Project Ceylon

A Better Java

Version: For internal discussion only

Table of Contents

A work in progress

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.

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, including: primitive types, arrays, constructors, getters/setters, checked exceptions, raw types, thread synchronization.

Ceylon has the following goals:

- to execute on the JVM, and interoperate with Java code,
- 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 least verbose/most cryptic language around,
- 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 improve compile-time typesafety using special handling for null values,
- to provide language-level modularity, and
- to improve on Java's very primitive facilities for meta-programming, thus making it easier to write frameworks.

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

Therefore, the Ceylon SDK will provide:

- the OpenJDK virtual machine,
- a compiler that compiles both Ceylon and Java source,
- Eclipse-based tooling,
- a module runtime, 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

Ceylon is a statically-typed, general-purpose, object-oriented language featuring a syntax derived from the class of languages that includes Java and C#. Ceylon programs execute in any standard Java Virtual Machine and, like Java, take advantage of the memory management and concurrency features of that environment. The Ceylon compiler is able to compile Ceylon code that calls Java classes or interfaces, and Java code that calls Ceylon classes or interfaces. Ceylon differs from Java by eliminating primitive types and arrays and introducing a number of improvements (inspired in some cases by dynamic languages such as SmallTalk, Python and Ruby) to the language and type system that reduce verbosity compared to Java or C#. Moreover, Ceylon provides its own native SDK as a replacement for the Java platform class libraries.

Ceylon features the same inheritance and generic type models as Java. There are no primitive types or arrays in Ceylon, so all values are instances of the type hierarchy root `lang.Object`. However, the Ceylon compiler is permitted to optimize certain code to take advantage of the better performance of primitive types on the JVM.

By default, Ceylon attributes and locals do not accept null values. Nullable locals and attributes must be explicitly declared using the `optional` annotation. Nullable expressions are not assignable to non-`optional` locals or attributes, except via use of the `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.

By default, Ceylon classes, attributes and locals are immutable. Mutable classes, attributes and locals must be explicitly declared using the `mutable` annotation. An immutable class may not declare mutable attributes or extend a mutable class. An immutable attribute must be assigned when the class is instantiated. An immutable local must not be assigned more than once. In addition, Ceylon supports closures, called *functors*. These language features encourage a more functional style of programming, resulting in code which is more typesafe, easier to reason about, and easier to refactor.

Ceylon classes do not contain fields, in the traditional sense. Instead, Ceylon supports only a higher-level construct: *attributes*, which are similar to C# properties.

Ceylon methods are similar to Java methods. However, Ceylon does not feature any Java-like constructor declaration and so each Ceylon class has exactly one "constructor". Instead, Ceylon provides a sophisticated object initialization syntax.

Ceylon control flow structures are enhanced versions of the traditional constructs found in C, C# and Java. Even better, Ceylon functors can be used together with a special method invocation protocol to achieve more specialized flow control and other more functional-style constructs such as comprehensions.

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 implements that type.

True open classes are not supported. However, Ceylon supports *extensions*, which allows addition of methods and interfaces to existing types, and transparent conversion between types, within a textual scope. Decorators only affect the operations provided by a type, not its state. This facility makes it easy for Ceylon code to transparently interoperate and intercompile with Java code.

Ceylon features an exceptions model inspired by Java and C#, but checked exceptions are not supported.

Ceylon introduces a set of syntax extensions that support the definition of domain-specific languages and expression of structured data. These extensions include specialized syntax for initializing objects and collections and expressing literal values. One application of this syntax is the support for Java/C#-like code annotations.

Ceylon provides sophisticated support for meta-programming, including a typesafe metamodel and events. This facility is inspired by similar features found in dynamic languages such as Smalltalk, Python and Ruby, and by the much more complex features found in aspect oriented languages like Aspect J. Ceylon does not, however, support aspect oriented programming as a language feature.

Ceylon features language-level package and module constructs, and language-level access control with four levels of visibility for program elements: private (the default), `package`, `module` and `public`. There is no equivalent of Java's `protected`.

1.1. A simple example

Here's a classic example, implemented in Ceylon:

```
doc "The classic Hello World program"
```

```
class Hello {

    main void hello() {
        log.info("Hello, World!");
    }

}
```

This code defines a Ceylon class named `Hello`, with a single method with no parameters and no return value, named `hello`. An *annotation* appears on the method declaration. The `main` annotation specifies that this method is called automatically when the virtual machine is started. The `hello()` method calls the `info()` method of an attribute named `log` defined by the `lang.Object` class. By default, this method displays its parameter on the console.

This improved version of the program takes a name as input from the console:

```
doc "A more personalized greeting"
class Hello {

    main void hello(Process process) {
        String name = process.args.firstOrNull ? "World";
        log.info("Hello, ${name}!");
    }

}
```

This time, the `hello()` method has a parameter. This parameter value is *injected* by Ceylon's built-in dependency management engine. The `Process` object has an attribute named `args`, which holds a `List` of the program's command line arguments. The local `args` is initialized from these arguments. The `?` operator returns the first argument that is not null. Finally, the value of the local is interpolated into the message string.

Finally, lets rewrite this program as a web page:

```
import html.Html;
import html.Head;
import html.Body;
import html.Div;
import web.Page;

doc "A web page that displays a greeting"
page "/hello.html"
class Hello(Request request)
        extends Html(request) {

    String name
        = request.parameters["name"].firstOrNull ? "World";

    head = Head { title="Hello World"; };

    body = Body {
        Div {
            cssClass = "greeting";
            "Hello, ${name}!"
        },
        Div {
            cssClass = "footer";
            "Powered by Ceylon."
        }
    };

}
```

This program demonstrates Ceylon's support for defining structured data, in this case, HTML. The `Hello` class extends Ceylon's `Html` class and initializes its `head` and `body` attributes. The `page` annotation specifies the URL at which this HTML should be accessible.

Chapter 2. Declarations

All classes, interfaces, methods, attributes and locals must be declared.

2.1. General declaration syntax

All declarations follow a general pattern.

2.1.1. Abstract declaration

Declarations conform to the following general schema:

```
Annotation*
keyword? Type? (TypeName|MemberName) TypeParams? FormalParams?
Supertype?
Interfaces?
TypeConstraints?
Declaration?
```

The Ceylon compiler enforces identifier naming conventions. Types must be named with an initial uppercase. Members, parameters and locals must be named with an initial lowercase or underscore.

```
PackageName := LIdentifier
```

```
TypeName := UIdentifier
```

```
MemberName := LIdentifier
```

```
ParameterName := LIdentifier
```

Toplevel declarations may declare imports:

```
Import* Declaration+
```

```
Import := "import" ImportElement ("." ImportElement)* ('.' '*' | "alias" ImportElement)? ";"
```

```
ImportElement := PackageName | TypeName | MemberName
```

2.1.2. Annotation list

Declarations may be preceded by a list of annotations.

```
Annotation := MemberName ( Arguments | Literal )?
```

Unlike Java, the name of an annotation may not be a qualified name.

An annotation is a static method call. For an annotation with no constructor parameters, or with a single literal-valued constructor parameter, the parentheses/braces may be omitted.

For example:

```
doc "The user login action"
author "Gavin King"
see #LogoutAction
scope(session)
action { description="Log In"; url="/login"; }
public deprecated
```

2.1.3. Formal parameter list

Method and class declarations may declare formal parameters, including defaulted parameters and a varargs parameter.

```
FormalParams :=
"("
FormalParam ("," FormalParam)* ("," DefaultParam)* ("," VarargsParam)? |
DefaultParam ("," DefaultParam)* ("," VarargsParam)? |
VarargsParam?
")"
```

```
FormalParam := Param | EntryParamPair | RangeParamPair
```

Each parameter is declared using with a type and name and may have annotations.

```
Param := Annotation* Type ParameterName
```

Defaulted parameters specify a default value.

```
DefaultParam := FormalParam Specifier
```

The = specifier is used throughout the language to indicate a value which cannot be reassigned.

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

A varargs parameter accepts a lists of arguments or a single argument of type `Iterable`.

```
VarargsParam := Annotation* Type "..." ParameterName
```

Parameters of type `Entry` or `Range` may be specified as a pair of variables.

```
EntryParamPair := Annotation* Type ParameterName "->" Type ParameterName
```

```
RangeParamPair := Annotation* Type ParameterName ".." ParameterName
```

For example:

```
(Product product, Integer quantity=1)
```

```
(Name name, optional Organization org=null, Address... addresses)
```

```
(Key key -> Value value)
```

A variable pair declaration of form `U u -> V v` results in a single parameter of type `Entry<U,V>`.

```
(Float value, Integer min..max)
```

A variable pair declaration of form `T x .. y` results in a single parameter of type `Range<T>`.

2.1.4. Generic type parameters

Method, class and interface declarations may declare generic type parameters.

```
TypeParams := "<" TypeParam ("," TypeParam)* ">"
```

```
TypeParam := Variance TypeName
```

A covariant type parameter is indicated using `out`. A contravariant type parameter is indicated using `in`.

```
Variance :=  ("out" | "in")?
```

For example:

```
Map<K, V>
```

```
Sender<in M>
```

```
Container<out T>
```

```
BinaryFunction<in X, in Y, out R>
```

2.1.5. Type declaration

Method and attribute declarations must declare a type.

```
Type := RegularType | FunctorType
```

Most types are classes or interfaces:

```
RegularType := QualifiedTypeName TypeArguments?
```

Unlike Java, the name of a type may not be qualified by the package name.

```
QualifiedTypeName := (TypeName ".")* TypeName
```

A generic type must specify arguments for the generic type parameters.

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

For example:

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

2.1.6. Functor type declaration

There are also functor types:

```
FunctorType := FunctorHeader FormalParams
```

```
FunctorHeader := "functor" Annotation* (Type | "void")
```

The parameter names in a functor type declaration do not affect assignability of the functor type. They are used when the functor is invoked using named parameters.

For example:

```
functor Comparison(X x, X y)
```

```
functor Boolean(Payment payment)
```

```
functor void(Y element)
```

```
functor optional Object()
```

```
functor Y(optional Y object, Factory<Y> factory)
```

2.1.7. Extended class

Classes may extend other classes using the `extends` clause.

```
Supertype := "extends" Instantiation
```

For example:

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

2.1.8. Implemented interfaces

Classes and interfaces may satisfy (implement or extend) interfaces, using the `satisfies` clause.

```
Interfaces = "satisfies" Type ("," Type)*
```

For example:

```
satisfies Sequence<T>, Collection<T>
```

2.1.9. Generic type constraint list

Method, class and interface declarations which declare generic type parameters may declare constraints upon the type parameters using the `where` clause.

```
TypeConstraints = "where" TypeConstraint (AMPERSAND TypeConstraint)*
```

```
TypeConstraint := TypeName ( (">="|"<=") Type | FormalParams )
```

There are three kinds of type constraints:

- upper bounds,
- lower bounds, and
- initialization parameter specifications.

For example:

```
where X >= Number<X> & Y >= Comparable<Y> & Y(Integer count)
```

Should we use `satisfies` instead of >=? For example `where X satisfies Number<X> & Y satisfies Comparable<Y>`.

2.2. Classes

A class is a stateful, instantiable type. Class are declared according to the following:

```
Annotation*
"class" TypeName TypeParams? FormalParams?
Supertype?
Interfaces?
TypeConstraints?
"{" Instances? Statement* "}"
```

2.2.1. Class inheritance

A class may extend another class, and implement any number of interfaces. For example:

```
public mutable entity
class Customer(Name name, optional Organization org = null)
        extends Person(name, org) {
    ...
}
```

```
class Token()
        extends Datetime()
        satisfies Comparable<Token>, Identifier {
    ...
}
```

The types listed after the `satisfies` keyword are the implemented interfaces. The type specified after the `extends` keyword is a superclass. The semantics of class inheritance are exactly the same as Java, and the above declarations are equivalent to the following Java declarations:

```
@entity public class Customer
        extends Person {
    public Customer(Name name) { this(name, null); }
    public Customer(Name name, Organization org) { super(name, org); }
    ...
}
```

```
class Token
        extends Datetime
        implements Comparable<Token>, Identifier {
    public Token() { super(); }
    ...
}
```

2.2.2. Class instantiation

Ceylon classes do not support a Java-like constructor declaration syntax. However, Ceylon supports *class initialization parameters*. A class initialization parameter may be used anywhere in the class body.

All non-`optional` attributes of the class must be explicitly initialized somewhere in the body of the class definition, unless the class is declared `abstract`, in which case they must be initialized within the body of every subclass definition.

