Report on the Experimental Language X10

Version 1.7

Vijay Saraswat
Nathaniel Nystrom

Please send comments to
Nathaniel Nystrom at nystrom@us.ibm.com

March 9, 2009

This report provides a description of the programming language X10. X10 is a single-inheritance class-based object-oriented (OO) programming language designed for high-performance, high-productivity computing on high-end computers supporting $\approx 10^5$ hardware threads and $\approx 10^{15}$ operations per second.

X10 is based on state-of-the-art object-oriented programming languages and deviates from them only as necessary to support its design goals. The language is intended to have a simple and clear semantics and be readily accessible to mainstream OO programmers. It is intended to support a wide variety of concurrent programming idioms.

The X10 design team consists of Ganesh Bikshandi, David Cunningham, Robert Fuhrer, David Grove, Sreedhar Kodali, Bruce Lucas, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Sayantan Sur, Olivier Tardieu, Pradeep Varma, and Krishna Nandivada Venkata.

Past members include David Bacon, Raj Barik, Bob Blainey, Philippe Charles, Perry Cheng, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Patrick Gallop, Christian Grothoff, Allan Kielstra, Sriram Krishnamoorthy, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon, Christoph von Praun, Jan Vitek, and Tong Wen.

For extended discussions and support we would like to thank: Gheorghe Almasi, Robert Blackmore, Robert Callahan, Calin Cascaval, Norman Cohen, Elmootaz Elnozahy, John Field, Kevin Gildea, Chulho Kim, Orren Krieger, Doug Lea, John McCalpin, Paul McKenney, Andrew Myers, Ram Rajamony, R.K. Shyamasundar, Filip Pizlo, V.T. Rajan, Frank Tip, Mandana Vaziri, and Hanhong Xue.

We thank Jonathan Rhees and William Clinger with help in obtaining the LaTeX style file and macros used in producing the Scheme report, on which this document is based. We acknowledge the influence of the Java™ Language Specification [5].

This document revises Version 1.5 of the Report, released in June 2007. It documents the language corresponding to Version 1.7 of the implementation, which is currently under development.

Earlier implementations benefited from significant contributions by Raj Barik, Philippe Charles, Christopher Donawa, Robert Fuhrer, Christian Grothoff, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Vivek Sarkar, Olivier Tardieu, Pradeep Varma, Krishna Nandivada Venkata, and Christoph von Praun.) Tong Wen has written many application programs in X10. Guojing Cong has helped in the development of many applications. The implementation of generics in X10 was influenced by the implementation of PolyJ [2] by Andrew Myers and Michael Clarkson.

Contents

1 Introduction

Background

Larger computational problems need more powerful computers capable of performing a larger number of operations per second. The era of increasing performance by simply increasing clocking frequency now seems to be behind us. It is becoming increasingly difficult to mange chip power and heat. Instead, computer designers are starting to look at *scale out* systems in which the system's computational capacity is increased by adding additional nodes of comparable power to existing nodes, and connecting nodes with a high-speed communication network.

A central problem with scale out systems is a definition of the *memory model*, that is, a model of the interaction between shared memory and simultaneous (read, write) operations on that memory by multiple processors. The traditional "one operation at a time, to completion" model that underlies Lamport's notion of *sequential consistency* (SC) proves too expensive to implement in hardware, at scale. Various models of *relaxed consistency* have proven too difficult for programmers to work with.

One response to this problem has been to move to a *fragmented memory model*. Multiple processors are made to interact via a relatively language-neutral message-passing format such as MPI [10]. This model has enjoyed some success: several high-performance applications have been written in this style. Unfortunately, this model leads to a *loss of programmer productivity*: the message-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code (in an SPMD fashion) etc.

One response to this problem has been the advent of the *partitioned global address space* (GAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [3, 11]. These languages permit the programmer to think of a single computation running across multiple processors, sharing a common address space. All data resides at some processors, which is said to have *affinity* to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global *barriers* are used to ensure that processors remain roughly in lock-step.

X10 is a modern object-oriented programming language in the GAS family. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end computers—for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads. X10 is based on state-of-the-art object-oriented programming ideas primarily to take advantage of their proven flexibility and ease-of-use for a wide spectrum of programming problems. X10 takes advantage of several years of research (e.g., in the context of the Java Grande forum, [7, 1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *value types* (such as `Int`, `Float`, `Complex` etc), supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [4]) and supports IEEE-standard floating point arithmetic. Some limited operator overloading is provided for a few "built in" classes in the `x10.lang` package. Future versions of the language will support user-definable operator overloading.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the GAS model to the *globally asynchronous, locally synchronous* (GALS) model originally developed in hardware and embedded software research. X10 introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more *activities*. Activities are extremely lightweight threads of execution. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. To access or update memory at other places, it must spawn activities asynchronously (either explicitly or implicitly). X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale. *Immutable* data needs no consistency management and may be freely copied by the implementation between places. One or more *clocks* may

be used to order activities running in multiple places. Arrays may be distributed across multiple places. Arrays support parallel collective operations. A novel exception flow model ensures that exceptions thrown by asynchronous activities can be caught at a suitable parent activity. The type system tracks which memory accesses are local. The programmer may introduce place casts which verify the access is local at run time. Linking with native code is supported.

X10 v1.7 adds to Version 1.1 the following features: The syntax of the language is more concise. The dependent type system is extended to support genericity and constraints on types. Type definitions and local type inference are supported. Arrays and futures are defined in the language library, rather than as type constructors. Closures and function types are supported.

X10 v1.7 is a transitional version. Previous versions of the language were based on Java. Future versions of the language will diverge more and more from the Java specification. However, for X10 v1.7, any omitted specification should default to the Java Language Specification, Second edition [5].

X10 is an experimental language. Several representative concurrent idioms have already found pleasant expression in X10. We intend to develop several full-scale applications to get better experience with the language, and revisit the design in the light of this experience.

2 Overview of X10

X10 is a statically typed object-oriented language, extending a sequential core language with *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *value* types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing.

