Project Ceylon

A Better Java

Version: For internal discussion only

Table of Contents

A work in progress iv
1. Introduction 1
1.1. A simple example 1
2. Declarations 3
2.1. General declaration syntax 3
2.1.1. Abstract declaration 3
2.1.2. Annotation list 3
2.1.3. Type declaration 3
2.1.4. Formal parameter list 4
2.1.5. Extended class 4
2.1.6. Implemented interfaces 4
2.1.7. Generic type constraint list 5
2.2. Classes 5
2.2.1. Class inheritance 5
2.2.2. Class instantiation 6
2.2.3. Optional parameters 7
2.2.4. Instance enumeration 8
2.3. Interfaces 9
2.3.1. Interface inheritance 9
2.4. Methods 9
2.4.1. Optional parameters 10
2.4.2. Interface methods and abstract methods 11
2.5. Attributes 11
2.5.1. Attributes with getter/setter code 12
2.5.2. Simple attributes and locals 12
2.5.3. Interface attributes and abstract attributes 13
2.6. Decorators 13
2.6.1. Introduced methods 14
2.6.2. Introduced types 14
2.7. Type aliases 15
2.8. Converters 15
2.9. Declaration modifiers 16
3. Blocks and control structures 17
3.1. Blocks and statements 17
3.2. Control directives 17
3.3. Control structures 17
3.3.1. if/else 18
3.3.2. switch/case/else 19
3.3.3. for/fail 20
3.3.4. do/while 20
3.3.5. try/catch/finally 21
4. Expressions 23
4.1. Literals 23
4.1.1. Datetime literals 23
4.1.2. Integer literals 24
4.1.3. Float literals 24
4.1.4. String literals 24
4.1.5. Character literals 24
4.1.6. Regex literals 25
4.1.7. Enumeration literals 25
4.1.8. Object literals 25
4.1.9. Type and member literals 26
4.2. Invocations 26
4.2.1. Method invocation 27
4.2.2. Static method invocation 27
4.2.3. Class instantiation 27
4.2.4. Enumeration instantiation 28

4.2.5. Inline classes 28
4.2.6. Attribute configuration 28
4.2.7. Functor invocation 28
4.2.8. Attribute access 29
4.3. Assignable expressions 29
4.4. Operators 30
4.4.1. List of operators 31
4.4.2. Operator precedence and associativity 34
4.5. Functor expressions 35
4.5.1. Syntax extension for functor parameters 36
4.5.1.1. Syntax extension for iterations 37
4.5.1.2. Syntax extension for variable definition 37
4.5.1.3. Syntax extension for lists of cases 38
5. Basic types 39
5.1. The root type 39
5.2. Types 39
5.3. Boolean values 40
5.4. Values and callables 40
5.5. Methods and attributes 40
5.6. Iterable objects and iterators 41
5.7. Cases and Selectors 41
5.8. Usables 42
5.9. Category, Correspondence and Container 42
5.10. Entries 43
5.11. Collections 44
5.11.1. Sets 44
5.11.2. Lists 44
5.11.3. Maps 45
5.11.4. Bags 46
5.12. Ordered values 46
5.13. Ranges and enumerations 46
5.14. Characters, character strings and ranges 47
5.15. Regular expressions 47
5.16. Numbers 48
5.17. Instants, intervals and durations 49
5.18. Control expressions 50
5.19. Primitive type optimization 51

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: 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 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.

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 significantly enhanced versions of the traditional constructs found in C, C# and Java, including features inspired by Python. Unlike C or Java, Ceylon's control flow structures may be used to produce a value, eliminating the need for many locals.

Ceylon features a large 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.

Ceylon supports closures, called *functors*.

True open classes are not supported. However, Ceylon supports *decorators*, which allows addition of methods and interfaces to existing types, and overriding of existing methods. Decorators only affect the behavior of a type, not its state.

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

Ceylon introduces certain syntax extensions that support the definition of domain-specific languages and expression of structured data. These syntax extensions include significantly improved support for expressing literal values, compared to Java. 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`.