This declaration:

```
public class Key(Lock lock) {
    public Lock lock = lock;
}
```

Is equivalent to this Java class:

```
public class Key {
    private final ReadAttribute<Lock> lock;
    public ReadAttribute<Lock> lock() { return lock; }

    public Key(Lock lock) {
        this.lock = new SimpleReadAttribute<Lock>(lock);
    }

}
```

This declaration:

```
public class Key(Lock lock) {
    public Lock lock { return lock };
}
```

Is equivalent to this Java class:

```
public class Key {
    private Lock _lock;

    private final ReadAttribute<Lock> lock = new ReadAttribute<Lock>() {
        @Override public Lock get() { return _lock; }
    };
    public ReadAttribute<Lock> lock() { return lock; }

    public Key(Lock lock) {
        _lock = lock;
    }

}
```

Class initialization parameters are optional. The following class:

```
public mutable class Point {
    public mutable Exact x := 0.0;
    public mutable Exact y := 0.0;
}
```

Is equivalent to this Java class with a default constructor:

```
public class Point {
    private final Attribute<Exact> x = new SimpleAttribute<Exact>( new Exact(0.0) );
    public SimpleAttribute<Exact> x() { return x; }

    private final Attribute<Exact> y = new SimpleAttribute<Exact>( new Exact(0.0) );
    public SimpleAttribute<Exact> y() { return y; }
}
```

TODO: is this the right thing to say? Alternatively, we could say that classes without a parameter list can't have statements or initialized simple attributes in the body of the class, and get the constructor automatically generated for the list of attributes.

A subclass must pass values to each superclass initialization parameter.

```
public class SpecialKey1()
        extends Key( new SpecialLock() ) {
    ...
}
```

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

Which are equivalent to the Java:

```
public class SpecialKey1
        extends Key {
    public SpecialKey1() {
        super( SpecialLock() );
    }
    ...
}
```

```
public class SpecialKey2
        extends Key {
    public SpecialKey2(Lock lock) {
        super(lock);
    }
    ...
}
```

The body of a class may contain arbitrary code, which is executed when the class is instantiated.

```
public mutable class DiagonalPoint(Exact position)
        extends Point() {

    Exact displacement = position**2/2;
    Integer sign = ifTrue (position.negative) then -1 else 1;

    x := sqrt(displacement) * sign;
    y := sqrt(displacement) * sign;

    assert "must have distance ${position} from origin"
        that x**2 + y**2 == position**2;

}
```

The compiler is permitted to optimize private attribute declarations. So the above class is equivalent to:

```
public class DiagonalPoint extends Point {

    public DiagonalPoint(final Exact position) {
        final Exact displacement = position.exponentiate(2).divide(2);
        final Integer sign = position.negative ? -1 : 1;

        x = sqrt(displacement).multiply(sign);
        y = sqrt(displacement).multiply(sign);

        assert_(new F0<String>() {
                   public String call() {
                       return "must have distance " + position + " from origin";
                   }
                },
                new F0<Boolean>() {
                   public Boolean call() {
                       return x.exponentiate(2) + y.exponentiate(2)
```

```
                                == position.exponentiate(2);
                     }
                  });
    }

}
```

TODO: should class initialization parameters be allowed to be declared `mutable`?

TODO: should class initialization parameters be allowed to be declared `public/package/module`, allowing a shortcut simple attribute declaration like in Scala?

2.2.3. Defaulted parameters

When a class with an defaulted parameter is instantiated, and a value is not assigned to the defaulted parameter by the caller, the default value defined by the specifier is used.

This class:

```
public class Counter(Integer initialCount=0) { ... }
```

Is equivalent to a class with three Java constructor declarations and an inner class:

```
public class Counter {

    public Counter() {
        Counter(0);
    }

    public Counter(Integer initialCount) {
        ...;
    }

    public Counter(CounterParameters namedParameters) {
        Counter( namedParameters.initialCount );
    }

    public static class CounterParameters {
        private Integer initialCount=0;
        CounterParameters initialCount(Integer initialCount) {
            this.initialCount = initialCount;
            return this;
        }
    }

}
```

This named parameter call:

```
Counter { initialCount=10; }
```

Is equivalent to this Java code:

```
new Counter ( new CounterParameters().initialCount(10) );
```

2.2.4. Annotations

Every annotation is a static (non-void) method call. This Ceylon class:

```
doc "Represents a person"
author "Gavin"
public class Person { ... }
```

Is equivalent to this Java code:

```
public class Person { ...

    static {
        Type<Person> type = Type.get(Person.class);
        type.addAnnotation( doc("Represents a person") );
        type.addAnnotation( author("Gavin") );
        type.addAnnotation( public() );
    }

    ...

}
```

2.2.5. Instance enumeration

A class may specify an enumerated list of instances:

```
Instances := "instances" Instance ("," Instance)* ("..." | ";")
```

```
Instance := MemberName Arguments?
```

The keyword `instances` is used to define a set of predefined instances.

```
public class DayOfWeek { instances mon, tues, wed, thurs, fri, sat, sun; }
```

```
public class DayOfWeek(String name) {
    instances
        mon("Monday"),
        tues("Tuesday"),
        wed("Wednesday"),
        thurs("Thursday"),
        fri("Friday"),
        sat("Saturday"),
        sun("Sunday");

    public String name = name;

}
```

A class with an `instances` declaration implicitly extends `lang.Selector`, a subclass of `java.lang.Enum`. The above declarations are equivalent to the following Java declarations:

```
public class DayOfWeek
        extends Selector<DayOfWeek> {

    public DayOfWeek mon = new DayOfWeek("mon", 0);
    public DayOfWeek tues = new DayOfWeek("tues", 1);
    public DayOfWeek wed = new DayOfWeek("wed, 2");
    public DayOfWeek thurs = new DayOfWeek("thurs", 3);
    public DayOfWeek fri = new DayOfWeek("fri", 4);
    public DayOfWeek sat = new DayOfWeek("sat", 5);
    public DayOfWeek sun = new DayOfWeek("sun", 6);

    private DayOfWeek(String id, int ord) {
        super(id, ord);
    }

}
```

```
public class DayOfWeek
        extends Selector<DayOfWeek> {

    private final ReadAttribute<String> name;

    public DayOfWeek mon = new DayOfWeek("Monday", "mon", 0);
    public DayOfWeek tues = new DayOfWeek("Tuesday", "tues", 1);
    public DayOfWeek wed = new DayOfWeek("Wednesday", "wed, 2");
    public DayOfWeek thurs = new DayOfWeek("Thursday", "thurs", 3);
    public DayOfWeek fri = new DayOfWeek("Friday", "fri", 4);
    public DayOfWeek sat = new DayOfWeek("Saturday", "sat", 5);
    public DayOfWeek sun = new DayOfWeek("Sunday", "sun", 6);

    private DayOfWeek(String name, String id, int ord)
    {
        super(id, ord);
        name = new SimpleReadAttribute(name);
    }

}
```

2.3. Interfaces

An interface is a type which does not specify implementation. Interfaces may not be directly instantiated. Interfaces are declared according to the following:

```
Annotation*
"interface" TypeName TypeParams?
Interfaces?
TypeConstraints?
"{" ( MethodStub | AttributeStub )* "}"
```

For example:

```
public interface Comparable<T> {
    Comparison compare(T other);
}
```

Which is equivalent to the following Java interface:

```
public interface Comparable<T> {
    Comparison compare(T other);
}
```

2.3.1. Interface inheritance

An interface may extend any number of other interfaces. For example:

```
public interface List<T>
        satisfies Sequence<T>, Collection<T> {
    ...
}
```

The types listed after the `satisfies` keyword are the supertypes. All supertypes of an interface must be interfaces. The semantics of interface inheritance are exactly the same as Java, and the above declaration is equivalent to the following Java declaration:

```
public interface List<T>
        extends Sequence<T>, Collection<T> {
    ...
}
```

2.4. Methods

A method is a callable block of code. Methods may have parameters and may return a value. Methods are declared according to the following:

```
Method :=
Annotation*
(Type | "void") MemberName TypeParams? FormalParams
TypeConstraints?
( ";" | Block )
```

For example:

```
public Integer add(Integer x, Integer y) {
    return x + y;
}
```

```
Identifier createToken() {
    return Token();
}
```

```
public optional U get(optional V key);
```

```
public void print(Object... objects) {
    for (Object object in objects) { log.info($object); }
}
```

```
public void addEntry(V key -> U value) { ... }
```

TODO: should we allow a method or attribute getter/setter to omit the braces if it consists of exactly one statement, just like we do for functor expressions? I have actually tried this and it parses.

If there is no method body, the method must be declared `abstract`.

The Ceylon compiler preserves the names of method parameters, using a Java annotation.

```
@FormalParameterNames({"x", "y"})
public Integer add(Integer x, Integer y) { ... }
```

A Ceylon method invocation is equivalent to a Java method invocation. The semantics of method declarations are identical to Java, except that Ceylon methods may declare optional parameters.

2.4.1. Defaulted parameters

Methods with defaulted parameters may not be overloaded.

When a method with a defaulted parameter is called, and a value is not assigned to the defaulted parameter by the caller, the default value defined by the specifier is used.

This method:

```
public class Counter {

    package void init(Integer initialCount=0) {
        count:=initialCount;
    }

    ...

}
```

Is equivalent to three Java method declarations and an inner class:

```
public class Counter {

    void init() {
        init(0);
    }

    void init(Integer initialCount) {
        count=initialCount;
    }

    void init(CounterInitParameters namedParameters) {
        init( namedParameters.initialCount );
    }

    static class CounterInitParameters {
        private Integer initialCount=0;
        CounterInitParameters initialCount(Integer initialCount) {
            this.initialCount = initialCount;
            return this;
        }
    }

}
```

This named parameter call:

```
counter.init { initialCount=10; }
```

Is equivalent to this Java code:

```
counter.init ( new CounterInitParameters().initialCount(10) );
```

2.4.2. Interface methods and abstract methods

Methods declared by interfaces and methods marked `abstract` may not specify a body:

```
MethodStub :=
Annotation*
(Type | "void") MemberName TypeParams? FormalParams
TypeConstraints?
";"
```

Interface methods and abstract methods must be implemented by every non-`abstract` class that implements the interface or subclasses the abstract class.

Classes which declare methods marked `abstract` must also be declared `abstract`, and may not be instantiated.

2.5. Attributes

There are three kinds of declarations related to attribute definition:

- Simple attribute declarations define state (very similar to a Java field).
- Attribute getter declarations define how the value of a derived attribute is obtained.
- Attribute setter declarations define how the value of a derived attribute is assigned.

Unlike Java fields, Ceylon attribute access is polymorphic and attribute definitions may be overridden by subclasses. If the attribute is not declared `optional`, it may not be overridden or implemented by an attribute declared `optional`.

For example:

```
package mutable String firstName;
```

```
mutable Integer count := 0;
```

```
public static Exact pi = calculatePi();
```

```
public String name { return join(firstName, lastName); }
public assign name { firstName = first(name); lastName = last(name); }
```

```
public Float total {
    Float sum = 0.0;
    for (LineItem li in lineItems) {
        sum += li.amount;
    }
    return sum;
}
```

An attribute declaration is equivalent to a Java method declaration together with a Java field declaration, both of type `lang.Attribute` or `lang.ReadAttribute`, both with the same name as the attribute.

TODO: Should we generate getters and setters, just for interop with Java?

The compiler is permitted to optimize private attributes to a simple Java field declaration or a local variable in a Java constructor. Private attributes may not be accessed via reflection.

2.5.1. Attributes with getter/setter code

An attribute getter is declared as follows:

```
AttributeGetter := Annotation* Type MemberName Block
```

When getter code is specified, the Java field is initialized to an instance of an anonymous inner subclass of `lang.Attribute` or `lang.ReadAttribute` that overrides the `get()` method with the content of the getter block. For example:

```
public Float total { return items.totalPrice; }
```

is equivalent to this Java code:

```
private final ReadAttribute<Float> total = new ReadAttribute<Float>() {
    @Override public Float get() { return items.get().totalPrice; }
};
public ReadAttribute<Float> total() { return total; }
```

An attribute setter is declared as follows:

```
AttributeSetter := Annotation* "assign" MemberName Block
```

When setter code is specified, the Java field is initialized to an instance of an anonymous inner subclass of `lang.Attribute` that overrides the `set()` method with the content of the setter block. For example:

```
public String name { return join(firstName, lastName); }
public assign name { firstName = first(name); lastName = last(name); }
```

is equivalent to this Java code:

```
private final Attribute<String> name = new Attribute<String>() {
    @Override public String get() { return join(firstName, lastName); }
    @Override public void set(String name) { firstName = first(name); lastName = last(name); }
};
public Attribute<String> name() { return name; }
```

The attribute name specified by the setter must correspond to a matching attribute getter.

TODO: should we allow overloaded attribute setters, for example:

```
assign Name name { firstName = name.firstName; lastName = name.lastName; }
```

2.5.2. Simple attributes and locals

Simple attribute defines state. Simple attributes are declared according to the following:

```
SimpleAttribute := Annotation* Type MemberName Initializer? ";"
```

The value of an immutable attribute is specified using =. Mutable attributes may be initialized using the assignment operator :=.

```
Initializer := ("="|":=") Expression
```

TODO: I would like to support `def` in place of the type for a simple attribute or local with an initializer. For example:

```
def names = List<String>();
```

Formal parameters of classes and methods are also considered to be simple attributes.

A local is really just a special case of a simple attribute declaration, but one that is optimized by the compiler.