2.1 Object-oriented features

The sequential core of X10 is a class-based object-oriented language similar to Java or Scala. Programmers write X10 code by writing *interfaces* (§8) and *classes* (§9).

An X10 class has fields, methods and inner types (interfaces, classes), subclasses another class, and implements one or more interfaces. Thus X10 classes live in a single-inheritance code hierarchy. An interface specifies a set of methods, constant fields, and inner types. Interfaces are multiply inherited.

Dependent types Classes and interfaces may declare *properties*: immutable object members bound at object construction. Types may be defined by constraining a class or interface's properties. *Value properties* enable the definition of *dependent types*.

For example, the following code declares a class for a two-dimensional `Point` class with an `add` method.

```
class Point(x: Int, y: Int) {
    def add(p: Point): Point { return new Point(x+p.x,y+p.y); }
}
```

The class has integer value properties `x` and `y`. The `add` method creates and returns a new point by element-wise addition. The dependent type `Point{x==0}` is the type of all points with `x` set to `0`; that is, all points along the y-axis.

Generic types Classes and interfaces may have type parameters, permitting the definition of *generic types*. For example, the following code declares a simple `List` class with a type parameter `T`.

```
class List[T] {
    var head: T;
    var tail: List[T];
    def this(h: T, t: List[T]) { head = h; tail = t; }
    def append(x: T) {
        if (this.tail == null)
            this.tail = new List(x, null);
        else
            this.tail.add(x);
    }
}
```

The constructor (`def this`) initializes the fields of the new object. The `add` method appends an element to the list. `List` is a generic type. When instances of `List` are allocated, the type parameter `T` must be bound to a concrete type. `List[Int]` is the type of lists of element type `Int`, `List[String]` is the type of lists of element type `String`.

Reference and value classes There are two kinds of classes: *reference* classes (§9.1) and *value* classes (§9.2). A reference class typically has updatable fields. Objects of such a class may not be freely copied from place to place. Methods may be invoked on such an object only by an activity in the same place. The `null` reference is a value of any reference type.

A value class (§9.2) has no updatable fields (defined directly or through inheritance), and allows no reference subclasses. (Fields may be typed at reference classes, so may contain references to objects with mutable state.) Objects of such a class may be freely copied from place to place, and may be implemented very efficiently. Methods may be invoked on such an object from any place.

X10 has no primitive classes. However, the standard library `x10.lang` supplies (final) value classes `Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`,

`Complex` and `String`. The user may defined additional arithmetic value classes using the facilities of the language.

2.2 The sequential core

Control flow. X10 supports standard sequential control flow constructs: `if` statements, `while` loops, `for` loops, `switch` statements, etc. X10 also supports exceptions: exceptions are raised by `throw` statements and are handled by `try`–`catch` statements.

Primitive operations. The language provides syntax for performing binary and unary operations on values of types defined in the standard library.

Closures. X10 provides closures (§12.5) to allow code to be used as values. The body of a closure may capture variables in the closure's environment. Closures are used implicitly in asyncs, futures, and array initializers. For example, the following method uses a closure to increment elements of an array.

```
def incr(A: Array[Int]): Array[Int] = {
    val f = (x: Int) => x+1; // e.g., f(1) == 2
    return A.lift(f);
}
```

Allocation. Objects are allocated with the `new` operator (§12.24), which takes a class name and type and value arguments to pass to the class's constructor. The constructor must ensure that all properties of the class and its superclasses are bound.

Coercions and conversions X10 supports implicit and explicit coercions and conversions (§4.10).

Values of one type can be converted to another type using the `as` operation:

```
val x: Int = 65535;
val y: Byte = x as Byte; // convert to Byte,
                         // retaining the lower 8 bits
```

The `as` operation does not necessarily preserve equality and for numeric values may result in a loss of precision.

References may be coerced to another type, preserving object identity. A run-time check is performed to ensure the reference is to an object of the target type. If not, a `ClassCastException` is thrown.

```
// C and D are immediate subclasses of B.
val x: B = new C();
val y: C = x as C; // run-time check succeeds
val z: D = x as D; // run-time check fails
```

2.3 Places and activities

An X10 program is intended to run on a wide range of computers, from uniprocessors to large clusters of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of *place* (§13). Conceptually, a place is a "virtual shared-memory multi-processor": a computational unit with a finite (though perhaps changing) number of hardware threads and a bounded amount of shared memory, uniformly accessible by all threads.

An X10 computation acts on *data objects*(§6) through the execution of lightweight threads called *activities*(§14). Objects are of two kinds. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place and stays at that place throughout its lifetime. An *aggregate* object has many fields (the number may be known only when the object is created), uniformly accessed through an index (e.g., an integer) and may be distributed across many places. The distribution of an aggregate object remains unchanged throughout the computation. X10 assumes an underlying garbage collector will dispose of (scalar and aggregate) objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation. (There are no operations in the language to allow a programmer to explicitly release memory.)

X10 has a *unified* or *global address space*. This means that an activity can reference objects at other places. However, an activity may synchronously access data items only in the current place (the place in which the activity is running). It may atomically update one or more data items, but only in the current place. To read a remote location, an activity must spawn another activity *asynchronously*

(§14.2). This operation returns immediately, leaving the spawning activity with a *future* (§14.8) for the result. Similarly, remote location can be written into only by asynchronously spawning an activity to run at that location.

Throughout its lifetime an activity executes at the same place. An activity may dynamically spawn activities in the current or remote places.

Place casts. The programmer may use the standard type cast mechanism (§12.25) to cast a value to a located type. A `BadPlaceException` is thrown if the value is not of the given type. This is the only language construct that throws a `BadPlaceException`.

Atomic blocks. X10 introduces statements of the form `atomic S` where `S` is a statement. The type system ensures that such a statement will dynamically access only local data. (The statement may throw a `BadPlaceException`—but only because of a failed place cast.) Such a statement is executed by the activity as if in a single step during which all other activities are frozen.

Asynchronous activities. An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`.