Finally, Ceylon provides an extensible type conversion facility that allows values of different types to be treated as interchangeable by the compiler. This facility makes it easy for Ceylon code to transparently interoperate and inter-compile with Java code.

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 sets the values of its attributes. The `page` annotation specifies the URL at which this HTML should be accessible.

Chapter 2. Declarations

All classes, interfaces, methods, attributes, locals, decorators and converters 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*
Type? keyword? Identifier TypeParams? FormalParams?
Supertype?
Interfaces?
TypeConstraints?
Declaration?
```

2.1.2. Annotation list

Declarations may be preceded by a list of annotations. An annotation is just an instantiation of a class.

```
Annotation :=
"@" Instantiation |
TypeName (StringLiteral|TypeLiteral|MemberLiteral|IntegerLiteral|FloatLiteral|RegexLiteral)?
```

For an annotation with no constructor parameters, or with just one string, type, member, integer, float or regex literal constructor parameter, the `@` and parentheses may be omitted.

For example:

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

2.1.3. Type declaration

Method, attribute and converter declarations must declare a type.

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

Most types are classes or interfaces:

```
RegularType := TypeName TypeParams?
```

```
TypeName := Identifier (DOT Identifier)*
```

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

```
TypeParams := "<" Type (COMMA Type)* ">"
```

For example:

```
lang.collections.Map<Key, List<Item>>
```

There are also functor types:

```
FunctorType := "(" ( Variable (COMMA Variable)* )? ")" ("produces" Annotation* Type | "void")
```

For example:

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

```
(Payment payment) produces Boolean
```

```
(Y element) void
```

```
() produces optional Object
```

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

2.1.4. Formal parameter list

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

```
FormalParams :=
"("
FormalParam (COMMA FormalParam)* (COMMA OptionalParam)* (COMMA VarargsParam)? |
OptionalParam (COMMA OptionalParam)* (COMMA VarargsParam)? |
VarargsParam?
")"
```

```
OptionalParam := FormalParam Initializer
```

```
Initializer := "=" Expression
```

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

```
FormalParam := Variable | VariablePair
```

```
Variable := Annotation* Type Identifier
```

```
VariablePair := Annotation* Type Identifier "->" Type Identifier
```

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>`.

2.1.5. Extended class

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

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

For example:

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

2.1.6. Implemented interfaces

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

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

For example:

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

2.1.7. Generic type constraint list

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

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

```
TypeConstraint := Ident ( (">="|"<=") (Type|Ident) | 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)
```

2.2. Classes

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

```
Annotation*
"class" Identifier TypeParams? FormalParams?
Supertype?
Interfaces?
TypeConstraints?
InstanceEnumeration?
Block
```

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.

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;
    public mutable Exact y;
}
```

Is equivalent to this Java class with a default constructor:

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

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

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 = if (position.negative) -1 else 1;

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

    doc "must have distance ${position} from origin"
    assert 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(Exact position) {
        Exact displacement = position.exponentiate(2).divide(2);
        Integer sign = position.negative ? -1 : 1;

        x = sqrt(displacement).multiply(sign);
        y = sqrt(displacement).multiply(sign);
        assert x.exponentiate(2) + y.exponentiate(2)
             == position.exponentiate(2) :
        "must have distance " + position + " from origin";
    }

}
```

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

2.2.3. Optional parameters

When a class with an optional parameter is instantiated, and a value is not assigned to the optional parameter by the caller, the default value specified by the initializer 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. Instance enumeration

A class may specify an enumerated list of instances:

```
InstanceEnumeration :=
"instances"
Identifier ParamValues?
(COMMA Identifier ParamValues?)*
"..."?
```

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" Identifier 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);
}
```

TODO: Fantom and Scala let interfaces declare methods with implementation. This is less needed in Ceylon because we have decorators. But it seems pretty harmless and useful.

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 accord-

ing to the following:

```
Method :=
Annotation*
(Type | "void") Identifier 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) { ... }
}
```

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

If there is no method body, and the method is not declared `abstract`, the method throws `UnsupportedMethodException` when invoked.

The Ceylon compiler preserves the names of method parameters.

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

TODO: should we allow a method or attribute getter/setter to omit the braces if it consists of exactly one statement. (What about converters?)