- An attribute declared inside the body of a class represents a local if it is not used inside a method, attribute setter or attribute getter declaration.
- An attribute declared inside the body of a method represents a local.
- A formal parameter of a class represents a local if it is not used inside a method, attribute setter or attribute getter declaration.
- A formal parameter of a method represents a local.

The semantics of locals are identical to Java local variables.

For a simple attribute that is not a local, the Java field is initialized to an instance of `lang.SimpleAttribute` or `lang.SimpleReadAttribute`. For example:

```
package mutable String firstName;
```

is equivalent to this Java code:

```
private final Attribute<String> firstName = new SimpleAttribute<String>();
Attribute<String> firstName() { return firstName; }
```

While:

```
mutable Integer count := 0;
```

is equivalent to this Java code:

```
private final Attribute<Integer> count = new SimpleAttribute<Integer>(0);
private Attribute<Integer> count() { return count; }
```

And:

```
public Integer max = 99;
```

is equivalent to this Java code:

```
private final ReadAttribute<Integer> max = new SimpleReadAttribute<Integer>(99);
public ReadAttribute<Integer> max() { return max; }
```

2.5.3. Interface attributes and abstract attributes

Attributes declared by interfaces and attributes marked `abstract` may not specify an initalizer, getter or setter:

```
AttributeStub := Annotation* Type MemberName ";"
```

Interface attributes and abstract attributes must be implemented by every non-`abstract` class that implements the interface or subclasses the abstract class.

Interface attributes and abstract attributes may be specified `mutable`, in which case every subtype must also define the attribute to be mutable.

Classes which declare methods marked `abstract` must also be declared `abstract`, and may not be instantiated.

2.6. Type aliases

A type alias allows a type to be referred to more compactly.

```
Annotation* "alias" TypeName TypeParams? Interfaces? TypeConstraints? ";"
```

A type alias may satisfy either:

- any number of interfaces and at most one class, or
- a single functor type.

Any expression which is assignable to all the satisfied types is assignable to the alias type.

For example:

```
public alias People satisfies List<Person>;
```

```
package alias ComparableCollection<X> satisfies Collection<X>, Comparable<X>;
```

```
alias Compare<T> satisfies functor Comparison(T x, T y);
```

A shortcut is provided for definition of private aliases.

```
import java.util.List alias JavaList;
```

2.7. Extensions

An extension allows values of one type to be transparently converted to values of another type. Extensions are declared by annotating a method, attribute or class `extension`. An extension method must be a static method with exactly one parameter, or a non-static method with no parameters. An extension class must have exactly one initialization parameter. An extension method or attribute must have a non-`optional` type.

For example:

```
public class Person {
    ...
    public extension User user;
}
```

```
public static extension User personToUser(Person person) {
    return person.user;
}
```

```
public extension CollectionUtils<T>(Collection<T> collection) {

    public Collection<T> nonZeroElements() {
        return collection.elements()
            having (T element) element!=0;
    }

    ...
}
```

An extension method is called a converter. An extension class is called a decorator.

The Ceylon compiler searches for an appropriate extension whenever a value of one type is assigned to a non-assignable type. If exactly one extension for the two types is found, the compiler inserts a call to the extension. For example, this Ceylon assignment:

```
import org.mydomain.myproject.Converters.personToUser;
...
Person person = ...;
User user = person;
```

Is equivalent to the following Java code:

```
Person person = ...;
User user = personToUser(person);
```

The Ceylon compiler also searches for an appropriate extension whenever a member is invoked that is not declared by the type. If exactly one extension that declares the member is found for the type, the compiler inserts a call to the extension. For example, this Ceylon method call:

```
import org.mydomain.myframework.CollectionUtils;
...
Collection<Integer> ints = ...;
Collection<Integer> result = ints.nullElements();
```

Is equivalent to this Java code:

```
Collection<Integer> ints = ...;
Collection<Integer> result = new CollectionUtils(collection).nullElements();
```

An extension is only available in a source file that expicitly `imports` the extension.

2.8. Declaration modifiers

The following annotations are compiler instructions:

- `public`, `module`, `package` determine the visibility of a declaration (by default, the declaration is visible only inside the same compilation unit).
- `abstract` specifies that a class cannot be instantiated, or that a method or attribute of an abstract class must be implemented by all subclasses.
- `static` specifies that a method can be called without an instance of the type that defines the method.
- `mutable` specifies that an attribute or local may be assigned, or that a class has assignable attributes.
- `optional` specifies that a value may be null.
- `final` indicates that a class may not be extended, or that a method or attribute may not be overridden.
- `override` indicates that a method or attribute overrides a method or attribute defined by a supertype.
- `extension` specifies that a method is a converter, or that a class is a decorator.
- `once` indicates that a method is executed at most once, and the resulting value is cached.
- `deprecated` indicates that a method, attribute or type is deprecated.
- `volatile` indicates a volatile simple attribute.

TODO: We can minimize backtracking in the parser by making all these "annotations" be keywords. It lets the parser recognize a member declaration a little bit more easily. But on the other hand it's a new special kind of thing.

TODO: should there be an `annotation` *modifier for static methods which can be used as annotations?*

The following annotations are instructions to the documentation compiler:

- `doc` specifies the documentation for a program element.
- `author` specifies the author of a program element.
- `see` specifies a related member or type.
- `throws` specifies a thrown exception type.

The string value of the `doc` and `author` annotations is parsed by the documentation compiler as Seam Text, a simple ANTLR-based wiki text format.

The following annotations are important to the Ceylon SDK.

- `id` specifies that an attribute should be tested by the `equals()` method, and included in the `hash`.
- `transient` specifies that an attribute is not included in the serialized form of the object.
- `read` and `write` indicate methods or attributes that are protected from multithreaded access using a reentrant read/write lock with deadlock detection.

Chapter 3. Blocks and control structures

Method, attribute, class and functor literal bodies contain procedural code that is executed when the method, functor or attribute is invoked, or when the class is instantiated. The code contains expressions and control directives and is organized using blocks and control structures.

3.1. Blocks and statements

A *block* is list of semicolon-delimited statements, method, attribute and local declarations, and control structures, surrounded by braces. Some blocks end in a control directive. The statements, local specifiers and control structures are executed sequentially.

A *statement* is an assignment, an invocation of a method or functor, an instantiation of a class, a control structure, a control directive, or a method, attribute or local declaration.

```
Block := "{" Statement* ControlStatement? "}"
```

```
Statement := ExpressionStatement | DirectiveStatement | Specification | Declaration
```

Only certain expressions are valid statements: assignment, prefix or postfix increment or decrement, invocation of a method or functor and instantiation of a class.

```
ExpressionStatement := ( Assignment | IncrementOrDecrement | Invocation ) ";"
```

```
Invocation := MethodInvocation | StaticMethodInvocation | FunctorInvocation | Instantiation
```

TODO: it would be possible to say that any expression is a valid statement, but this seems to just open up more potential programming errors. So I think it's better to limit statements to assignments, invocations and instantiations.

Control directive statements end execution of the current block and force the flow of execution to resume in some outer scope. They may only appear at the end of a block, so the semicolon terminator is optional.

```
DirectiveStatement := Directive ";"?
```

The Ceylon language distinguishes between assignment to a mutable value (the `:=` operator) and specification of the value of an immutable local or attribute (using `=`). A specification is not an expression.

```
Specification := MemberName Specifier ";"
```

Blocks may contain declarations, which are, by default, only visible inside the block:

```
Declaration := Method | SimpleAttribute | AttributeGetter | AttributeSetter | Local
```

TODO: should we allow statements to be annotated, for example:

```
doc "unsafe assignment" suppressWarnings(typesafety): apple = orange;
```

3.2. Control directives

A control directive is a statement that affects the flow of execution.

Ceylon provides the following control directives:

- the `return` directive—to return a value from a method or attribute getter,
- the `produce` directive—to return a value from a functor,
- the `found` directive—to terminate a `for/fail` loop successfully,
- the `break` directive—to terminate a loop unsuccessfully, and

- the `throw` directive—to raise an exception.

```
Directive := "return" Expression? | "produce" Expression | "throw" Expression | "found" | "break"
```

The `return` directive may not be used outside the body of an attribute getter or a non-`void` method.

The `produce` directive may not be used outside the body of a non-`void` functor expression.

The `break` directive may not be used outside the body of a loop.

The `found` directive may not be used outside the body of a `for/fail` loop.

TODO: instead of `break` and `found`, we could support `break success` and `break failure`.

TODO: We could support `for/fail` loops that return a value by adding a `found expr` directive. We could support conditionals that return a value by adding a `then expr` directive.

3.3. Control structures

Control of execution flow may be achieved using control directives and control structures. Control structures include conditionals, loops, and exception management. For many tasks, the use of these traditional control structures is considered bad style, and the use of expressions with functor parameters is preferred. However, control structures support the use of the `return`, `break` and `found` directives, whereas functors do not. Therefore, some tasks may be accomplished only with control structures.

Control structures are not considered to be expressions in Ceylon.

Ceylon provides the following control structures:

- the `if/else` conditional—for controlling execution based upon a boolean value, and dealing with null values,
- the `switch/case/else` conditional—for controlling execution using an enumerated list of values,
- the `do/while` loop—for loops which terminate based upon the value of a boolean expression,
- the `for/fail` loop—for looping over elements of a collection, and
- the `try/catch/finally` exception manager—for managing exceptions and controlling the lifecycle of objects which require explicit destruction.

```
ControlStructure := IfElse | SwitchCaseElse | DoWhile | ForFail | TryCatchFinally
```

Some control structures allow embedded declaration of a local that is available inside the control structure body.

```
Variable := Type MemberName
```

Some control structures expect conditions:

```
Condition := Expression | ExistsCondition | IsCondition
```

```
ExistsCondition := ("exists" | "nonempty") (Variable Specifier | Expression)
```

```
IsCondition := "is" (Variable Specifier | Type Expression)
```

The semantics of a condition depend upon whether the `exists`, `nonempty` or `is` modifier appears:

- If no `is`, `exists` or `nonempty` modifier appears, the condition must be an expression of type `Boolean`. The condition is satisfied if the expression evaluates to `true` at runtime.
- If the `is` modifier appears, the condition must be an expression or local specifier of type `Object`. The condition is satisfied if the expression or specifier evaluates to an instance of the specified type at runtime.

- If the `exists` modifier appears, the condition must be an expression or local specifier of type `optional Object`. The condition is satisfied if the expression or specifier evaluates to a non-null value at runtime.
- If the `nonempty` modifier appears, the condition must be an expression or local specifier of type `optional Container`. The condition is satisfied if the expression or specifier evaluates to a non-null value at runtime, and if the resulting `Container` is non-empty.

For `exists` or `nonempty` conditions:

- if the condition is a local variable specifier, the local variable may be declared without the `optional` annotation, even though the specifier expression is of type `optional`, or
- if the condition is a local variable, the local will be treated by the compiler as having non-null type inside the block that follows immediately, relaxing the usual compile-time restrictions upon `optional` types.

For `is` conditions:

- if the condition is a local variable specifier, the local variable may be declared with the specified type, even though the specifier expression is not of that type, or
- if the condition is a local variable, the local will be treated by the compiler as having the specified type inside the block that follows immediately.

TODO: Should we support, like Java, single-statement control structure bodies without the braces.

3.3.1. `if/else`

The `if/else` conditional has the following form:

```
IfElse :=
"if" "(" Condition ")" Block
("else" "if" "(" Condition ")" Block)*
("else" Block)?
```

If the condition is satisfied, the first block is executed. Otherwise, the second block is executed, if it is defined.

For example:

```
if (payment.amount <= account.balance) {
    account.balance -= payment.amount;
    payment.paid := true;
}
else {
    throw NotEnoughMoneyException();
}
```

```
public void welcome(optional User user) {
    if (exists user) {
        log.info("Hi ${user.name}!");
    }
    else {
        log.info("Hello World!");
    }
}
```

```
public Payment payment(Order order) {
    if (exists Payment p = order.payment) { return p }
    else { return Payment(order) }
}
```

```
if (exists Payment p = order.payment) {
    if (p.paid) { log.info("already paid"); }
}
```

```
if (is CardPayment p = order.payment) {
    return p.card;
}
```

3.3.2. `switch/case/else`

The `switch/case/else` conditional has the following form:

```
SwitchCaseElse := "switch" "(" Expression ")" (Cases | "{" Cases "}")
```

```
Cases :=
("case" "(" Case ")" Block)+
("else" Block)?
```

The switch expression may be of any type. The case values must be expressions of type `Case<X>`, where `X` is the switch expression type, or an explicit `null`.

```
Case := Expression ("," Expression)* | "is" Type | "null"
```

If a `case (null)` is defined, the switch expression type must be `optional`.

If the switch expression type is `optional`, there must be an explicit `case (null)` defined.

If no `else` block is defined, the switch expression must be of type `Selector`, and all enumerated instances of the class must be explicitly listed.

If the switch expression is of type `Selector`, and all enumerated instances of the class are explicitly listed, no `else` block may be specified.