An asynchronous expression of type `Future[T]` has the form `future (p) e` where `e` is an expression of type `T`. The expression `e` may reference final and shared variables declared in the lexically enclosing environment. It executes the expression `e` at the place `p` as an asynchronous activity, immediately returning with a future. The future may later be forced causing the activity to be blocked until the return value has been computed by the asynchronous activity.

2.4 Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of (possibly diverse) activities in an X10 computation. Instead, it becomes necessary to allow a computation to use multiple barriers. X10 *clocks* (§15) are designed to offer the functionality of multiple barriers in a dynamic context while still supporting determinate, deadlock-free parallel computation.

Activities may use clocks to repeatedly detect quiescence of arbitrary programmer-specified, data-dependent set of activities. Each activity is spawned with a known set of clocks and may dynamically create new clocks. At any given time an activity is *registered* with zero or more clocks. It may register newly created activities with a clock, un-register itself with a clock, suspend on a clock or require that a statement (possibly involving execution of new async activities) be executed to completion before the clock can advance. At any given step of the execution a clock is in a given phase. It advances to the next phase only when all its registered activities have *quiesced* (by executing a `next` operation on the clock). When a clock advances, all its activities may now resume execution.

Thus clocks act as *barriers* for a dynamically varying collection of activities. They generalize the barriers found in MPI style program in that an activity may use multiple clocks simultaneously. Yet programs using clocks are guaranteed not to suffer from deadlock.

In future versions of the language, clocks will be integrated into the X10 type system, permitting variables to be declared so that they are final in each phase of a clock.

2.5 Arrays, regions and distributions

An X10 array type is a map from a *distribution* (§16.3) to a type, which may itself be an array type.

A distribution is a map from a *region* (§16.2) to places. A region is a collection of *points* or *indices*. For instance, the region `[0..200,1..100]` specifies a collection of two-dimensional points `[i,j]` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. Points are used in array index expressions to pick out a particular array element.

Operations are provided to construct regions from other regions, and to iterate over regions. Standard set operations, such as union, disjunction and set difference are available for regions.

A primitive set of distributions is provided, together with operations on distributions. A *sub-distribution* of a distribution is one defined on a smaller region and agrees with the distribution at all points. The standard operations on regions are extended to distributions.

A new array can be created by restricting an existing array to a sub-distribution, by combining multiple arrays, and by performing pointwise operations on arrays with the same distribution.

X10 allows array constructors to iterate over the underlying distribution and specify a value at each item in the underlying region. Such a constructor may spawn activities at multiple places.

In future versions of the language, a programmer may specify new distributions, and new operations on distributions.

2.6 Annotations

X10 supports annotations on classes and interfaces, methods and constructors, variables, types, expressions and statements. These annotations may be processed by compiler plugins.

2.7 Translating MPI programs to X10

While X10 permits considerably greater flexibility in writing distributed programs and data structures than MPI, it is instructive to examine how to translate MPI programs to X10.

Each separate MPI process can be translated into an X10 place. Async activities may be used to read and write variables located at different processes. A single clock may be used for barrier synchronization between multiple MPI processes. X10 collective operations may be used to implement MPI collective operations. X10 is more general than MPI in (a) not requiring synchronization between two processes in order to enable one to read and write the other's values, (b) permitting the use of high-level atomic blocks within a process to obtain mutual exclusion between multiple activities running in the same node (c) permitting the use of multiple clocks to combine the expression of different physics (e.g., computations modeling blood coagulation together with computations involving the flow of blood), (d) not requiring an SPMD style of computation.

2.8 Summary and future work

2.8.1 Design for scalability

X10 is designed for scalability. An activity may atomically access only multiple locations in the current place. Unconditional atomic blocks are statically guaranteed to be non-blocking, and may be implemented using non-blocking techniques that avoid mutual exclusion bottlenecks. Data-flow synchronization permits point-to-point coordination between reader/writer activities, obviating the need for barrier-based or lock-based synchronization in many cases.

2.8.2 Design for productivity

X10 is designed for productivity.

Safety and correctness. Programs written in X10 are guaranteed to be statically *type safe*, *memory safe* and *pointer safe*. Static type safety guarantees that at run time a location contains only those values who's dynamic type satisfies the constraints imposed by the location's static type and every run-time operation performed on the value in a location is permitted by the static type of the location.

Memory safety guarantees that an object may only access memory within its representation, and other objects it has a reference to. X10 supports no pointer arithmetic, and bound-checks array accesses dynamically if necessary. X10 uses dynamic garbage collection to collect objects no longer referenced by the computation. X10 guarantees that no object can retain a reference to an object whose memory has been reclaimed. Further, X10 guarantees that every location is initialized at run time before it is read, and every value read from a location has previously been written into that location.

Because places are reflected in the type system, static type safety also implies *place safety*: a location may contain references to only those objects whose location satisfies the restrictions of the static place type of the location.

X10 programs that use only clocks and unconditional atomic blocks are guaranteed not to deadlock. Unconditional atomic blocks are non-blocking, hence cannot introduce deadlocks (assuming the implementation is correct).

Many concurrent programs can be shown to be determinate (hence race-free) statically.

Integration. A key issue for any new programming language is how well it can be integrated with existing (external) languages, system environments, libraries and tools.

We believe that X10, like Java, will be able to support a large number of libraries and tools. An area where we expect future versions of X10 to improve on Java like languages is *native integration* (§18). Specifically, X10 will permit permit multi-dimensional local arrays to be operated on natively by native code.

2.8.3 Conclusion

X10 is considerably higher-level than thread-based languages in that it supports dynamically spawning very lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection.

Yet it is much more concrete than languages like HPF in that it forces the programmer to explicitly deal with distribution of data objects. In this the language reflects the designers' belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent.[1] At the same time we believe that the place-based type system and support for generic programming will allow the X10 programmer to be highly productive; many of the tedious details of distribution-specific code can be handled in a generic fashion.

We expect the next version of the language to be significantly informed by experience in implementing and using the language. We expect it to have constructs to support continuous program optimization, and allow the programmer to provide guidance on clustering places to (hardware) nodes. For instance, we may introduce a notion of hierarchical clustering of places.