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. Optional parameters

Methods with optional parameters may not be overloaded.

When a method with an optional parameter is called, and a value is not assigned to the optional parameter by the caller, the default value specified by the initializer 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") Identifier 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 String name { return join(firstName, lastName); }
public assign name { firstName = first(name); lastName = last(name); }
```

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

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.

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 Identifier 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" Identifier 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 Identifier Initializer? ";"
```

Formal parameters of classes, methods, decorators and converters 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.

- An formal parameter of a class or decorator represents a local if it is not used inside a method, attribute setter or attribute getter declaration.
- A formal parameter of a method or converter 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; }
```

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

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 Identifier ";"
```

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. Decorators

A decorator introduces methods and interfaces to a given type. Decorators are declared according to the following:

```
Annotation*
"decorator" Identifier TypeParams? "(" FormalParam ")"
Interfaces?
TypeConstraintList?
"{" ( Method | AttributeGetter | AttributeSetter )* "}"
```

Decorators may introduce interfaces and methods.

2.6.1. Introduced methods

Methods declared by a decorator are called *introduced methods*.

For example, this decorator:

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

    public Collection<T> nonZeroElements() {
        return collection.exclude(0);
    }

    ...
}
```

is equivalent to this Java declaration:

```
public final class CollectionUtils<T>
        extends Collection {

    private final Collection<T> collection

    public CollectionUtils(Collection<T> collection) {
        this.collection = collection;
    }

    public Collection<T> nonZeroElements() {
        return collection.exclude(0);
    }

    ...
}
```

This introduced method call:

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

Is equivalent to this Java code:

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

Introduced methods are only available in a source file that expicitly `imports` the decorator.

A decorator may not override a method defined by the decorated type. Methods defined by the decorated type are delegated to the decorated instance. TODO: is this the right thing to say??

2.6.2. Introduced types

TODO: there are holes in the semantics of this stuff!

Types appearing after the `satisfies` keyword must be interfaces, and are called the *introduced types*.

This decorator declaration:

```
public decorator ComparableUser(User user)
        satisfies Comparable<User> {

    public Comparison compare(User other)
    {
        return user.username <=> other.username;
    }

}
```

is equivalent to this Java declaration:

```
public final class ComparableUser
        extends User
        implements Comparable<User> {

    private final User user;
```

```
    public ComparableUser(User user) {
        this.user = user;
    }

    public Comparison compareTo(User other) {
        return Comparison.compare(other.username, user.username);
    }
}
```

2.7. Type aliases

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

```
Annotation* "alias" Identifier 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 (T x,T y) produces Comparison;
```

2.8. Converters

A converter allows immutable values of one type to be transparently converted to values of another type. Converters are declared according to the following:

```
Annotation* Type
"converter" Identifier TypeParams?
TypeConstraintList?
"(" FormalParam ")"
Block
```

TODO: would it be better to just let you annotate a static method `converter:`, instead of having a whole special declaration?

For example:

```
public User converter PersonUser(Person person) { return person.user; }
```

The parameter type must be immutable.

A converter declaration is equivalent to a Java class declaration. The example above is equivalent to the following Java class:

```
public final class PersonUser
        extends Converter<Person, User> {
    @Override User convert(Person person)
    {
        return person.user;
    }
}
```

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

ment:

```
Person person = ...;
User user = person;
```

Is equivalent to the following Java code:

```
Person person = ...;
User user = new PersonUser().convert(person);
```

A conversion is only available in a source file that expicitly `imports` the converter.

2.9. 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.
- `once` indicates that a method is executed at most once, and the resulting value is cached.
- `assert` indicates that a statement must evaluate to true when assertions are enabled, and is not executed when assertions are disabled.
- `deprecated` indicates that a method, attribute or type is deprecated.

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.

Chapter 3. Blocks and control structures

Method, attribute, converter, class and functor literal bodies contain procedural code that is executed when the method, attribute or converter is invoked. 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 expressions, local initializers and control structures are executed sequentially.

A *statement* is an expression, control structure, control statement or a method, attribute or local declaration.

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

```
Statement := Annotation* Expression ";" | ControlStructure | Declaration
```

```
ControlStatement := ControlDirective ";"
```

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

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, attribute getter or converter,
- the `produces` 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.