When the construct is executed, the switch expression value is tested against the case values (using `Case.test()`) , and the case block for the first case value that tests true is executed. If no case value tests true, and an `else` block is defined, the `else` block is executed.

For an `is` type case, if the switch expression is a local, then the local will be treated by the compiler as having the specified type inside the `case` block.

TODO: support `catch`-style syntax instead of `case (is ...)`?

For example:

```
public PaymentProcessor processor {
    switch (payment.type)
    case (null) { throw NoPaymentTypeException() }
    case (credit, debit) { return cardPaymentProcessor }
    case (check) { return checkPaymentProcessor }
    else { return interactivePaymentProcessor }
}
```

```
switch (payment.type) {
    case (credit, debit) {
        log.debug("card payment");
        cardPaymentProcessor.process(payment, user.card);
    }
    case (check) {
        log.debug("check payment");
        checkPaymentProcessor.process(payment);
    }
    else {
        log.debug("other payment type");
    }
}
```

```
switch (payment) {
    case (is CardPayment) {
        pay(payment.amount, payment.card);
    }
    case (is CheckPayment) {
        pay(payment.amount, payment.check);
    }
    else {
        log.debug("other payment type");
    }
}
```

3.3.3. `for/fail`

The `for/fail` loop has the following form:

```
ForFail :=
"for" "(" ForIterator ")" Block
("fail" Block)?
```

An iteration variable declaration must specify an iterated expression that contains the range of values to be iterated.

```
ForIterator := Variable ("->" Variable)? "in" Expression
```

Each iterated expression must be of type `Iterable` or `Iterator`. If two iteration variables are defined, it must be of type `Iterable<Entry>` or `Iterator<Entry>`.

The body of the loop is executed once for each iterated element.

If the loop exits early via execution of one of the control directives `found`, `return` or `throw`, the fail block is not executed. Otherwise, if the loop completes, or if the loop exits early via execution of the control directive `break`, the fail block is executed, if it is defined.

For example:

```
for (Person p in people) { log.info(p.name); }
```

```
for (String key -> Integer value in map) {
    log.info("${key} = ${value}");
}
```

```
for (Person p in people) {
    log.debug("found ${p.name}");
    if (p.age >= 18) {
        log.info("found an adult: ${p.name}");
        found
    }
}
fail {
    log.info("no adults");
}
```

3.3.4. `do/while`

The `do/while` loop has the form:

```
DoWhile :=
( "do" ("(" DoIterator ")")? Block? )?
"while" "(" Condition ")" (";" | Block)
```

The loop may declare an iterator variable, which may be mutable.

```
DoIterator := Annotation* Variable Initializer
```

Both blocks are executed repeatedly, until the termination condition first evaluates to false, at which point iteration ends. In each iteration, the first block is executed before the condition is evaluated, and the second block is executed after it is evaluated.

TODO: does `do/while` need a fail block? Python has it, but what is the real usecase?

For example:

```
do (mutable Integer i:=0) {
    log.info("count = " + $i);
}
while (i<=10) { i++; }
```

```
do (Iterator<Person> iter = org.employees.iterator)
while (iter.more) {
    log.info( iter.next().name );
}
```

```
do (Iterator<Person> iter = people.iterator)
while (iter.more) {
    Person p = iter.next();
    log.debug(p.name);
    p.greet();
}
```

```
mutable Person person := ....;
while (exists Person parent = person.parent) {
    log.info(parent.name);
    person := parent;
}
```

```
mutable Person person := ....;
do {
    log.info(person.name);
    person := person.parent;
}
while (!person.dead);
```

```
do (mutable Person person := ...) {
    log.info(person.name);
}
while (!person.parent.dead) {
    person := person.parent;
}
```

TODO: `do/while` is significantly enhanced compared to other Java-like languages. Is this truly a good thing?

3.3.5. `try/catch/finally`

The `try/catch/finally` exception manager has the form:

```
TryCatchFinally :=
"try" ( "(" Resource ("," Resource)* ")" )? Block
("catch" "(" Variable ")" Block)*
("finally" Block)?
```

When an exception occurs in the try block, the first matching catch block is executed, if any. The finally block is always executed.

The type of each catch local must extend `lang.Exception`.

Each resource expression must be of type `Usable`.

```
Resource := Variable Specifier | Expression
```

When the construct is executed, `begin()` is called upon the resource, the try block is executed, and then `end()` is called upon the resource, with the thrown exception, if any.

For example:

```
try ( File file = File(name) ) {
    file.open(readOnly);
    ...
}
catch (FileNotFoundException fnfe) {
    log.info("file not found: ${name}");
}
catch (FileReadException fre) {
    log.info("could not read from file: ${name}");
}
finally {
    if (file.open) file.close();
}
```

```
try (semaphore) { map[key] := value; }
```

(This example shows the Ceylon version of Java's `synchronized` keyword.)

```
try ( Transaction(), Session s = Session() ) {
    Person p = s.get(#Person, id)
    ...
    return p
}
catch (NotFoundException e) {
    return null
}
```

Chapter 4. Expressions

Ceylon expressions are significantly more powerful than Java, allowing a more declarative style of programming.

Expressions are formed from:

- literal values,
- invocations of methods, attributes and functors, and instantiations of classes and enumerations,
- operators, and
- functor expressions.

4.1. Literals

Ceylon supports a special literal value syntax for each of the following types: `Date`, `Time Integer`, `Float`, `Character`, `String` and `Regex`.

```
Literal :=
DateLiteral | TimeLiteral
IntegerLiteral | FloatLiteral |
CharacterLiteral | StringLiteral | RegexLiteral |
TypeLiteral | MemberLiteral
"this" | "super" | "null" | "none"
```

The keywords `this`, `super` and `null` are equivalent to the Java forms. The keyword `none` represents an empty enumeration.

4.1.1. Datetime literals

A datetime literal has the form:

```
DateLiteral :=
"'"
Digit{1,2} "/" Digit{1,2} "/" Digit{4}
"'"
```

```
TimeLiteral :=
"'"
Digit{1,2} ":" Digit{2} ( ":" Digit{2} ( ":" Digit{3} )? )?
(" " "AM"|"PM")?
(" " Character{3,4})?
"'"
```

For example:

```
Date date = '25/03/2005';
```

```
Time time = '12:00 AM PST';
```

Datetimes may be composed from dates and times using the `@` operator.

```
Datetime datetime = '25/03/2005' @ '12:00 AM PST';
```

TODO: should we allow wildcards and ranges, to get cron-style expressions like `'1/*/*' @ '00:00 GMT'`?

The `..` operator lets us construct intervals:

```
Interval<Date> timeRange = '0:0:00' .. '12:59:59 PM';
```

```
Interval<Date> dateRange = '1/1/2008' .. '31/12/2007';
```

```
Interval<Datetime> datetimeRange = '1/1/2008' @ '0:0:00' .. '31/12/2007' @ '12:59:59 PM';
```

4.1.2. Integer literals

An integer literal has this form:

```
IntegerLiteral =
Digit+ |
"'" ( HexDigit{4} | HexDigit{8} ) "'"
```

For example:

```
Integer i = i + 10;
```

```
panel.backgroundColor = 'FF33';
```

4.1.3. Float literals

A float literal has this form:

```
FloatLiteral :=
Digit+ "." Digit+
( ("E"|"e")? ("+"|"-")? Digit+ )?
```

For example:

```
public static Float pi = 3.14159;
```

Equivalent to this Java code:

```
public static final Float pi = new lang.Float(3.14159f);
```

4.1.4. String literals

A string literal has this form:

```
StringLiteral = """ ( Character+ | "${" Expression "}" )* """
```

For example:

```
person.name = "Gavin";
```

```
log.info("${Time()} ${message}");
```

```
String multilineString = "Strings may
span multiple lines
if you prefer.";
```

The first example is equivalent to this Java code:

```
person.name().set( new lang.String("Gavin") );
```

4.1.5. Character literals

A character literal has this form:

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

For example:

```
if ( string[i] == '+' ) { ... }
```

Equivalent to this Java code:

```
if ( string.at(i).equals( new lang.Character('+') ) ) { ... }
```

TODO: do we really need character literals??

4.1.6. Regex literals

A regex literal has this form:

```
RegexLiteral := "`" RegularExpression "`"
```

For example:

```
Boolean isEmail = email.matches( `^\w+@((\w+)\.)+$` );
```

```
Integer quotedWords = `\W"w+"\W`.matcher(text).count();
```

The second example is equivalent to this Java code:

```
Integer quotedWords = new lang.Regex("\\W\"w+\"\\W").matcher(text).count();
```

4.1.7. Type and member literals

The `Type` object for a type, the `Method` object for a method, or the `Attribute` object for an attribute may be referred to using a special literal syntax:

```
TypeLiteral := HASH Type
```

```
MemberLiteral := (Type)? HASH MemberName
```

For example:

```
Type<List<String>> stringListType = #List<String>>;
```

```
Attribute<Person, String> nameAttribute = Person#name;
```

```
Method<Person, (String) void>> sayMethod = Person#say;
```

Note that the `#` curry operator may also be applied to an arbitrary expression, to obtain a `Callable<functor R(P p, Q q,...)>` where `P p, Q q,...` are the method parameter types and `R` is the method return type, or a `Value<T>` or `OpenValue<T>` where `T` is the attribute type.

4.2. Objects and collections

There are no true literals for user-defined classes. Rather, there is a nice syntax for calling the class constructor. For example:

```
Person gavin = Person {
    firstName = "Gavin";
    initial = 'A';
    lastName = "King";
    address = Address { ... };
    birthdate = Date { day = 25; month = MARCH; year = ... }
    employer = jboss;
};
```

Likewise, there are no true literals for lists, sets or maps. However, the `..` and `->` operators, together with the convenient enumeration constructor syntax, and some built-in extensions help us achieve the desired effect.

```
List<Integer> numbers = 1..10;
```

```
List<String> languages = { "Java", "Ceylon", "Smalltalk" };
```

Enumerations are transparently converted to sets or maps, allowing sets and maps to be initialized as follows:

```
Map<String, String> map = { "Java"->"Boring...", "Scala"->"Difficult :-(", "Ceylon"->"Fun!" };
```

```
Set<String> set = { "Java", "Ceylon", "Scala" };
```

```
OpenList<String> list = {};
```

4.3. Invocations

Methods, classes, class instances and functors are *invokable*. Invocation of a class is called *instantiation*. Invocation of a class instance is called *attribute configuration*.

Any invocation must specify values for parameters, either by listing or naming parameter values.

```
Arguments := PositionalArguments | NamedArguments
```

Required parameters must be specified. Defaulted parameters and varargs may also be specified.

When parameter values are listed, required parameters are assigned first, in the order in which they were declared, followed by defaulted parameters, in the order they were declared. If there are any remaining defaulted parameters, they will be assigned their default values. On the other hand, if any parameter values are unassigned, they will be treated as varargs.

```
PositionalArguments := "(" ( Expression ("," Expression)* )? ")"
```

When parameter values are named, required and defaulted parameter values are specified by name. Vararg parameter values are specified by listing them.

```
NamedArguments := "{" NamedArgument* VarargArguments? "}"
```

Named arguments are followed by semicolons.

```
NamedArgument := ParameterName Specifier ";"
```

Vararg arguments are seperated by commas.

```
VarargArguments := VarargArgument ("," VarargArgument)*
```

```
VarargArgument := Expression | Variable Specifier
```

A vararg parameter may be a local declaration. Multiple vararg parameters may be constructed using a `for` comprehension.

TODO: exactly what types are accepted by a vararg parameter? Any `Iterable`? Exactly how is it accessed in the body of the method? As a `List`?

TODO: should there be a special syntax to "spread" the values of a list into vararg parameters, eg. `*list`.

4.3.1. Method invocation

Method invocations follow the following schema.

```
MethodInvocation := MemberReference TypeArguments? Arguments
```

```
MemberReference := (Expression DOT)? MemberName
```

For example:

```
log.info("Hello world!")
```

```
log.info { message = "Hello world!"; }
```

```
printer.print { join = ", "; "Gavin", "Emmanuel", "Max", "Steve" }
```

```
printer.print { "Names: ", from (Person p in people) p.name }
```

The value of a method invocation is the return value of the method. The parameter values are passed to the formal parameters of the method.

Methods may not be invoked on an expression of type `optional`.

4.3.2. Static method invocation

Static method invocations follow the following schema.

```
StaticMethodInvocation := (RegularType DOT)? MemberName TypeArguments? Arguments
```

For example:

```
HashCode.calculate(default, firstName, initial, lastName)
```

```
HashCode.calculate { algorithm=default; firstName, initial, lastName }
```

The value of a static method invocation is the return value of the static method. The parameter values are passed to the formal parameters of the method.