[1] In this X10 is similar to more modern languages such as ZPL [4].

3 Lexical structure

In general, X10 follows Java rules [5, Chapter 3] for lexical structure.

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators.

Whitespace ASCII space, horizontal tab (HT), form feed (FF) and line terminators constitute white space.

Comments All text included within the ASCII characters "`/*`" and "`*/`" is considered a comment and ignored; nested comments are not allowed. All text from the ASCII characters "`//`" to the end of line is considered a comment and is ignored.

Identifiers Identifiers are defined as in Java. Identifiers consist of a single letter followed by zero or more letters or digits. Letters are defined as the characters for which the Java method `Character.isJavaIdentifierStart` returns true. Digits are defined as the ASCII characters `0` through `9`.

Keywords X10 reserves the following keywords:

`abstract`	`any`	`as`	`async`
`at`	`ateach`	`atomic`	`await`
`break`	`case`	`catch`	`class`
`clocked`	`const`	`continue`	`current`
`def`	`default`	`do`	`else`
`extends`	`extern`	`final`	`finally`
`finish`	`for`	`foreach`	`future`

```
goto          has           here          if
implements    import        instanceof    interface
local         native        new           next
nonblocking   or            package       private
protected     property      public        return
safe          self          shared        static
super         switch        this          throw
throws        to            try           type
val           value         var           when
while
```

Note that the primitive types are not considered keywords. The keyword `goto` is reserved, but not used.

Literals Literals are either integers, floating point numbers, booleans, characters, strings, and `null`. X10 v1.7 defines literal syntax in the same way as Java does.

Separators X10 has the following separators and delimiters:

```
( )  { }  [ ]  ;  ,  .
```

Operators X10 has the following operators:

```
==  !=  <   >   <=  >=
&&  ||  &   |   ^
<<  >>  >>>
+   -   *   /   %
++  --  !   ~
&=  |=  ^=
<<= >>= >>>
+=  -=  *=  /=  %=
=   ?   :   =>  ->
<:  :>  @   ..
```

4 Types

X10 is a *strongly typed* object-oriented language: every variable and expression has a type that is known at compile-time. Further, X10 has a *unified* type system: all data items created at run time are *objects* (§6). Types limit the values that variables can hold and specify the places at which these values lie.

X10 supports two kinds of objects, *reference objects* and *value objects*. Reference objects are instances of *reference classes* (§9.1). They may contain mutable fields and must stay resident in the place in which they were created. Value objects are instances of *value classes* (§9.2). They are immutable and may be freely copied from place to place. Either reference or value objects may be *scalar* (instances of a non-array class) or *aggregate* (instances of arrays). Only variables of reference types may be assigned `null`.

Types are used in variable declarations, explicit coercions and conversions, object creation, array creation, class literals and `instanceof` expressions.[1]

Types in X10 are specified through declarations and through type constructors, described in the remainder of the chapter:

- A class declaration defines a *class type* (§4.1), which may be either a reference class or a value class.
- An interface declaration defines an *interface type* (§4.1.2).
- The *boxed type* constructor produces a reference type from a value type (§**??**).

[1] In order to allow this version of the language to focus on the core new ideas, X10 v1.7 does not have user-definable class loaders, though there is no technical reason why they could not have been added.

- Classes and interface have *type parameters*. A class or interface with one or more type parameters is a *generic class* or *generic interface* (§4.2.1).
- New type constructors may be defined with *type definitions* (§4.3).
- Methods, constructors, closures, and type definitions may have *type parameters*, which are instantiated with concrete types at invocation (§4.2).
- *Function type* constructors are used to define function types; closures and method selectors have function type (§4.6).
- A *constrained type* constrains the properties of a base type (§4.5).
- Types may be marked with user-defined annotations. *Annotated types* (§4.7) may be processed by compiler plugins.

Type	::=	*FunctionType*
	|	*ConstrainedType*
FunctionType	::=	*TypeParameters*? `(` *Formals*? `)` *Constraint*? *Throws*? `=>` *Type*
TypeParameters	::=	`[` *TypeParameter* (`,` *TypeParameter*)* `]`
TypeParameter	::=	*Identifier*
Throws	::=	`throws` (*TypeName* `,` *TypeName*)*
ConstrainedType	::=	*Annotation** *BaseType Constraint*? *PlaceConstraint*?
BaseType	::=	*ClassBaseType*
	|	*InterfaceBaseType*
	|	*PathType*
	|	`(` *Type* `)`
ClassType	::=	*Annotation** *ClassBaseType Constraint*? *PlaceConstraint*?
InterfaceType	::=	*Annotation** *InterfaceBaseType Constraint*? *PlaceConstraint*?
PathType	::=	*Expression* `.` *Identifier*
Annotation	::=	`@` *InterfaceBaseType Constraint*?
ClassOrInterfaceType	::=	*ClassType*
	|	*InterfaceType*
ClassBaseType	::=	*TypeName*
InterfaceBaseType	::=	*TypeName*

4.1 Classes and interfaces

4.1.1 Class types

A *class declaration* (§9) introduces a *class type* containing all instances of the class.

Class instances are created via constructor calls. Class instances have fields and methods, type members, and value properties bound at construction time. In addition, classes have static members: constant fields, type definitions, and member classes and member interfaces.

A class with type parameters is *generic*. A class type is legal only if all of its parameters are instantiated on concrete types.

X10 does not permit mutable static state, so the role of static methods and initializers is quite limited. Instead programmers should use singleton classes to carry mutable static state.

Classes are structured in a single-inheritance hierarchy. All reference classes extend the class `x10.lang.Ref`. All value classes extend the class `x10.lang.Value`. Classes are declared to extend a single superclass (except for `Ref` and `Value`, which extend no other class), All classes implement the interface `x10.lang.Object` and zero or more other interfaces.