```
ControlDirective := ("return" | "produces") Expression | "throw" Expression | "found" | "break"
```

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

The `produces` 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, exception management and assertions. 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.

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/fail` 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
```

Control structures are not considered to be expressions in Ceylon. However, all control structures have a corresponding *control expression* which may be used anywhere an expression may be used.

A control structure body consists of an expression, a control directive or a block.

```
OpenBlock := Block | Expression | ControlDirective
```

```
ClosedBlock := Block | (Expression | ControlDirective) ";"
```

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

Some control structures expect conditions:

```
Condition := Expression | (Variable Initializer | Expression) ("exists"|"nonempty")
```

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

- If no `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 `exists` modifier appears, the condition must be an expression or local initializer of type `optional Object`. The condition is satisfied if the expression or initializer evaluates to a non-null value at runtime.
- If the `nonempty` modifier appears, the condition must be an expression or local initializer of type `optional Container`. The condition is satisfied if the expression or initializer 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 initializer, the local variable may be declared without the `optional` annotation, even though the initializer 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.

3.3.1. `if/else`

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

```
IfElse := "if" "(" Condition ")" (ClosedBlock | OpenBlock "else" ClosedBlock)
```

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 (user exists)
        log.info("Hi ${user.name}!")
    else
        log.info("Hello World!");
}
```

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

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

3.3.2. `switch/case/else`

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

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

```
Cases := CaseNull? Case+ CaseElse?
```

```
CaseNull := "case" "null" OpenBlock
```

```
Case := "case" "(" Case ")" OpenBlock
```

```
CaseElse := "else" OpenBlock
```

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.

```
Case := Expression (COMMA Expression)*
```

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 classs must be explicitly listed.

If the switch expression is of type `Selector`, and all values of the selector 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 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");
    }
}
```

3.3.3. `for/fail`

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

```
ForFail := "for" "(" ForIterator ")" (ClosedBlock | OpenBlock "fail" ClosedBlock)
```

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

```
ForIterator := (Variable | VariablePair) "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");
}
```

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

3.3.4. `do/while`

The `do/while` loop has the form:

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

```
DoIterator := 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 (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();
}
```

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

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

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

TODO: do/while/fail 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 ")")?
( ClosedBlock | OpenBlock (Catch OpenBlock)* (Catch ClosedBlock | "finally" ClosedBlock) )
```

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`.

```
Catch := "catch" "(" Variable ")"
```

The resource expression must be of type `Usable`.

```
Resource := Variable Initializer | 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 ( Transaction() ) try ( Session s = Session() ) {
    Person p = s.get(#Person, id);
    ...
}
```

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

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

```
try ( Transaction() ) try ( 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,
- operators and control expressions, 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 |
EnumerationLiteral |
"this" | "super" | "null"
```

The keywords `this`, `super` and `null` are equivalent to the Java forms.

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';
```

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('+') ) ) { ... }
```

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. Enumeration literals

The following literal is supported, representing an empty enumeration:

```
"none"
```

For example:

```
Enumeration<String> enum = none;
```

Equivalent to this Java code:

```
Enumeration<String> enum = collections.Enumeration.emptyEnumeration<String>();
```

TODO: is this really a special literal, or is it just the single enumerated instance of the `EmptyEnumeration` class, which has a converter to `Enumeration<X>`?

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 converters 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 = none;
```

4.1.8. Object literals

There are no true literals for objects. Rather, there is a nice syntax for calling the class constructor, assigning attribute values (including constant attribute values) and (optionally) overriding methods and attributes. For example:

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

```
};
```

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

4.1.9. 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 Identifier
```

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<(P p,Q q,...) produces R>` 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. 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.

```
ParamValues := OrderedParamValues | NamedParamValues
```

Required parameters must be specified. Optional 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 optional parameters, in the order they were declared. If there are any remaining optional parameters, they will be assigned their default values. On the other hand, if any parameter values are unassigned, they will be treated as varargs.

