.beta/org.hibernate-3.0.0.beta.src
```

```
http://jboss.org/ceylon/modules/org/hibernate/3.0.0.beta/org.hibernate-3.0.0.beta.doc.zip
```

The module archive `org.h2-1.2.141.car` and legacy archive `org.h2-1.2.141.jar` belonging to the repository `/usr/bin/ceylon/modules` are obtained from the following addresses:

```
/usr/bin/ceylon/modules/org/h2/1.2.141/org.h2-1.2.141.car
```

```
/usr/bin/ceylon/modules/org/h2/1.2.141/org.h2-1.2.141.jar
```

For each archive, the module repository may contain a SHA-1 checksum file. The checksum file is a plain text file containing just the SHA-1 checksum of the archive. The address of a checksum file adheres to the following standard form:

```
<repository>/<module-path>/<version>/<archive>.sha1
```

The compiler or module runtime verifies the checksum after downloading the archive from the module repository.

A module repository may contain documentation generated by the Ceylon documentation compiler in exploded form. A module's documentation resides in the *module documentation directory*, a directory with address adhering to the following standard form:

```
<repository>/<module-path>/<version>/module-doc/
```

For example, the home page for the documentation of the module `org.hibernate` is:

```
http://jboss.org/ceylon/modules/org/hibernate/module-doc/index.html
```

9.3.11. Package descriptors

A *package descriptor* is defined in a source file named `package.ceylon` in the package it describes.

```
PackageDescriptor: Annotations "package" FullPackageName ";"
```

A `package` may be annotated `shared`. A `shared` package is visible outside the containing module, that is, in any module which imports the containing module.

The package descriptor is optional for unshared packages.

```
"The typesafe query API."
license "http://www.gnu.org/licenses/lgpl.html"
shared package org.hibernate.query;
```

9.3.12. Module descriptors

A *module descriptor* is defined in a source file named `module.ceylon` in the root package of the module it describes (the package with the same name as the module).

```
ModuleDescriptor: Annotations "module" FullPackageName StringLiteral ModuleBody
```

The literal string after the module name specifies the version of the module.

A module may import other modules.

```
ModuleBody: "{" ModuleImport* "}"
```

```
ModuleImport: Annotations "import" (FullPackageName|StringLiteral) StringLiteral ";"
```

The name of the imported module may be specified using the usual syntax for a module name, or as a literal string, to allow interoperation with legacy module repositories existing outside the Ceylon ecosystem.

Note: this enables interoperation with Maven.

Note: in Ceylon 1.0 it is illegal to explicitly import the module `ceylon.language`. The language module is always implicitly imported.

The string literal after the imported module name specifies the version of the imported module.

An imported module may be annotated `optional` and/or `shared`.

- If module `x` has a `shared` import of module `y`, then any module that imports `x` implicitly imports `y`.
- If module `x` has an `optional` import of module `y`, then `x` may be executed even if `y` is not available at runtime.

If a declaration belonging to module `x` is visible outside the module and involves types imported from a different module `y`, then the module import of `y` in the module descriptor for `x` must be `shared`.

```
"The best-ever ORM solution!"
license "http://www.gnu.org/licenses/lgpl.html"
module org.hibernate "3.0.0.beta" {
    shared import ceylon.language "1.0.1";
    import javax.sql "4.0";
}
```

```
"The test suite for Hibernate"
license "http://www.gnu.org/licenses/lgpl.html"
module org.hibernate.test "3.0.0.beta" {
    import org.hibernate "3.0.0.beta";
    TestSuite().run();
}
```

TODO: do we allow procedural code in the body of a `module`?