4.3.3. Class instantiation

Classes may be instantiated according to the following schema:

```
Instantiation := RegularType TypeArguments? Arguments
```

For example:

```
Map<String, Person>(entries)
```

```
Point { x=1.1; y=-2.3; }
```

```
ArrayList<String> { capacity=10; "gavin", "max", "emmanuel", "steve", "christian" }
```

```
Panel {
    label = "Hello";
    Input i = Input(),
    Button {
        label = "Hello";
        action = () {
            log.info(i.value);
        }
    }
}
```

The value of a class instantiation is a new instance of the class. The parameter values are passed to the initialization parameters of the class. If the class has no initialization parameters, they are assigned directly to attributes of the class (in this case, named parameters must be used).

4.3.4. Functor invocation

Functor invocations follow the following schema.

```
FunctorInvocation := Expression Arguments
```

For example:

```
compare("AAA", "aaa")
```

```
compare { x = "AAA"; y = "aaa"; }
```

The value of a functor invocation is the return value of the functor. The parameter values are passed to the formal parameters of the functor implementation.

4.3.5. Attribute access

Attribute get access follows this schema:

```
AttributeGet := MemberReference
```

This attribute getter call:

```
String name = person.name;
```

is equivalent to the following Java code:

```
String name = person.name().get();
```

Attribute set access follows the following schema:

```
AttributeSet := MemberReference AssignmentOperator Expression
```

This attribute setter call:

```
person.name := "Gavin";
```

is equivalent to the following Java code:

```
person.name().set("Gavin");
```

If getter code is specified, and `assign` is not specified, the attribute is not settable, and any attempt to assign to the attribute will result in a compiler error.

4.4. Inline classes

Inline classes may be instantiated according to:

```
InlineClass := "new" Annotation* Instantiation Interfaces "{" Statement* "}"
```

For example:

```
Task task = new Task() {
    timeout := 1000;
    override void run() { ... }
    override void fail(Exception e) { ... }
}
```

```
return new transactional Database()
        satisfies Resource {
    url := "jdbc:hsqldb:.";
    username := "gavin";
    password = "foobar";
    override void create() { open(); }
    override void destroy() { close(); }
};
```

The value of an inline class instantiation is a new instance of the inline class. The block may contain method and attribute declarations and arbitrary code.

4.5. Enumeration instantiation

Enumerations may be instantiated according to the following simplified syntax:

```
EnumerationInstantiation := "{" ( Expression ("," Expression)* )? "}"
```

In this case, there is no need to explicitly specify the type.

For example:

```
Enumeration<String> names = { "gavin", "max", "emmanuel", "steve", "christian" };
```

```
OpenList<Connection> connections = {};
```

Empty braces `{}` and `none` are synonyms for a special value that can be assigned to any enumeration type.

Note: `none` can appear as a functor body, but `{ ... }` cannot (unless surrounded by parentheses).

4.6. Assignable expressions

Certain expressions are *assignable*. An assignable expression may appear as the LHS of the := (assign) operator, and possibly, depending upon the type of the expression, as the LHS of the numeric or logical assignment operators `=, +=, -=, *=, /=, %=, &=, |=, ^=, &&=, ||=` or as the subject of the increment or decrement operators `++, --`.

The following expressions are assignable:

- a local declared `mutable`, for example `count`,
- any attribute expression where the underlying attribute has a setter or is a simple attribute declared `mutable`, for example `person.name`, and
- element expressions for the type `OpenCorrespondence`, for example `order.lineItems[0]`

When an assignment expression is executed, the value of the local or attribute is set to the new value, or the `define()` method of `OpenCorrespondence` is called.

Thus, the following statement:

```
order.lineItems[0] := LineItem { product = prod; quantity = 1; };
```

Is equivalent to the Java:

```
order.lineItems.define( 0, new LineItem(prod, 1) );
```

4.7. Operators

Operators are syntactic shorthand for more complex expressions involving method invocation or attribute access. Each operator is defined for a particular type. There is support for user-defined operator *overloading*. However, the semantics of an operator may be customized by the implementation of the type that the operator applies to.

For example, the following Ceylon code examples:

```
Double z = x * y;
```

```
++count;
```

```
Integer j = i++;
```

```
x *= 2;
```

```
if ( x > 100 ) { ... }
```

```
User gavin = users["Gavin"];
```

```
List<Item> firstPage = list[0..20];
```

```
for ( Integer i in 1..10 ) { ... }
```

```
if ( name == value ) return ... ;
```

```
log.info( "Hello " + $person + "!")
```

```
List<String> names = ArrayList<Person>()^.add(person1)^.add(person2)*.name;
```

```
optional String name = person?.name;
```

Are equivalent to the following (Ceylon) code:

```
Double z = x.multiply(y);
```

```
count = count.successor;
```

```
Integer j = ( i = i.successor ).predecessor;
```

```
z = z.multiply(2);
```

```
if ( x.compare(100).larger ) { ... }
```

```
User gavin = users.value("Gavin");
```

```
List<Item> firstPage = list.range(0..20);
```

```
for ( Integer i in Range(1,10) ) { ... }
```

```
if ( nullsafeEquals(name, value) ) return ... ;
```

```
log.info( "Hello ".join(person.string).join("!") )
```

```
List<String> names =
    Spread<String> {
        lhs = Chain<String> {
            lhs = Chain<String> {
                lhs = ArrayList();
                override void call() { lhs.add(person1); } }.lhs
            }
            override void call() { lhs.add(person2); } }.lhs
        }
        override void call(String element) { element.name; }
    }.result;
```

```
optional String name = if (exists person) person.name else null;
```

4.7.1. List of operators

The following table defines the semantics of the Ceylon operators:

Table 4.1.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Assignment						
`:=`	`lhs := rhs`	assign		`mutable optional Object`	`optional Object`	`optional Object`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Member reference						
`#`	`lhs#member`	curry		`Object`		`Callable<functor R(P p, Q q,...)>`, `Value<T>` or `OpenValue<T>`
Member invocation						
`.`	`lhs.member`	invoke		`Object`		Member type
`^.`	`lhs^.member`	chain invoke		`X`		`X`
`?.`	`lhs?.member`	nullsafe invoke	`if (exists lhs) lhs.member else null`	`optional Object`		`optional` member type
`*.`	`lhs*.member`	spread invoke	`for (X x in lhs) x.member`	`Iterable<X>`		`List` of member type
Equality						
`===`	`lhs === rhs`	identical	`Object.identical(lhs, rhs)`	`optional Object`	`optional Object`	`Boolean`
`==`	`lhs == rhs`	equal	`if (exists lhs) lhs.equals(rhs) else if (exists rhs) false else true`	`optional Object`	`optional Object`	`Boolean`
`!=`	`lhs != rhs`	not equal	`if (exists lhs) lhs.equals(rhs).negation else if (exists rhs) true else false`	`optional Object`	`optional 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).smallAs`	`Comparable<T>`	`T`	`Boolean`
`>=`	`lhs >= rhs`	large as	`lhs.compare(rhs).largeAs`	`Comparable<T>`	`T`	`Boolean`
Logical operations						
`!`	`!rhs`	logical negation	`rhs.negation`		`Boolean`	`Boolean`
`|`	`lhs | rhs`	disjunction	`lhs.or(rhs)`	`Boolean`	`Boolean`	`Boolean`
`&`	`lhs & rhs`	conjunction	`lhs.and(rhs)`	`Boolean`	`Boolean`	`Boolean`
`^`	`lhs ^ rhs`	exclusive dis-	`lhs.xor(rhs)`	`Boolean`	`Boolean`	`Boolean`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
		junction				
`||`	`lhs || rhs`	shortcircuit disjunction	`if (lhs) true else rhs`	`Boolean`	`Boolean`	`Boolean`
`&&`	`lhs && rhs`	shortcircuit conjunction	`if (lhs) rhs else false`	`Boolean`	`Boolean`	`Boolean`
`=>`	`lhs => rhs`	implication	`if (lhs) rhs else true`	`Boolean`	`Boolean`	`Boolean`
Logical assignment						
`|=`	`lhs |= rhs`	or	`lhs = lhs.or(rhs)`	`Boolean`	`Boolean`	`Boolean`
`&=`	`lhs &= rhs`	and	`lhs = lhs.and(rhs)`	`Boolean`	`Boolean`	`Boolean`
`^=`	`lhs ^= rhs`	xor	`lhs = lhs.xor(rhs)`	`Boolean`	`Boolean`	`Boolean`
`||=`	`lhs ||= rhs`	shortcircuit or	`lhs = if (lhs) true else rhs`	`Boolean`	`Boolean`	`Boolean`
`&&=`	`lhs &&= rhs`	shortcircuit and	`lhs = if (lhs) rhs else false`	`Boolean`	`Boolean`	`Boolean`
Existence (null value handling)						
`ex-ists`	`lhs exists`	exists	`if (exists lhs) true else false`	`optional Ob-ject`		`Boolean`
`nonem pty`	`lhs nonempty`	nonempty	`if (exists lhs) lhs.empty.negation else false`	`optional Container`		`Boolean`
`?`	`lhs ? rhs`	default	`if (exists lhs) lhs else rhs`	`optional T`	`T`	`T`
Default assignment						
`?=`	`lhs ?= rhs`	default as-signment	`if (exists lhs) lhs else lhs=rhs`	`optional T`	`T`	`T`
Containment						
`in`	`lhs in rhs`	in	`lhs.in(rhs)`	`X`	`Cat-egory<X>` or `Iter-able<X>`	`Boolean`
Assignability						
`is`	`lhs is Rhs`	is	`lhs.instanceOf(#Rhs)`	`Object`		`Boolean`
Concatenation						
`+`	`lhs + rhs`	join	`lhs.join(rhs)`	`List<X>`	`List<X>`	`List<X>`
Keyed element access						
`[]`	`lhs[index]`	lookup	`lhs.value(index)`	`Correspond-ence<X,Y>`	`X`	`Y`
`[]`	`lhs[indices]`	lookup	`lhs.values(index)`	`Correspond-ence<X,Y>`	`List<X>`	`List<Y>`
`[]`	`lhs[indices]`	lookup	`lhs.values(index)`	`Correspond-ence<X,Y>`	`Set<X>`	`Set<Y>`
`[,,]`	`lhs[x, y, z]`	enumerated	`Enumeration( lhs.lookup(x),`	`List<X>`	`Integer`	`Enumera-`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
		range	`lhs.lookup(y), lhs.lookup(z) )`			`tion<X>`
Subranges						
`[..]`	`lhs[x..y]`	subrange	`lhs.range(x,y)`	`List<X>`	`Integer`	`List<X>`
`[...]`	`lhs[x...]`	upper range	`lhs.tail(x)`	`List<X>`	`Integer`	`List<X>`
`[...]`	`lhs[...y]`	lower range	`lhs.head(y)`	`List<X>`	`Integer`	`List<X>`
Constructors						
`..`	`lhs .. rhs`	range	`Range(lhs, rhs)`	`Ordinal<T>`	`T`	`Range<T>`
`->`	`lhs -> rhs`	entry	`Entry(lhs, rhs)`	`U`	`V`	`Entry<U,V>`
`@`	`lhs @ rhs`	datetime	`Datetime(lhs, rhs)`	`Date`	`Time`	`Datetime`
`..`	`lhs .. rhs`	interval	`Interval(lhs, rhs)`	`Instant`	`Instant`	`Interval`
Render						
`$`	`$rhs`	render	`if (exists rhs) rhs.string else ""`		`optional Object`	`String`
Increment, decrement						
`++`	`++rhs`	increment	`rhs = rhs.successor`		`Ordinal<T>`	`T`
`--`	`--rhs`	decrement	`rhs = rhs.predecessor`		`Ordinal<T>`	`T`
`++`	`lhs++`	successor	`(lhs = lhs.successor).predecessor`	`Ordinal<T>`		`T`
`--`	`lhs--`	predecessor	`(lhs = lhs.predecessor).successor`	`Ordinal<T>`		`T`
Numeric operations						
`-`	`-rhs`	negation	`rhs.inverse`		`Number`	`Number`
`+`	`lhs + rhs`	sum	`lhs.add(rhs)`	`Number`	`Number`	`Number`
`-`	`lhs - rhs`	difference	`lhs.subtract(rhs)`	`Number`	`Number`	`Number`
`*`	`lhs * rhs`	product	`lhs.multiply(rhs)`	`Number`	`Number`	`Number`
`/`	`lhs / rhs`	quotient	`lhs.divide(rhs)`	`Number`	`Number`	`Number`
`%`	`lhs % rhs`	remainder	`lhs.remainder(rhs)`	`Number`	`Number`	`Number`
`**`	`lhs ** rhs`	exponentiate	`lhs.exponentiate(rhs)`	`Number`	`Number`	`Number`
Numeric assignment						
`+=`	`lhs += rhs`	add	`lhs = lhs.add(rhs)`	`Number`	`Number`	`Number`
`-=`	`lhs -= rhs`	subtract	`lhs = lhs.subtract(rhs)`	`Number`	`Number`	`Number`
`*=`	`lhs *= rhs`	multiply	`lhs = lhs.multiply(rhs)`	`Number`	`Number`	`Number`
`/=`	`lhs /= rhs`	divide	`lhs = lhs.divide(rhs)`	`Number`	`Number`	`Number`
`%=`	`lhs %= rhs`	remainder	`lhs = lhs.remainder(rhs)`	`Number`	`Number`	`Number`

4.7.2. Operator precedence and associativity

This table defines operator precedence from highest to lowest, along with associativity rules:

Table 4.2.