```
OrderedParamValues := "(" ( Expression (COMMA Expression)* )? ")"
```

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

```
NamedParamValues := "{" NamedParamValue* ( VarargParamValue (COMMA VarargParamValue)* )? "}"
```

```
NamedParamValue := Identifier Initializer ";"
```

```
VarargParamValue := Expression | Variable Initializer
```

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`.

4.2.1. Method invocation

Method invocations follow the following schema.

```
MethodInvocation := MethodReference TypeParams? ParamValues
```

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

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`.

4.2.2. Static method invocation

Static method invocations follow the following schema.

```
StaticMethodInvocation := (RegularType DOT)? Identifier TypeParams? ParamValues
```

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.

4.2.3. Class instantiation

Classes may be instantiated according to the following schema:

```
Instantiation := RegularType ParamValues
```

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).

4.2.4. Enumeration instantiation

Enumerations may be instantiated according to the following simplified syntax:

```
EnumerationInstantiation := "{" Expression (COMMA Expression)* "}"
```

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

For example:

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

4.2.5. Inline classes

Inline classes may be instantiated according to:

```
InlineClass := Annotation* Instantiation Interfaces Block
```

For example:

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

```
return 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 named parameter values are assigned directly to attributes of the superclass.

4.2.6. Attribute configuration

Attribute configuration follows the following schema.

```
AttributeConfiguration := Expression NamedParamValues
```

For example:

```
person { firstName="Gavin"; initial='A'; lastName="King"; }
```

The value of an attribute configuration is the instance being configured. The parameter values are assigned to attributes of the instance.

4.2.7. Functor invocation

Functor invocations follow the following schema.

```
FunctorInvocation := Expression ParamValues
```

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.

4.2.8. Attribute access

Attribute get access follows the following 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 "=" 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.

4.3. 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) );
```

4.4. 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 ... ;
```

```
if ( x>10 || x<0 ) { ... }
```

```
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 ( if (name exists) name.equals(value) else if (value exists) false else true ) return ... ;
```

```
if ( x.compare(10).larger ) true else x.compare(0).smaller ) { ... }
```

```
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 (person exists) person.name else null;
```

4.4.1. List of operators

The following table defines the semantics of the Ceylon operators:

TODO: if there's one thing C syntax got wrong, it's the use of `=` for assignment. `:=` is a much, much better choice. Should we fix this, or is consistency with other C-like languages more important? One interesting possibility would be to use `:=` only for re-assigning mutable values.

Table 4.1.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
			Assignment			
`=`	`lhs = rhs`	assign		`optional Object`	`optional Object`	`optional Object`
			Member reference			
`#`	`lhs#member`	curry		`Object`		`Callable<(P p, Q q,...) produces R>`, `Value<T>` or `OpenValue<T>`
			Member invocation			
`.`	`lhs.member`	invoke		`Object`		Member type
`^.`	`lhs^.member`	chain invoke		`X`		`X`
`?.`	`lhs?.member`	nullsafe invoke	`if (lhs exists) 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 (lhs exists) lhs.equals(rhs) else if`	`optional Object`	`optional Object`	`Boolean`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
			`(rhs exists) false else true`			
`!=`	`lhs != rhs`	not equal	`if (lhs exists) lhs.equals(rhs).negation else if (rhs exists) 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 disjunction	`lhs.xor(rhs)`	`Boolean`	`Boolean`	`Boolean`
`||`	`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)						
`exists`	`lhs exists`	exists	`if (lhs exists) true else false`	`optional Object`		`Boolean`
`?`	`lhs ? rhs`	default	`if (lhs exists) lhs else rhs`	`optional T`	`T`	`T`
Default assignment						