`Object` has the following signature:

```
package x10.lang;

public interface Object {
    public def toString(): String;
    public def equals(Object): Boolean;
    public def hashCode(): Int;
    public def className(): String;
}
```

Classes may be declared to be either reference classes, or value classes. A *value class type* is a class type in which all fields of the class are final. Variables of reference class type may be `null`; variables of value class type may not be `null`. All reference classes extend the `x10.lang.Ref` class; all value classes extend the `x10.lang.Value` class.

It is a static error if a value class extends `Ref` (or any subclass of `Ref`). It is a static error if a reference class extends `Value` (or any subclass of `Value`).

`Ref` and `Value` have the following signatures:

```
package x10.lang;

public class Ref(location: Place) {
    public Ref() { property(here); }
}

public class Value { }
```

4.1.2 Interface types

An *interface declaration* (§8) defines an *interface type*, which specifies a set of methods, type members, and properties to be implemented by any class declared to implement the interface. Interfaces also have static members: constant fields, type definitions, and member classes and interfaces.

An interface may extend multiple interfaces. All interfaces extend `x10.lang.Object`.

Classes may be declared to implement multiple interfaces. Semantically, the interface type is the set of all objects that are instances of (value or reference) classes that implement the interface. A class implements an interface if it is declared to and if it implements all the methods and properties defined in the interface.

4.1.3 Value properties

Classes and interfaces may have *value properties*, public final fields bound to object creation. For example, the following code declares a class named `Point` with properties `x` and `y` and a `move` method. The properties are bound using the `property` statement in the constructor.

```
class Point(x: Int, y: Int) {
  def this(x: Int, y: Int) { property(x, y); }
  def move(dx: Int, dy: Int) = new Point(x+dx, y+dy);
}
```

The value properties of a class or interface may be constrained with a boolean expression. The type `Point{x==0}` is the set of all points whose `x` property is `0`.

4.2 Type parameters

A class, interface, method, or closure may have type parameters whose scope is the signature and body of the declaring class, interface, method, or closure. Similarly, a type definition may have type parameters that scope over the body of the type definition.

Type parameters may be constrained by a *guard* on the declaration (§9, §4.3, §9.7,§12.5). The type parameters of classes and interfaces must be bound to concrete types (possibly to a type parameter) for the type to be legal; thus `List[int]` and `List[C]` are legal types, but `List` alone is not. The type parameters of methods and closures must be bound to concrete types at invocation. Parametrized type definitions specify new type constructors; the type parameters of a type definition must be bound to yield a type.

4.2.1 Generic types

A *generic class* is a class declared with one or more type parameters. Generic classes can be instantiated by instantiating the type parameters of the base type.

Consider the following declaration of a `Cell` class.

```
class Cell[X] {
    var x: X;
    def this(x: X) { this.x = x; }
    def get(): X = x;
    def set(x: X) = { this.x = x: }
}
```

This declares a class `Cell` with a type parameter `X`. `Cell` may be used as a type by instantiating `X`.

`Cell[int]` is the type of all `Cell` containing an `int`. The `get` method returns an `int`; the `set` method takes an `int` as argument. Note that `Cell` alone is not a legal type because the parameter is not bound.

Parameters may be declares as invariant, covariant, or contravariant. The `X` parameter of `Cell` above is invariant. Consider the following classes:

```
class Get[+X] {
    var x: X;
```

```
    def this(x: X) { this.x = x; }
    def get(): X = x;
}

class Set[-X] {
    var x: X;
    def this(x: X) { this.x = x; }
    def set(x: X) = { this.x = x: }
}
```

The `X` parameter of the `Get` class is covariant; the `X` parameter of the `Set` class is contravariant.

Given types `S` and `T`.

- If the parameter of `Get` is covariant, then `Get[S]` is a subtype of `Get[T]` if `S` is a *subtype* of `T`.
- If the parameter of `Set` is covariant, then `Set[S]` is a subtype of `Set[T]` if `S` is a *supertype* of `T`.
- If the parameter of `Cell` is invariant, then `Cell[S]` is a subtype of `Cell[T]` if `S` is a *equal* to `T`.

4.3 Type definitions

Types may be defined through *type definitions*. A type definition is a type alias or a type-valued function that maps value and type parameters to another type. A type definition may appear as a static or instance class or interface member or in a statement block.

Type definitions may have zero or more type parameters and value parameters. The general form of a type definition is:

```
type X[Y_1, ..., Y_m](x_1: T_1, ... x_n: T_n){c} = U;
```

$\texttt{Y}_1, \ldots, \texttt{Y}_m$ are the type parameters of `X`; $\texttt{x}_1, \ldots, \texttt{x}_n$ are the value parameters of `X` with types $\texttt{T}_1, \ldots, \texttt{T}_n$. The constraint `c` is called the *guard* of `X`. Finally, `U` is the defining type of `X`.

The expresion `X[`S_1`, ..., `S_m`](`e_1`, ..., `e_n`)` denotes a type whenever S_1, ..., S_m are type arguments and e_1, ..., e_n are value arguments with types T_1, ..., T_n such that the guard `c` holds. The type denoted is the defining type `U` with the appropriate substitutions applied: `U[`S_1`/`Y_1`,...,`S_m`/`Y_m`,`e_1`/x`$_1$`,...,`e_n`/x`$_n$`)`.

If the type definition is a static member of a class or interface `C`, then the type definition defines the type constructor `C.X` and `C.X[`S_1`, ..., `S_m`](`e_1`, ..., `e_n`)` is a type.

If the type definition is an instance member of a class or interface `C`, then, for a final access path expression `e` of type `C`, the type definition defines the type constructor `e.X` and `e.X[`S_1`, ..., `S_m`](`e_1`, ..., `e_n`)` is a type.

If the type definition appears in a statement block, then it defines the type constructor `X`, visible in the remainder of the block, and `X[`S_1`, ..., `S_m`](`e_1`, ..., `e_n`)` is a type.