Operations	Operators	Type	Associativity
Member invocation and lookup, subrange, reflection:	`.`, `^.`, `*.`, `?.`, `[]`, `[..]`, `[...]`, `[,,]`, `#`	Binary / ternary / N-ary	Left
Prefix increment, decrement, negation, render:	`++`, `--`, `-`, `$`,	Unary prefix	Right
Postfix increment, decrement:	`++`, `--`	Unary postfix	Left
Exponentiation:	`**`	Binary	Right
Multiplication, division, remainder for numbers:	`*`, `/`, `%`	Binary	Left
Addition, subtraction, concatenation for numbers and lists:	`+`, `-`	Binary	Left
Date/time composition:	`@`	Binary	None
Range, interval and entry construction:	`..`, `->`	Binary	None
Existence, emptiness:	`exists`, `nonempty`	Unary postfix	None
Default:	`?`	Binary	Right
Comparison, containment, assignability:	`<=>`, `<`, `>`, `<=`, `>=`, `in`, `is`	Binary	None
Equality:	`==`, `!=`, `===`	Binary	None
Negation:	`!`	Unary prefix	Right
Conjunction:	`&&`, `&`	Binary	Left
Disjunction:	`||`, `|`, `^`	Binary	Left
Implication:	`=>`	Binary	None
Assignment, numeric assignment, logical assignment, default assignment:	`:=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `&&=`, `||=`, `?=`	Binary	Right

TODO: should `?` have a higher precedence?

TODO: should `^` have a higher precedence than `|`?

4.8. Functor expressions

A functor expression has this form:

```
FunctorExpression := FunctorHeader? FormalParams FunctorBody
```

Functor expressions must specify a formal parameter list.

A functor expression may optionally specify annotations and/or return type. In this case, the keyword `functor` is required.

A functor body may be a single expression, or a block.

```
FunctorBody = Expression | Block
```

If the body of a non-`void` functor is a block, the value of the functor, when executed, is determined by the `produce` directive. A functor body may not contain a `return` directive.

For example:

```
functor Comparison(Float x, Float y) order = functor Comparison(Float x, Float y) { produce x<=>y };
```

The keyword `functor`, and the return type may be omitted:

```
functor Comparison(Float x, Float y) order = (Float x, Float y) { produce x<=>y };
```

The braces around the functor body and the `produce` keyword may be omitted.

```
people.sort( (Person x, Person y) y.name <=> x.name );
```

```
people.select( (Person p) p.age>18 )
    .collect( (Person p) p.name );
```

The braces are required for a `void` functor.

```
namedValues.each( (String name->Object value) { log.info("${name} ${value}"); } );
```

A functor expression can only appear as the right side of an assignment, initializer or specifier, as an argument, in a control directive, as an element of an enumeration instantiation, or surrounded by parentheses.

Consider the following functor expression:

```
functor Comparison(String x, String y) order = (String x, String y) { produce x <=> y }
```

Or, equivalently:

```
functor Comparison(String x, String y) order = (String x, String y) x <=> y;
```

These expressions are both equivalent to this Java code:

```
F2<String,String,Comparison> order =
    new F2<String,String,Comparison>() {
        public Comparison call(String x, String y) {
            return Comparison.compare(x,y);
        }
    };
```

And this functor invocation:

```
Comparison result = order("Gavin", "Emmanuel");
```

Is equivalent to this Java code:

```
Comparison result = order.call("Gavin", "Emmanuel");
```

Functors may refer to immutable locals and members of the containing class. They may not refer to mutable locals.

The following code:

```
(0..10).each( (Integer num) { log.info(num); } );
```

Is equivalent to:

```
new Range<Integer>(0, 10).each(
    new F1<Integer,Boolean>() {
        @Override Boolean call(Integer x) {
            ParentClass.this.log.info(num);
        }
    } );
```

This code:

```
Integer min = 0;
Integer max = 10;
```

```
List list = list.select( (Integer x) x > min && x < max );
```

Is equivalent to this Java code:

```
final Integer min = 0;
final Integer max = 10;
List list = list.select( new F1<Integer,Boolean>() {
        @Override Boolean call(Integer x) {
            return x > min && x < max;
        }
    } );
```

4.9. Smalltalk-style method invocation

For parameters of a method which are of functor type, Ceylon provides a special method invocation protocol, inspired by Smalltalk:

```
PartialMethodInvocation FunctorArguments | FunctorInvocation FunctorArguments?
```

The protocol begins with either:

- a normal method invocation with a period before the method name and an ordered list of parameters surrounded be parentheses, where there must be at least one parameter, or
- a special method invocation with a functor argument, which omits the period and parentheses, where the functor parameter list may be omitted for functors with no parameters.

In neither case are all parameters of the method required.

```
PartialMethodInvocation := MemberReference TypeArguments? "(" Expression ("," Expression)* ")"
```

```
FunctorInvocation := Expression ParameterName SimpleFunctorExpression
```

For example:

```
namedValues each (String name->Object value) {
    log.info("${name} ${value}");
};
```

```
Set<People> adults = people elements (Person p) p.age>18;
```

```
int index = lineItems firstIndex (LineItem li) !li.product.available;
```

```
amounts sort (Float x, Float y) { produce x<=>y };
```

```
people sort (Person p, Person q) p.name<=>q.name;
```

```
optional String label = x>10 isTrue "big";
```

Next, a named list of functor parameters may be specified:

```
FunctorArguments := (ParameterName SimpleFunctorExpression)+
```

The identifiers listed after the ordered parameter values are names of method parameters of functor type, in the order in which they appear in the method declaration. Each of these arguments is a functor expression, where the functor parameter list may be omitted for functors with no parameters:

```
SimpleFunctorExpression := FormalParams? FunctorBody
```

For example:

```
String label = x>10 isTrue "big" isFalse "little";
```

```
people select (Person p) p.name having (Person p) p.age>18;
```

```
optional specialPerson = search (people) findFirst (Person p) p.special orIfNoneFound null;
```

```
x>10
isTrue {
    log.debug("big");
    big(x);
}
isFalse {
    log.debug("little");
    little(x);
};
```

4.9.1. Iteration

A specialized invocation syntax is provided for methods which iterate collections. If the first parameter of the method is of type `Iterable<X>`, annotated `iterated`, and all remaining parameters are of functor type, then the method may be invoked according to the following protocol:

```
MemberReference "(" ForIterator ")" FunctorArguments
```

And each functor expression for a parameter annotated `coordinated` of type `functor Y(X x)` or `functor void(X x)` need not declare its parameter. Instead, its parameter is declared by the iterator.

For example, for the following method declaration:

```
public static
List<Y> from<X,Y>(iterated Iterable<X> elements,
                  coordinated functor Boolean(X x) having,
                  coordinated functor Y(X x) select);
```

We may invoke the method as follows:

```
List<String> names = from (Person p in people) having p>20 select p.name;
```

Which is equivalent to:

```
List<String> names = from (people) having (Person p) p>20 select (Person p) p.name;
```

Or, we may invoke the method as follows:

```
List<String> labels = from (Key key -> Value value in namedValues)
                      having user.authorized(key)
                      select "${key} ${value}";
```

Which is equivalent to:

```
List<String> labels = from (namedValues)
                      having (Key key -> Value value) user.authorized(key)
                      select (Key key -> Value value) "${key} ${value}";
```

4.9.2. Local definition

A specialized invocation syntax is also provided for methods which define a local. If the first parameter of the method is of type `X`, annotated `specified`, and all remaining parameters are of functor type, then the method may be invoked according to the following protocol:

```
MemberReference "(" Variable Specifier ")" FunctorArguments
```

And each functor expression for a parameter annotated `coordinated` of type `functor Y(X x)` or `functor void(X x)` need not declare its parameter. Instead, its parameter is declared by the specifier.

For example, for the following method declaration:

```
public static
Y ifExists<X,Y>(specified optional X value,
                coordinated functor Y(X x) then,
                functor Y() otherwise);
```

We may invoke the method as follows:

```
Exact amount = ifExists(Payment p = order.payment) then p.amount otherwise 0.0;
```

Which is equivalent to:

```
Exact amount = ifExists(order.payment) then (Payment p) p.amount otherwise 0.0;
```

And for the following method declaration:

```
public static
Y using<X,Y>(specified X resource,
             coordinated functor Y(X x) seek)
  where X >= Usable;
```

We may invoke the method as follows:

```
Order order = using (Session s = Session()) seek s.get(#Order, oid);
```

Which is equivalent to:

```
Order order = using (Session()) seek (Session s) s.get(#Order, oid);
```

4.9.3. Lists of cases

Finally, a specialized invocation syntax is provided for methods which define a list of cases. If the method has a parameter of type `Iterable<Entry<Case<X>, functor Y()>>` annotated `cases`, the parameter may be specified according to:

```
(case "(" Case ")" SimpleFunctorExpression)+
```

For example, for the following method declaration:

```
public static
Y select<X, Y>(X selector,
              cases Entry<Case<X>, functor Y()>... cases);
```

We may invoke the method as follows:

```
return select (payment.type)
    case (check) payByCheck(payment)
    case (card) payByCard(payment);
```

Chapter 5. Basic types

There are no primitive types in Ceylon, however, there are certain important types provided by the package `lang`. Many of these types support *operators*.

5.1. The root type

The `lang.Object` class is the root of the type hierarchy and supports the binary operators == (equals), `!=` (not equals), === (identity equals), `.` (invoke), `^.` (chain invoke), `in` (in), `is` (is), the unary prefix operator `$` (render), and the binary operator `:=` (assign).

In addition, references of type `optional:Object` support the binary operators `?.` (nullsafe invoke) and `?` (default) and the unary operator `exists`, along with the binary operators == (equals), `!=` (not equals), === (identity equals) and = (assign).

```
public abstract class Object {

    doc "The equals operator x == y. Default implementation compares
         attributes annotated |id|, or performs identity comparison."
    see #id
    public Boolean equals(Object that) {
        ...;
    }

    doc "Compares the given attributes of this instance with the given
         attributes of the given instance."
    public Boolean equals(Object that, Attribute... attributes) { ... }

    doc "The hash code of the instance. Default implementation compares
         attributes annotated |id|, or assumes identity equality."
    see #id
    public Integer hash {
        return ...;
    }

    doc "Computes the hash code of the instance using the given attributes."
    public Integer hash(Attribute... attributes) { ... }

    doc "The unary render operator $x. A developer-friendly string
         representing the instance. By default, the string contains
         the name of the type, and the values of all attributes
         annotated |id|."
    public String string { ... }

    doc "Determine if the instance belongs to the given Category."
    see #Category
    public Boolean in<Y>(Category<Y> cat)
            where (Y<=X) {
        return cat.contains(this);
    }

    doc "Determine if the instance belongs to the given Iterable
         object."
    see #Iterable
    public Boolean in<Y>(Iterable<Y> iterable)
            where (Y<=X) {
        return forAny (Object elem in iterable) some elem == this;
    }

    doc "The Type of the instance."
    public Type<X+> type { ... }

    doc "Binary assignability operator x is Y. Determine if the instance
         is of the given Type."
    public Boolean instanceOf(Type<Object+> type) {
        return this.type.assignableTo(type);
    }

    doc "A log obect for the type."
    public static Log log = Log(type);

    ...

}
```

5.2. Boolean values

The `lang.Boolean` class represents boolean values, and supports the binary `||, &&, =>, |, &, ^` and unary `!` operators.

```
public final class Boolean
        satisfies Case<Boolean> {
    instances true, false;

    doc "The binary or operator x | y"
    public Boolean or(Boolean boolean) { ... }

    doc "The binary and operator x & y"
    public Boolean and(Boolean boolean) { ... }

    doc "The binary xor operator x ^ y"
    public Boolean xor(Boolean boolean) { ... }

    doc "The unary not operator !x"
    public Boolean negation {
        return switch(this) {
            case (true) false
            case (false) true
        };
    }

}
```

5.3. Values and callables

The interface `Callable` represents a callable reference.

```
public interface Callable<out S>
        where S >= Functor {
    public S invoke;
    public onCall( ... );
}
```

The interface `Value` models a readable value.

```
public interface Value<out T> {
    public functor T() get;
    public onGet( functor T(T value) interceptor );
}
```

The interface `OpenValue` models a readable and writeable value.

```
public interface OpenValue<T>
        satisfies Value<T> {
    public functor void(T value) set;
    public onSet( functor T(T value) interceptor );
}
```