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
`?=`	`lhs ?= rhs`	default assignment	`if (lhs exists) lhs else lhs=rhs`	`optional T`	`T`	`T`
`nonempty`	`lhs nonempty`	nonempty	`if (lhs exists) lhs.empty.negation else false`	`optional Container`		`Boolean`
Containment						
`in`	`lhs in rhs`	in	`lhs.in(rhs)`	`X`	`Category<X>` or `Iterable<X>`	`Boolean`
Concatenation						
`+`	`lhs + rhs`	join	`lhs.join(rhs)`	`List<X>`	`List<X>`	`List<X>`
Keyed element access						
`[]`	`lhs[index]`	lookup	`lhs.value(index)`	`Correspondence<X,Y>`	`X`	`Y`
`[]`	`lhs[indices]`	lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`List<X>`	`List<Y>`
`[]`	`lhs[indices]`	lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`Set<X>`	`Set<Y>`
`[,,]`	`lhs[x, y, z]`	enumerated range	`Enumeration( lhs.lookup(x), lhs.lookup(y), lhs.lookup(z) )`	`List<X>`	`Integer`	`Enumeration<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 (rhs exists) 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						

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
`-`	`-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`

4.4.2. 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	Left
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:	`<=>`, `<`, `>`, `<=`, `>=`, `in`	Binary	None
Equality:	`==`, `!=`, `===`	Binary	None
Negation:	`!`	Unary prefix	Right
Conjunction:	`&&`, `&`	Binary	Left
Disjunction:	`||`, `|`, `^`	Binary	Left

Operations	Operators	Type	Associativity
Implication:	`=>`	Binary	None
Assignment, numeric assignment, logical assignment, default assignment:	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `&&=`, `||=`, `?=`	Binary	Right

4.5. Functor expressions

A functor expression has this form:

```
FunctorExpression := FunctorLocals OpenBlock
```

```
FunctorLocals := "(" ( Variable | VariablePair (COMMA Variable | VariablePair)* ) ")"
```

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

For example:

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

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

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

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

Consider the following functor expression:

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

Or:

```
(String x, String y) produces Comparison 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");
```

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 = 0;
List list = list.select(Integer x)
    return x > min && x < max;
```

Is equivalent to this Java code:

```
Integer min = 0;
Integer max = 10;
List list = list.select( new F1<Integer,Boolean>(min, max) {
        final Integer min = (Integer) locals[0];
        final Integer max = (Integer) locals[1];
        @Override Boolean call(Integer x) {
            return x > min && x < max;
        }
    } );
```

4.5.1. Syntax extension for functor parameters

Furthermore, for parameters of a method which are functor types, Ceylon provides a special method invocation protocol:

```
MemberReference OrderedParamValues FunctorParamValues
```

After the list of ordered parameters, a named list of functor parameters may be specified:

```
FunctorParamValues := SimpleFunctorExpression? (Identifier 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 := FunctorLocals? OpenBlock
```

For example:

```
String label = (x>10).if() isTrue "big" isFalse "little";
```

```
amounts.sort() by (Float x, Float y) { produces x<=>y }
```

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

```
people.select() only (Person p) { produces p.age>18 }
    .collect() each (Person p) { produces p.name };
```

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

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

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

The name of the first parameter may be omitted, if desired.

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

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

4.5.1.1. Syntax extension for iterations

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 ")" FunctorParamValues
```

And each functor expression for a parameter annotated `coordinated` of type `(X x) produces Y` or `(X x) void` 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 (X x) produces Y select,
                  coordinated (X x) produces Boolean if);
```

We may invoke the method as follows:

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

Which is equivalent to:

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

Or, we may invoke the method as follows:

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

Which is equivalent to:

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

4.5.1.2. Syntax extension for variable definition

A specialized invocation syntax is also provided for methods which define a variable. 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 Initializer ")" FunctorParamValues
```

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

For example, for the following method declaration:

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

We may invoke the method as follows:

```
Exact amount = ifExists(Payment p = order.payment) 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 (X x) produces Y seek)
  where X >= Usable;
```

We may invoke the method as follows:

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

Which is equivalent to:

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

4.5.1.3. Syntax extension for lists of cases

A specialized invocation syntax is also provided for methods which define a list of cases. If the method has a parameter of type `Enumeration<(X x) produces Y>` annotated `cases`, the parameter value may be specified according to:

```
(Identifier FunctorExpression)+
```

Or, if the method has a parameter of type `Enumeration<Entry<Case<X>, () produces Y>>` annotated `cases`, the parameter value may be specified according to:

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

For example, for the following method declaration:

```
public static
X attempt<X>(() produces X seek,
             cases (E e) produces X where E>=Exception... except);
```