4.4 Path types and path type constructors

Instance member type definitions declare *path types*. If the type definition is parametrized, the definition specifies a type constructor; instantiating the parameters yields a type. A path type (constructor) is of the form `p.X`, and consists of a final access path `p`, and a type member `X`. The final access path `p` must be either `this`, `self`, a final local variable, or a final field or value property whose target is itself a final access path. The path must type-check as a normal expression in the scope in which it appears. The type member or property `X` must be an instance member of the type of the path `p`.

The name resolution rules for `X` are identical to the name resolution rules for fields and local variables. If an identifier `X` resolves to a type member or property of an enclosing class or interface `T`, then `X` is equivalent to the path type `T.this.X`.

4.4.1 Final access paths

To ensure soundness of the type system, the expression `e` used in a path type `e.X` must be a *final access path*, either a final local variable or formal parameter (including the special variables `this` and `self`), or a final field or value property of a final access path.

4.5 Constrained types

Given a type `T`, a *constrained type* `T{e}` may be constructed by constraining its properties with a boolean expression `e`.

`T{e}` is a *dependent type*, that is, a type dependent on values. The type `T` is called the *base type* and `e` is called the *constraint*. For reference types, the constraint may specify the places at which the object resides.

For brevity, the constraint may be omitted and interpreted as `true`.

Constraints on properties induce a natural subtyping relationship: `C{c}` is a subtype of `D{d}` if `C` is a subclass of `D` and `c` entails `d`.

Type parameters cannot be constrained.

4.5.1 Constraints

Expressions used as constraints are restricted by the constraint system in use to ensure that the constraints can be solved at compile time. The X10 compiler allows compiler plugins to be installed to extend the constraint language and the constraint system. Constraints must be of type `Boolean`. The compiler supports the following constraint syntax.

Constraint	::=	*ValueArguments Guard*$^?$
	|	*ValueArguments*$^?$ *Guard*
ValueArguments	::=	`(` *ArgumentList*$^?$ `)`
ArgumentList	::=	*Expression* (`,` *Expression*)*
Guard	::=	`{` *DepExpression* `}`
DepExpression	::=	(*Formal* `;`)* *ArgumentList*

The default X10 constraint system supports equality expressions (==), conjunction (&&), and subtyping and supertyping (<: and :>) expressions over constants, final access paths, and types, and existential quantification over typed variables.

Subsequent implementations are intended to support boolean algebra, arithmetic, relational algebra, etc., to permit types over regions and distributions. We envision this as a major step towards removing most, if not all, dynamic array bounds and place checks from X10.

4.5.2 Place constraints

Recall that an X10 computation spans multiple places (§13). Each place constains data and activities that operate on that data. X10 v1.7 does not permit the dynamic creation of a place. Each X10 computation is initiated with a fixed number of places, as determined by a configuration parameter. In this section we discuss how the programmer may supply place type information, thereby allowing the compiler to check data locality, i.e., that data items being accessed in an atomic section are local.

PlaceConstraint ::= `!` *Place*$^{?}$
Place ::= `current`
| *Expression*

Because of the importance of places in the X10 design, special syntactic support is provided for constrained types involving places.

All X10 reference classes extend the class `x10.lang.Ref`, which defines a property `location` of type `Place`.

```
package x10.lang;
public class Ref(location: place) { ... }
```

If a constrained reference type `T` has an `!p` suffix, the constraint for `T` is implicitly assumed to contain the clause `self.location==p`; that is, `C{c}!p` is equivalent to `C{self.location==p && c}`.

If the place `p` is ommitted, `here` is assumed; that is, `C{c}!` is equivalent to `C{self.location==here && c}`.

The place specifier `current` on an array base type specifies that an object with that type at point `p` in the array is located at `dist(p)`. The `current` specifier can be used only with array types.

STATIC SEMANTICS RULE: It is a compile time error for the `!`-annotation to be used for value types.

4.5.3 Constraint semantics

STATIC SEMANTICS RULE (Variable occurrence): In a dependent type `T = C{c}`, the only variables that may occur in `c` are (a) `self`, (b) properties visible at `T`,

(c) final local variables, final method parameters or final constructor parameters visible at `T`, (d) final fields visible at `T`'s lexical place in the source program.

STATIC SEMANTICS RULE (Restrictions on `this`): The special variable `this` may be used in a dependent clause for a type `T` only if (a) `T` occurs in a property declaration for a class, (b) `T` occurs in an instance method, (c) `T` occurs in an instance field, (d) `T` occurs in an instance initializer.

In particular, `this` may not be used in types that occur in a static context, or in the arguments, body or return type of a constructor or in the extends or implements clauses of class and interface definitions. In these contexts, the object that `this` would correspond to is not defined.

STATIC SEMANTICS RULE (Variable visibility): If a type `T` occurs in a field, method or constructor declaration, then all variables used in `T` must have at least the same visibility as the declaration. The relation "at least the same visibility as" is given by the transitive closure of:

```
public > protected > package > private
```

All inherited properties of a type `T` are visible in the property list of `T`, and the body of `T`.

In general, variables (i.e., local variables, parameters, properties, fields) are visible at `T` if they are defined before `T` in the program. This rule applies to types in property lists as well as parameter lists (for methods and constructors). A formal parameter is visible in the types of all other formal parameters of the same method, constructor, or type definition, as well as in the method or constructor body itself. Properties are accessible via their containing object–`this` within the body of their class declaration. The special variable `this` is in scope at each property declaration, constructor signatures and bodies, instance method signatures and bodies, and instance field signatures and initializers, but not in scope at `static` method or field declarations or `static` initializers.

We permit variable declarations `v: T` where `T` is obtained from a dependent type `C{c}` by replacing one or more occurrences of `self` in `c` by `v`. (If such a declaration `v: T` is type-correct, it must be the case that the variable `v` is not visible at the type `T`. Hence we can always recover the underlying dependent type `C{c}` by replacing all occurrences of `v` in the constraint of `T` by `self`.)

For instance, `v: Int{v > 0}` is shorthand for `v: Int{self > 0}`.

STATIC SEMANTICS RULE (Constraint type): The type of a constraint `c` must be `Boolean`.

A variable occurring in the constraint `c` of a dependent type, other than `self` or a property of `self`, is said to be a *parameter* of `c`.