5.4. Methods and attributes

```
public interface Annotated {
    public Boolean annotated(Type<Object+> type);
    public T annotation<T>(Type<T> type);
    public Set<Object> annotations;
}
```

```
public interface Member<in X>
        satisfies Annotated {
    public Type<X> declaringType;
    public String name;
    public Visibility visibility;
    public Boolean abstract;
    public Boolean override;
}
```

```
public interface Method<in X, out S>
        where S >= Functor
        satisfies Member<X> {
    public Type<Object+> returnType;
    public List<Type<Object+>> parameterTypes;
    public Callable<S> curry(X instance);
    public Boolean static;
    public Boolean once;
    public onCall( ... );
}
```

```
public interface Attribute<in X, out T>
        satisfies Member<X> {
    public Type<T> attributeType;
    public boolean mutable;
    public Value<T> curry(X instance);
    public onGet( functor T(X instance, T value) interceptor );
}
```

```
public interface MutableAttribute<in X, T>
        satisfies Attribute<X,T> {
    public override OpenValue<T> curry(X instance);
    public onSet( functor T(X instance,T value) interceptor );
}
```

5.5. Iterable objects and iterators

The `lang.Iterable<X>` interface represents a type that may be iterated over using a `lang.Iterator<X>`. It supports the binary operator `*.` (spread).

```
public interface Iterable<out X> {

    doc "Produce an iterator."
    public Iterator<X> iterator();

}
```

```
public interface Iterator<out X> {

    public Boolean more;
    public X current;
    public X next();

}
```

Some iterable objects may support element removal during iteration.

```
public mutable interface OpenIterable<X>
        satisfies Iterable<X> {

    public override OpenIterator<X> iterator();

}
```

```
public mutable interface OpenIterator<X>
        satisfies OpenIterator<X> {

    public void remove();
    public Boolean replace(X replacement);

}
```

5.6. Cases and Selectors

The interface `lang.Case<X>` represents a type that may be used as a case in the `switch` construct.

```
public interface Case<in X> {

    public Boolean test(X value);

}
```

Classes with enumerated instances implicitly extend `lang.Selector`

```
public abstract class Selector<X>(String name, int ordinal)
        satisfies Case<X>
        where X >= Selector<X> { ... }
```

5.7. Usables

The interface `lang.Usable` represents an object with a lifecycle controlled by `try`.

```
public interface Usable {

    public void begin();
    public void end();
    public void end(Exception e);

}
```

5.8. Category, Correspondence and Container

The interface `lang.Category` represents the abstract notion of an object that contains other objects.

```
public interface Category {

    doc "Determine if the given element belongs to the category."
    public Boolean contains(Object element);

}
```

```
public extension Categories(Category category) {

    doc "Determine if all the given elements belong to the category."
    public Boolean contains(Iterable<Object> elements) {
        return forAll (X x in elements) every category.contains(x);
    }

}
```

There is a mutable subtype, representing a category to which objects may be added.

```
public mutable interface OpenCategory<in X>
        satisfies Category<X> {

    doc "Add the given element to the category."
    public Boolean add(X element);

}
```

```
public extension OpenCategories<in X>(OpenCategory<X> category) {

    doc "Add the given elements to the category."
    public Boolean add(Iterable<X> elements) {
        Boolean added = false;
        for (X x in elements) {
            added |= category.add(x);
        }
        return added;
    }

}
```

The interface `lang.Correspondence` represents the abstract notion of an object that maps value of one type to values of some other type. It supports the binary operator `[key]` (lookup).

```
public interface Correspondence<in U, out V> {

    doc "Binary lookup operator x[key]. Returns the value associated
         with the given key."
    public V value(U key);

    doc "Determine if there is a value associated with the given key."
    public Boolean defines(U key);

}
```

```
public extension Correspondences<in U, out V>(Correspondence<U, V> correspondence) {

    doc "Binary lookup operator x[keys]. Returns a list of values
         associated with the given keys, in order."
    public List<V> values(List<U> keys) {
        return from (U key in keys) select correspondence.lookup(key);
    }

    doc "Binary lookup operator x[keys]. Returns a set of values
         associated with the given set of keys."
    public Set<V> values(Set<U> keys) {
        return ( from (U key in keys) select correspondence.lookup(key) ).elements;
    }

    doc "Determine if there are values associated with all the given keys."
    public Boolean defines(Iterable<U> keys) {
        for (U key in keys) {
            if ( !correspondence.defines(key) ) {
                return false
            }
        }
        fail {
            return true;
        }
    }

}
```

There is a mutable subtype, representing a correspondence for which new mappings may be defined, and existing mappings modified. It provides for the use of element expressions in assignments.

```
public mutable interface OpenCorrespondence<in U, V>
        satisfies Correspondence<U, V> {

     doc "Element assignment x[key] = value. Assign a value to the given
          key."
     public optional V define(U key, V value);

}
```

```
public extension OpenCorrespondences<in U, V>(OpenCorrespondence<U, V> correspondence) {

     doc "Assign the given values to the given keys."
     public void define(Iterable<Entry<U, V>> definitions) {
         for (U key->V value) {
             correspondence.define(key, value);
         }
     }

     doc "Add the given entry."
     public void define(U key -> V value) {
         correspondence.define(key, value);
     }

}
```

The interface `lang.Container` represents the abstract notion of an object that may be empty. It supports the unary postfix operator `nonempty`.

```
public interface Container {

    doc "Determine if the container is empty."
    public Boolean empty;

}
```

5.9. Entries

The `Entry` class represents a pair of associated objects.

Entries may be constructed using the `->` operator.

```
public class Entry<out U, out V>(U key, V value) {

    public U key = key;
    public V value = value;

    override public Boolean equals(Object that) {
        return equals(that, Entry#key, Entry#value);
    }

    override public Integer hash {
        return hash(Entry#key, Entry#value);
    }

}
```

5.10. Collections

The interface `lang.Collection` is the root of the Ceylon collections framework.

```
public interface Collection<out X>
        satisfies Iterable<X>, Category, Container {

    public Integer size;

    public Integer count(Object element);
    public Integer count(functor Boolean(X element) having);

    public Boolean contains(functor Boolean(X element) having);

    public Set<X> elements;
    public Set<X> elements(functor Boolean(X element) having);

    public List<X> sort();
    public List<X> sort(functor Comparison(X x, X y) by);

    public OpenCollection<T> copy<T>() where T <= X;

}
```

Mutable collections implement `lang.OpenCollection`:

```
public mutable interface OpenCollection<X>
        satisfies OpenIterable<X>, OpenCategory<X>, Collection<X> {

    public Boolean clear();

    public Boolean remove(X element);
    public Integer remove(functor Boolean(X element) having);

}
```

5.10.1. Sets

Sets implement the following interface:

```
public interface Set<out X>
        satisfies Collection<X>, Correspondence<Object, Boolean> {

    public Boolean superset(Set<Object> set);
    public Boolean subset(Set<Object> set);

    public OpenSet<T> copy<T>() where T <= X;

}
```

There is a mutable subtype:

```
public mutable interface OpenSet<X>
        satisfies Set<X>, OpenCollection<X>, OpenCorrespondence<Object, Boolean> {}
```

5.10.2. Lists

Lists implement the following interface, and support the binary operators `+` (join), `[...j]`, `[i...]` (lower, upper range) and the ternary operator `[i..j]` (subrange) in addition to operators inherited from `Collection` and `Correspondence`:

```
public interface List<out X>
        satisfies Collection<X>, Correspondence<Integer, X> {

    public X first;
    public X last;

    public optional X firstOrNull;
    public optional X lastOrNull;

    public Integer firstIndex;
    public Integer lastIndex;

    public Integer firstIndex(functor Boolean(X element) having);
    public Integer lastIndex(functor Boolean(X element) having);

    public optional Integer firstIndexOrNull(functor Boolean(X element) having);
    public optional Integer lastIndexOrNull(functor Boolean(X element) having);

    public List<X> head(Integer to=firstIndex);
    public List<X> tail(Integer from=lastIndex);

    doc "The binary range operator x[from..to]"
    public List<X> range(Integer from, Integer to);

    doc "The binary join operator x + y"
    public List<T> join(List<T> elements) where T<=X;

    public List<X> sublist(Integer from, Integer to);
    public List<X> reversed;
    public Bag<X> unsorted;
    public Map<Integer,X> map;

    public List<Y> transform<Y>(functor Y(X element) select);

    public OpenList<T> copy<T>() where T <= X;

}
```

There is a mutable subtype:

```
public mutable interface OpenList<X>
        satisfies List<X>, OpenCollection<X>, OpenCorrespondence<Integer, X> {

    public void prepend(X element);
    public void prepend(List<X> elements);
    public void append(X element);
    public void append(List<X> elements);

    public void insert(Integer at, X element);
    public X removeIndex(Integer at);

    public void delete(Integer from, Integer to);
    public void truncate(Integer from);

    public X removeFirst();
    public X removeLast();

    public void reverse();
    public void resort();
    public void resort(functor Comparison(X x, X y) by);

    override public OpenList<X> sublist(Integer from, Integer to);
    override public OpenList<X> reversed;
    override public OpenMap<Integer,X> map;

}
```

5.10.3. Maps

Maps implement the following interface:

TODO: is it OK that maps are not contravariant in U?

```
public interface Map<U, out V>
    satisfies Collection<Entry<U,V>>, Correspondence<U, V> {

    public Set<U> keys;
    public Bag<V> values;
    public Map<V, Set<U>> inverse;

    public optional V valueOrNull(U key);

    public Map<U, W> transform<W>(functor optional W(U key -> V value) select);

    public Map<U, V> entries(functor Boolean(U key -> V value) having);

    public OpenMap<U,T> copy<U,T>() where T <= V;

}
```

There is a mutable subtype:

```
public mutable interface OpenMap<U,V>
        satisfies Map<U,V>, OpenCollection<Entry<U,V>>, OpenCorrespondence<U, V> {

    override public OpenSet<U> keys;
    override public OpenBag<V> values;
    override public OpenMap<V, Set<U>> inverse;

    public optional V removeKey(U key);
    public Bag<V> removeKeys(functor Boolean(U key) having);

}
```

5.10.4. Bags

Bags implement the following interface:

```
public interface Bag<out X>
        satisfies Collection<X>, Correspondence<Object, Integer> {

    public OpenBag<T> copy<T>() where T <= X;

    public Map<X,Integer> map;

}
```

There is a mutable subtype:

```
public mutable interface OpenBag<X>
        satisfies Bag<X>, OpenCollection<X>, OpenCorrespondence<Object, Integer> {

    override public OpenMap<X,Integer> map;

}
```

5.10.5. Collection operations

```
public Collections {

    public static List<X> join<X>(List<X> list...) {
        return new List<X> {
            ...
        };
    }

    public static Bag<X> union<X>(Bag<X> bag...) {
        return new Bag<X> {
            ...
        };
    }

    public static Set<X> union<X>(Set<X> set...) {
        return new Set<X> {
            ...
        };
    }

    public static Set<X> intersection<X>(Set<X> set...) {
        return new Set<X> {
            ...
        };
    }

    public static Set<X> complement<X>(Set<X> set, Set<Object> sets...) {
        return new Set<X> {
            ...
        };
    }

}
```

5.11. Ordered values

The `lang.Comparable<T>` interface represents totally ordered types, and supports the binary operators `>`, `<`, `<=`, `>=` and `<=>` (compare).

```
public interface Comparable<in T> {

    doc "The binary compare operator <=>. Compares this
         instance with the given instance."
    public Comparison compare(T other);

}
```

```
public class Comparison {

    instances larger, smaller, equal;

    public Boolean larger return this==larger;
    public Boolean smaller return this==smaller;
    public Boolean equal return this==equal;
    public Boolean unequal return this!=equal;
    public Boolean largeAs return this!=smaller;
    public Boolean smallAs return this!=larger;

}
```

The `lang.Ordinal<T>` interface represents objects in a sequence, and supports the binary operator `..` (range) and postfix unary operators `++` (successor) and `--` (predecessor). In addition, variables support the prefix unary operators `++` (increment) and `--` (decrement).

```
public interface Ordinal<T> {

    doc "The unary "++" operator. The successor of this instance."
    public T successor;

    doc "The unary "--" operator. The predecessor of this instance."
    public T predecessor;

}
```

5.12. Ranges and enumerations

Ranges and enumerations both implement `List`, therefore they support the join, subrange, contains and lookup operators, among others. Ranges may be constructed using the `..` operator:

```
public class Range<X>(X first, X last)
        satisfies List<X>, Case<Object>
        where X>=Ordinal & X>=Comparable {

    public X first = first;
    public X last = last;

    ...
}
```

Enumerations represent an explicit list of values and may be constructed using a simplified syntax:

```
public class Enumeration<out X>(X... values)
        satisfies List<X>, Case<Object> {
    ...

    public static extension
    OpenMap<U, V> toOpenMap<U, V>(Enumeration<Entry<U,V>> enum) {
        return HashMap(enum);
    }

    public extension
    OpenSet<X> toOpenSet<X>() {
        return elements.copy();
    }

    public extension
    OpenList<X> toOpenList<X>() {
        return copy();
    }

    public static extension
    Enumeration<T> singleton<T>(T object) {
        return Enumeration(object);
    }

}
```

5.13. Characters and strings

Characters are represented by the following class:

```
public class Character
        satisfies Ordinal<Character>, Comparable<Character>, Case<Character> { ... }
```

Strings implement `List`, therefore they support the join, subrange, contains and lookup operators, among others.

```
public class String(Character... characters)
        satisfies Comparable<String>, List<Character>, Case<String> {

    ...

    public Iterable<String> tokens(Iterable<Character> separators=" ,;\n\l\r\t") { return ...; }

    public String lowercase { return ...; }
    public String uppercase { return ...; }

    public String strip(Iterable<Character> whitespace = " \n\l\r\t") { return ...; }
    public String normalize(Iterable<Character> whitespace = " \n\l\r\t") { return ...; }

    public String join(String... strings) { return ...; }
    public String join(Iterable<String> strings) { return ...; }

}
```

TODO: do we need character ranges, like `'a'..'z'`, or is it enough that we have regular expressions like `` `[a-z]` ``?