We may invoke the method as follows:

```
attempt() {
    produces doSomething();
}
except (SomethingWrong sw) {
    log.info(sw);
    produces -1;
}
except (SomethingElseWrong sew) {
    log.warn(sew);
    produces -2;
}
```

And for the following method declaration:

```
public static
void select<X>(X selector),
               cases Entry<Case<X>, () produces X>... value);
```

We may invoke the method as follows:

```
select(payment.type)
value (CHECK) {
    paybyCheck(payment);
}
value (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), 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 performs
         identity comparison."
    public Boolean equals(Object that) {
        return this===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 assumes
         identity equality.
    public Integer hash {
        return identityHash(this);
    }

    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."
    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 for any ( Object elem in iterable )
            elem == this;
    }

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

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

    doc "Return a reference to the instance, of the given
         Type, or throw an exception if the instance is not
         of the given type."
    public T as<T>(Type<T> type) { ... }

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

    ...

}
```

5.2. Types

```
public interface Type<X>
        satisfies Case<Object> {
    ...
}
```

5.3. Boolean values

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

```
public final class 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
        };
    }

}
```

5.4. Values and callables

The interface `Callable` represents a callable reference.

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

The interface `Value` models a readable value.

```
public interface Value<T> {
    public () produces T get;
    public onGet( (T t) produces T interceptor );
}
```

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

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

5.5. 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<X>
        satisfies Annotated {
    public Type<X> declaringType;
    public String name;
    public Visibility visibility;
    public Boolean abstract;
    public Boolean override;
```

```
}
```

```
public interface Method<X,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<X,T>
        satisfies Member<X> {
    public Type<T> attributeType;
    public boolean mutable;
    public Value<T> curry(X instance);
    public onGet( (X x,T t) produces T interceptor );
}
```

```
public interface MutableAttribute<X,T>
        satisfies Attribute<X,T> {
    public override OpenValue<T> curry(X instance);
    public onSet( (X x,T t) produces T interceptor );
}
```

5.6. 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<X> {

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

}
```

```
public interface Iterator<X> {

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

}
```

The following decorator exists:

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();

}
```

5.7. 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<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> { ... }
```

5.8. 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);

}
```

5.9. Category, Correspondence and Container

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

```
public interface Category<X> {

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

}
```

```
public decorator Categories<X> (Category<X> category) {

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

}
```

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

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

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

}
```

```
public decorator OpenCategories<X> (OpenCategory<X> category) {

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

}
```

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<U, 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 decorator Correspondences<X> (Correspondence<X> 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 for (U key in keys)
            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 ( for (U key in keys) correspondence.lookup(key) ).elements;
    }

    doc "Determine if there are values associated with all the given keys."
    public Boolean defines(Collection<U> keys) {
        return for (U key in keys)
                if ( !correspondence.defines(key) ) found false
            fail 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<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 decorator OpenCorrespondences<X> (OpenCorrespondence<X> correspondence) {

     doc "Assign the given values to the given keys."
     public void define(Iterable<Entry<U, V>> definitions);

     doc "Add the given entry."
     public void define(U key -> V 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."
    Boolean empty;

}
```

5.10. Entries

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

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

```
public class Entry<U,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);
    }

}
```

5.11. Collections

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

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

    public Integer size;

    public Integer count(X element);

    public Set<X> elements;

    public OpenCollection<X> copy();

}
```

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(Collection<X> elements);
    public Integer retain(Collection<X> elements);

}
```

5.11.1. Sets

Sets implement the following interface:

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

    public Set<X> union(Set<X> set);
    public Set<X> intersection(Set<X> set);
    public Set<X> complement(Set<X> set);

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

    public OpenSet<X> copy();

}
```

There is a mutable subtype:

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

5.11.2. 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<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(X element);
    public Integer lastIndex(X element);

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

    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<X> join(List<X> elements);

    public List<X> sublist(Integer from, Integer to);

    public List<X> reversed();

    public Map<Integer,X> map;

    public OpenList<X> copy();