An instance `o` of `C` is said to be of type `C{c}` (or: *belong to* `C{c}`) if the predicate `c` evaluates to `true` in the current lexical environment, augmented with the binding `self` $\mapsto$ `o`. We shall use the function $[\![$`C{c}`$]\!]$ to denote the set of objects that belong to `C{c}`.

4.5.4 Type invariants

A class or interface declaration may specify a *guard*, an invariant on all instances of that type. All the properties of the type, together with inherited properties, may appear in the class invariant. The class invariant may also constrain the class's type parameters. A guard `c` with value property list $\texttt{x}_1\texttt{: T}_1\texttt{, } \ldots\texttt{, x}_n\texttt{: T}_n$ for a class `C` is said to be consistent if each of the $\texttt{T}_i$ are consistent and the constraint

$$\exists \texttt{x}_1\texttt{: T}_1\texttt{, } \ldots\texttt{, x}_n\texttt{: T}_n\texttt{, self: C. c}$$

is valid (always true).

With every defined class or interface `T` we associate a *type invariant* $inv(\texttt{T})$ as follows. The type invariant associated with `x10.lang.Object` is `true`.

The type invariant associated with any interface `I` that extends interfaces $\texttt{I}_1\texttt{, } \ldots\texttt{, I}_k$ and defines properties $\texttt{x}_1\texttt{: P}_1\texttt{, } \ldots\texttt{, x}_n\texttt{: P}_n$ and specifies a guard `c` is given by:

$$inv(\texttt{I}_1)\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ inv(\texttt{I}_k)\ \texttt{\&\& self.x}_1\texttt{: P}_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\& self.x}_n\texttt{: P}_n\ \texttt{\&\& c}$$

Similarly the type invariant associated with any class `C` that implements interfaces $\texttt{I}_1\texttt{, } \ldots\texttt{, I}_k$, extends class `D` and defines properties $\texttt{x}_1\texttt{: P}_1\texttt{, } \ldots\texttt{, x}_n\texttt{: P}_n$ and specifies a guard `c` is given by:

$$inv(\texttt{D})\ \texttt{\&\&}\ inv(\texttt{I}_1)\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ inv(\texttt{I}_k)\ \texttt{\&\& self.x}_1\texttt{: P}_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\& self.x}_n\texttt{: P}_n\ \texttt{\&\& c}$$

It is required that the type invariant associated with a class entail the type invariants of each interface that it implements.

It is guaranteed that for any variable `v` of type `T{c}` (where `T` is an interface name or a class name) the only objects `o` that may be stored in `v` are such that `o` satisfies $inv(\texttt{T}[\texttt{o}/\texttt{this}]) \wedge \texttt{c}[\texttt{o}/\texttt{self}]$.

4.5.5 Consistency of dependent types

A dependent type `C{c}` may contain zero or more parameters. We require that a type never be empty—so that it is possible for a variable of the type to contain a value. This is accomplished by requiring that the constraint `c` must be satisfiable *regardless* of the value assumed by parameters to the constraint (if any). Formally, consider a type `T = C{c}`, with the variables $\texttt{f}_1\texttt{: F}_1\texttt{, } \ldots\texttt{, f}_k\texttt{: F}_k$ free in `c`. Let $S = \texttt{\{f}_1\texttt{: F}_1\texttt{, } \ldots\texttt{, f}_k\texttt{: F}_k\texttt{, f}_{k+1}\texttt{: F}_{k+1}\texttt{, } \ldots\texttt{, f}_n\texttt{: F}_n\texttt{\}}$ be the smallest set of declarations containing $\texttt{f}_1\texttt{: F}_1\texttt{, } \ldots\texttt{, f}_k\texttt{: F}_k$ and closed under the rule: `f: F` in S if a reference to variable `f` (which is declared as `f: F`) occurs in a type in S.

(NOTE: The syntax rules for the language ensure that S is always finite. The type for a variable `v` cannot reference a variable whose type depends on `v`.)

We say that `T = C{c}` is *parametrically consistent* (in brief: *consistent*) if:

- Each type $\texttt{F}_1\texttt{, } \ldots\texttt{, F}_n$ is (recursively) parametrically consistent, and
- It can be established that $\forall\texttt{f}_1\texttt{: F}_1\texttt{, } \ldots\texttt{, f}_n\texttt{: F}_n\texttt{. } \exists\texttt{self: C. c \&\& } inv\texttt{(C)}$.

where inv`(C)` is the invariant associated with the type `C` (§4.5.4). Note by definition of S the formula above has no free variables.

STATIC SEMANTICS RULE: For a declaration `v: T` to be type-correct, `T` must be parametrically consistent. The compiler issues an error if it cannot determine the type is parametrically consistent.

Example 4.5.1 A class that represents a line has two distinct points:

```
class Line(start: Point,
           end: Point{self != this.start}) {...}
```

□

One can use dependent type to define other closed geometric figures as well.

Example 4.5.2 Here is an example:

```
class Point(x: Int, y: Int) {...}
```

To see that the declaration `end: Point{self != start}` is parametrically consistent, note that the following formula is valid:

$\forall$`this: Line.` $\exists$`self: Point. self != this.start`

since the set of all `Point`s has more than one element. □

Example 4.5.3 A triangle has three lines sharing three vertices.

```
class Triangle
 (a: Line,
  b: Line{a.end == b.start},
  c: Line{b.end == c.start && c.end == a.start})
 { ... }
```

Given `a: Line`, the type `b: Line{a.end == b.start}` is consistent, and given the two, the type `c: Line{b.end == c.start, c.end == a.start}` is consistent.

□

4.6 Function types

Function types are defined via the => type constructor. Closures (§12.5) and method selectors (§12.6) are of function type. The general form of a function type is:

```
[X1, ..., Xm](x1: T1, ..., xn: Tn){c} => T
        throws S1, ..., Sk
```

This is the type of functions that take type parameters $\mathtt{X}_i$ and value parameters $\mathtt{x}_i$ of types $\mathtt{T}_i$ such that the guard `c` holds and returns a value of type `T` or throws exceptions of types $\mathtt{S}_i$.