5.14. Regular expressions

```
public class Regex
        satisfies Case<String> {
    public List<Match> matchList(String string);
    public Boolean matches(String string);
    ...
}
```

TODO: I assume these are just Java (Perl 5-style) regular expressions. Is there some other better syntax around?

5.15. Numbers

The `lang.Number<T>` interface represents numeric values, and supports the binary operators `+`,`-`, `*`, `/`, `%`, `**`, and the unary prefix operators `-`, `+`. In addition, variables of type `lang.Number` support `+=`, `-=`, `/=`, `*=`.

```
public interface Number<T>
        satisfies Comparable<Number<?>>, Ordinal<T> {

    //binary "+" operator
    public T add(T number);
    public Number add(Number number);

    //binary "-" operator
    public T subtract(T number);
    public Number subtract(Number number);

    //binary "*" operator
    public T multiply(T number);
    public Number multiply(Number number);

    //binary "/" operator
    public T divide(T number);
    public T divide(Number number);

    //binary "%" operator
    public T remainder(T number);
    public T remainder(Number number);

    //unary "-" operator
    public T inverse;

    public T magnitude;

    public Boolean integral;
    public Boolean positive;
    public Boolean negative;
    public Boolean zero;
    public Boolean unit;

    public Exact exact;
    public Whole whole;
    public Natural natural;
    public Integer integer;
    public Float float;
    public Double double;
    public Long long;

    public Exact fractionalPart;
    public Integer scale;
    public Integer precision;

}
```

Seven numeric types are built in:

```
public final class Natural
        satisfies Number<Natural>, Case<Natural> { ... }
```

```
public final class Integer
        satisfies Number<Integer>, Case<Integer> {
    ...

    public void times(Iteration iteration) { ... }
    public void upto(Integer max, Iteration iteration) { ... }
    public void downto(Integer min, Iteration iteration) { ... }

}
```

```
public final class Long
        satisfies Number<Long>, Case<Long> { ... }
```

```
public final class Float
        satisfies Number<Float> { ... }
```

```
public final class Double
        satisfies Number<Double> { ... }
```

```
public class Exact
        satisfies Number<Exact> { ... }
```

```
public class Whole
        satisfies Number<Whole>, Case<Whole> { ... }
```

5.16. Instants, intervals and durations

TODO: this stuff is just for illustration, the real date/time API will be much more complex and fully internationalized.

```
public class Instant {
    ...
}
```

```
public class Time(Integer hours, Integer minutes,
        optional Integer seconds=null, optional Integer milliseconds=null,
        optional Integer timezone=null)
        extends Instant {
    public Integer hours = hours;
    public Integer minutes = minutes;
    public optional Integer seconds = seconds;
    public optional Integer milliseconds = milliseconds;
    public optional Timezone timezone = timezone;
    ...
}
```

```
public class Date(Integer year, Integer month, Integer day)
        extends Instant {
    public Integer year = year;
    public Integer month = month;
    public Integer day = day;
    ...
}
```

```
public class Datetime(Time time, Date date)
        extends Instant {
    public Time time = time;
    public Date date = date;
    ...
}
```

```
public class Interval<X>(X start, X end)
        where X >= Instant {
    public X start = start;
    public X end = end;
    public Duration<X> duration { return ...; }
    ...
}
```

```
public class Duration<X>(Map<Granularity<X>, Integer> magnitude)
        where X >= Instant {

    public Map<Granularity<X>, Integer> magnitude = magnitude;

    public X before(X instant) { ... }
    public X after(X instant) { ... }

    public Datetime before(Datetime instant) { ... }
    public Datetime after(Datetime instant) { ... }

    public Duration<X> add(Duration<X> duration) { ... }
    public Duration<X> subtract(Duration<X> duration) { ... }

    ...
}
```

```
public interface Granularity<X>
        where X >= Instant {}
```

```
public class DateGranularity
        satisfies Granularity<Date>
        instances year, month, week, day {}
```

```
public class DateGranularity
        satisfies Granularity<Time>
        instances hour, minute, second, millisecond {}
```

5.17. Control expressions

The `lang` package defines several classes containing static methods for building complex expressons.

```
public class Assertions {

    doc "Assert that the block evaluates to true. The block
         is executed only when assertions are enabled. If
         the block evaluates to false, throw an
         |AssertionException| with the given message."
    public static void assert(functor String() message
                              functor Boolean() that) {
        if ( assertionsEnabled() && !evaluate() ) {
            throw new AssertionException( message() );
        }
    }

}
```

```
public class Conditionals {

    doc "If the condition is true, evaluate first block,
         and return the result. Otherwise, return a null
         value."
    public static optional Y ifTrue<Y>(Boolean condition,
                                        functor Y() then) {
        if (condition) {
            return then();
        }
        else {
            return null;
        }
    }

    doc "If the condition is true, evaluate first block,
         otherwise, evaluate second block. Return result
         of evaluation."
    public static Y ifTrue<Y>(Boolean condition,
                              functor Y() then,
                              functor Y() otherwise) {
        if (condition) {
            return then();
        }
        else {
            return otherwise();
        }
    }

    doc "If the value is non-null, evaluate first block,
         and return the result. Otherwise, return a null
         value."
    public static optional Y ifExists<X,Y>(specified optional X value,
                                            coordinated functor Y(X x) then) {
        if (exists value) {
            return then(value);
        }
        else {
            return null;
        }
    }

    doc "If the value is non-null, evaluate first block,
         otherwise, evaluate second block. Return result
         of evaluation."
    public static Y ifExists<X,Y>(specified optional X value,
                                  coordinated functor Y(X x) then,
                                  functor Y() otherwise) {
        if (exists value) {
            return then(value);
        }
        else {
            return otherwise();
        }
    }

    doc "Evaluate the block which matches the selector value, and
         return the result of the evaluation. If no block matches
         the selector value, return a null value."
    public static optional Y select<X,Y>(X selector,
                                          cases Iterable<Entry<Case<X>, functor Y()>> value) {
        for (Case<X> match -> functor X() evaluate in value) {
            if ( match.test(selector) ) {
                return evaluate(value);
            }
        }
        return null;
    }

    doc "Evaluate the block which matches the selector value, and
         return the result of the evaluation. If no block matches
         the selector value, evaluate the last block and return
         the result of the evaluation."
    public static Y select<X,Y>(X selector,
                                cases Iterable<Entry<Case<X>, functor Y()>> value
                                functor Y() otherwise) {
        if (exists Y y = select(selector, value)) {
            return y;
        }
        else {
            return otherwise();
        }
    }

}
```

```
public class Quantifiers {

    doc "Count the elements for with the block evaluates to true."
    public static Integer count<X>(iterated Iterable<X> elements,
                                   coordinated functor Boolean(X x) having) {
        mutable Integer count := 0;
        for (X x in elements) {
            if ( having(x) ) {
                ++count;
            }
        }
        return count;
    }

    doc "Return true iff for every element, the block evaluates to true."
    public static Boolean forAll<X>(iterated Iterable<X> elements,
                                    coordinated functor Boolean(X x) every) {
        for (X x in elements) {
            if ( !every(x) ) {
                return false;
            }
        }
        return true;
    }

    doc "Return true iff for some element, the block evaluates to true."
    public static Boolean forAny<X>(iterated Iterable<X> elements,
                                    coordinated functor Boolean(X x) some) {
        return !forAll(elements, (X x) !some(x));
    }

    doc "Return the first element for which the block evaluates to true,
         or a null value if no such element is found."
    public static optional X first<X>(iterated Iterable<X> elements,
                                      coordinated functor Boolean(X x) having) {
        for (X x in elements) {
            if ( having(x) ) {
                return x;
            }
        }
        return null;
    }

    doc "Return the first element for which the first block evaluates to
         true, or the result of evaluating the second block, if no such
         element is found."
    public static X first<X>(iterated Iterable<X> elements,
                             coordinated functor Boolean(X x) having
                             functor X() otherwise) {
        if(exists X first = first(elements, having)) {
            return first;
        }
        else {
            return otherwise();
        }
    }

}
```

```
public class ListComprehensions {

    doc "Iterate elements and return those for which the first
         block evaluates to true, ordered using the second block,
         if specified."
    public static List<X> from<X>(iterated Iterable<X> elements,
                                  coordinated functor Boolean(X x) having,
                                  optional coordinated functor Comparison(X x, X y) by = null) {
        return from(elements, having, (X x) x, by);
    }

    doc "Iterate elements and for each element evaluate the first block.
         Build a list of the resulting values, ordered using the second
         block, if specified."
    public static List<Y> from<X,Y>(iterated Iterable<X> elements,
                                    coordinated functor Y(X x) select,
                                    optional coordinated functor Comparable(X x) by = null)  {
        return from(elements, (X x) true, select, by);
    }

    doc "Iterate elements and select those for which the first block
         evaluates to true. For each of these, evaluate the second block.
         Build a list of the resulting values, ordered using the third
         block, if specified."
    public static List<Y> from<X,Y>(iterated Iterable<X> elements,
                                    coordinated functor Boolean(X x) having,
                                    coordinated functor Y(X x) select,
                                    optional coordinated functor Comparable(Y x, Y y) by = null) {
        OpenList<Y> list = ArrayList<Y>();
        for (X x in elements) {
            if ( having(x) ) {
                list.add( select(x) );
            }
        }
        if (exists by) {
            list.sort(by);
        }
        return list;
    }

}
```

```
public class MapComprehensions {

    doc "Construct a |Map| by evaluating the block for each given key.
         Each |Entry| is constructed from the key and the value result
         of the evaluation."
    public static Map<U,V> mapFrom<U,V>(iterated Iterable<U> keys,
                                        coordinated functor V(U key) to) {
        return map(keys, (U key) key->to(key));
    }

    doc "Construct a |Map| by evaluating the block for each given value.
         Each |Entry| is constructed from the value and the key result
         of the evaluation."
    public static Map<U,V> mapTo<U,V>(iterated Iterable<V> values,
                                      coordinated functor V(U key) from) {
        return map(values, (V value) value->from(value));
    }

    doc "Construct a |Map| by evaluating the block for each given object
         and collecting the resulting |Entry|s."
    public static Map<U,V> map<X,U,V>(iterated Iterable<X> elements,
                                      coordinated functor Entry<U,V>(X element) of) {
        OpenMap<U,V> map = HashMap<U,V>();
        for (X x in elements) {
            map.add( of(x) );
        }
    }

}
```

```
public class Handlers {

    doc "Attempt to evaluate the first block. If an exception occurs that
         matches the second block, evaluate the block."
    public static Y seek<Y,E>(functor Y() seek,
                              functor Y(E e) except) {
        try {
            return seek();
        }
        catch (E e) {
            return except(e);
        }
    }

    doc "Using the given resource, attempt to evaluate the first block."
    public static Y using<X,Y>(specified X resource,
                               coordinated functor Y(X x) seek)
                  where X >= Usable {
        try (resource) {
            return seek(resource);
        }
    }

    doc "Using the given resource, attempt to evaluate the first block.
         If an exception occurs that matches the block, evaluate the
         second block."
    public static Y using<X,Y,E>(specified X resource,
                                 coordinated functor Y(X x) seek,
                                 functor Y(E e) except)
                  where X >= Usable {
        try (resource) {
            return seek(resource);
        }
        catch (E e) {
            return except(e);
        }
    }

}
```

5.18. Primitive type optimization

For certain types, the Ceylon compiler is permitted to transform local declarations to Java primitive types, literal values to Java literals, and operator invocations to use of native Java operators, as long as the transformation does not affect the semantics of the code.

For this example:

```
Integer calc(Integer j) {
    Integer i = list.size;
    i++;
    return i * j + 1000;
}
```

the following equivalent Java code is acceptable:

```
Integer calc(Integer j) {
    int i = list.size().get();
    i++;
    return new Integer( i * lang.Util.intValue(j) + 1000 );
}
```

The following optimizations are allowed:

- `lang.Integer` to Java `int`
- `lang.Long` to Java `long`
- `lang.Float` to Java `float`
- `lang.Double` to Java `double`
- `lang.Boolean` to Java `boolean`

However, these optimizations may never be performed for locals, attributes or method types declared `optional`.

The following operators may be optimized: `+, -, *, /, ++, --, +=, -=, *=, /=, >, <, <=, >=, ==, &&, ||, !`.

Finally, integer, float and boolean literals may be optimized.