}
```

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 key, X element);
    public void truncate(Integer from, Integer to);

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

    public void reverse();
    public void sort((X x, X y) produces Comparison compare);

    public OpenList<X> sublist(Integer from, Integer to);

}
```

5.11.3. Maps

Maps implement the following interface:

```
public interface Map<U,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 Set<Entry<U,V>> entries;

    public OpenMap<X> copy();

}
```

There is a mutable subtype:

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

    public OpenSet<U> keys;
    public OpenBag<V> values;
```

```
    public OpenMap<V, Set<U>> inverse;

    public Integer removeKeys(Collection<U> keys);
    public Integer retainKeys(Collection<U> keys);

}
```

5.11.4. Bags

Bags implement the following interface:

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

    public OpenBag<X> copy();

    public Map<X,Integer> map;

}
```

There is a mutable subtype:

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

5.12. Ordered values

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

```
public interface Comparable<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;

}
```

5.13. 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<X>
        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<X>(X... values)
        satisfies List<X>, Case<X> {

    ...

    static Enumeration<Y> emptyEnumeration<Y>() {
        return Enumeration<X>();
    }

}
```

Enumerations may be freely converted to sets or maps:

```
public Map<U, V> converter EnumerationToMap(Enumeration<Entry<U,V>> enum) { ... }
```

```
public Set<X> converter EnumerationToSet(Enumeration<X> enum) { ... }
```

```
public OpenMap<U, V> converter EnumerationToOpenMap(Enumeration<Entry<U,V>> enum) { ... }
```

```
public OpenSet<X> converter EnumerationToOpenSet(Enumeration<X> enum) { ... }
```

```
public OpenList<X> converter EnumerationToOpenList(Enumeration<X> enum) { ... }
```

Furthermore, any object may be transparently promoted to an enumeration:

```
public Enumeration<T> converter ObjectToEnumeration(Object object) { return Enumeration(object); }
```

5.14. Characters, character strings and ranges

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 List<Character> characters = characters;

    ...

    public Iterable<String> tokens(Iterable<Character> separators=" ,;") { return ...; }
    public String join(String... strings);
    public String join(Iterable<String> strings);

}
```

5.15. Regular expressions

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

5.16. 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> { ... }
```

5.17. Instants, intervals and durations

```
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 {}
```

5.18. Control expressions

The `Util` class defines several methods for building complex expressons.

```
public class Util {

    doc "Evaluate the block. Useful for turning a series
         of statements into an expression."
    public static X evaluate<X>(() produces X do);

    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,
                                       () produces Y then);

    doc "If the condition is true, evaluate first block,
         otherwise, evaluate second block. Return result
         of evaluation."
    public static Y ifTrue<Y>(Boolean condition,
                              () produces Y then,
                              () produces Y 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 (X x) produces Y then);

    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 (X x) produces Y then,
                                  () produces Y otherwise);

    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 (X x) produces Boolean having,
                                  optional coordinated (X x) produces Comparable by = null);

    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 (X x) produces Y select,
                                    optional coordinated (X x) produces Comparable by = null);

    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 (X x) produces Boolean having,
                                    coordinated (X x) produces Y select,
                                    optional coordinated (X x) produces Comparable by = null);

    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 (X x) produces Boolean having);

    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 (X x) produces Boolean having
                             () produces X otherwise);

    doc "Count the elements for with the block evaluates to true."
    public static Integer count<X>(iterated Iterable<X> elements,
                                   coordinated (X x) produces Boolean having);

    doc "Return true iff for every element, the block evaluates to true."
    public static Boolean forAll<X>(iterated Iterable<X> elements,
                                    coordinated (X x) produces Boolean every);

    doc "Return true iff for some element, the block evaluates to true."
    public static Boolean forAny<X>(iterated Iterable<X> elements,
                                    coordinated (X x) produces Boolean exists);

    doc "Using the given resource, attempt to evaluate the first block.
```

```
         If an exception occurs, and there is a matching block, evaluate
         the exception block."
    public static Y using<X,Y>(specified X resource,
                               coordinated (X x) produces Y seek,
                               (E e) produces Y where E>=Exception... except)
                      where X >= Usable;

}
```

5.19. 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();
    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.