The type and value parameters are in scope throughout the function signature—they may be used in the types of other formal parameters and in the return type. Value parameters names (but not types) may be omitted if they are not used. Like other types, function types may be constrained.

FunctionType ::= *TypeParameters*? `(` *Formals*? `)` *Constraint*? `=>` *Type Throws*?
TypeParameters ::= `[` *TypeParameter* (`,` *TypeParameter*)* `]`
TypeParameter ::= *Identifier*
Formals ::= *Formal* (`,` *Formal*)*

4.6.1 Functions are value objects

Functions in X10 are value objects. The function type `[`$\mathtt{X}_1$`, ..., `$\mathtt{X}_k$`](`$\mathtt{x}_1$`: `$\mathtt{T}_1$ `..., `$\mathtt{x}_n$`: `$\mathtt{T}_n$`) => S` may be considered equivalent to an interface type with a method:

```
def apply[X_1, ..., X_k](x_1: T_1, ..., x_n: T_n): Z;
```

Classes and interfaces may implement or extend a function type by defining the `apply` method. Since each function type is anonymous, a class or interface may implement or extend more than one function type as long as the `apply` methods do not have the same signature.

As objects, closure body may refer to `this`, which is a reference to the current function, and use it to invoke the closure recursively.

4.7 Annotated types

Any X10 type may be annotated with zero or more user-defined *type annotations* (§17).

Annotations are defined as (constrained) interface types and are processed by compiler plugins, which may interpret the annotation symbolically.

A type `T` is annotated by with interface types $\mathtt{A}_1, \ldots, \mathtt{A}_n$ using the syntax `@`$\mathtt{A}_1$ ... `@`$\mathtt{A}_n$ `T`.

4.8 Subtyping and type equivalence

Subtyping is relation between types. It is the reflexive, transitive closure of the *direct subtyping* relation, defined as follows.

Class types. A class type is a direct subtype of any class it is declared to extend. A class type is direct subtype of any interfaces it is declared to implement.

Interface types. An interface type is a direct subtype of any interfaces it is declared to extend.

Boxed types. The type `Box[T]` is a subtype of any interface implemented by `T`.

Function types. Function types are covariant on their return type and contravariant on their argument types. For instance, a function type `(S1) => T1` is a subtype of another function type `(S2) => T2` if `S2` is a subtype of `S1` and `T1` is a subtype of `T2`.

Constrained types. Two dependent types `C{c}` and `C{d}` are said to be *equivalent* if `c` is true whenever `d` is, and vice versa. Thus, $[\![\texttt{C\{c\}}]\!] = [\![\texttt{C\{d\}}]\!]$.

Note that two dependent type that are syntactically different may be equivalent. For instance, `Int{self >= 0}` and `Int{self == 0 || self > 0}` are equivalent though they are syntactically distinct. The Java type system is essentially a nominal system—two types are the same if and only if they have the same name. The X10 type system extends the nominal type system of Java to permit constraint-based equivalence.

A dependent type `C{c}` is a subtype of a type `C{d}` if `c` implies `d`. In such a case we have $[\![\texttt{C\{c\}}]\!]$ is a subset of $[\![\texttt{C\{d\}}]\!]$. All dependent types defined on a class `C` refine the unconstrained class type `C`; `C` is equivalent to `C{true}`.

Path types. A path type `p.X` is a subtype of a type `T` if `p` has type `C{c}` (where `C` has a type member `X`) and `c` implies that `self.X` is a subtype of `T`.

Type parameters. A type parameter `X` of a class or interface `C` is a subtype of a type `T` if the class invariant of `C` implies that `X` is a subtype of `T`. Similarly, `T` is a subtype of parameter `X` if the class invariant implies the relationship.

A type parameter `X` of a method `m` is a subtype of a type `T` if the guard of `m` implies that `X` is a subtype of `T`. Similarly, `T` is a subtype of parameter `X` if the guard implies the relationship.

4.9 Least common ancestor of types

To compute the type of conditional expressions (§12.22), and of rail constructors (§12.29), the least common ancestor of types must be computed.

The least common ancestor of two types $\texttt{T}_1$ and $\texttt{T}_2$ is the unique most-specific type that is a supertype of both $\texttt{T}_1$ and $\texttt{T}_2$.

If the most-specific type is not unique (which can happen when $\texttt{T}_1$ and $\texttt{T}_2$ both implement two or more incomparable interfaces), then least common ancestor type is `x10.lang.Object`.

4.10 Coercions and conversions

X10 v1.7 supports the following coercions and conversions

4.10.1 Coercions

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast.

Subsumption coercion. A subtype may be implicitly coerced to any supertype.

Explicit coercion (casting with `as`) A reference type may be explicitly coerced to any other reference type using the `as` operation. If the value coerced is not an instance of the target type, a `TypeCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

4.10.2 Conversions

A *conversion* may change object identity if the type being converted to is not the same as the type converted from.

Narrowing conversion. A value class may be explicitly converted to any superclass using the `as` operation.

Widening numeric conversion. A numeric type may be implicitly converted to a wider numeric type. In particular, an implicit conversion may be performed between a numeric type and a type to its right, below:

```
Byte < Short < Int < Long < Float < Double
```

Boxing conversion. A type `T` may be implicitly converted to the boxed type `Box[T]`.

Unboxing conversion. A value type `Box[T]` may be implicitly converted to the type `T`.

Explicit conversion (casting with `as`) A type may be explicitly converted to any type to which it can be implicitly converted or implicitly or explicitly coerced.

String conversion. Any object that is an operand of the binary + operator may be converted to `String` if the other operand is a `String`. A conversion to `String` is performed by invoking the `toString()` method of the object.

4.11 Built-in types

The package `x10.lang` provides a number of built-in class and interface declarations that can be used to construct types.

4.11.1 The interface `Object`

The interface `x10.lang.Object` is the supertype of all reference and value classes. A variable of this type can hold a reference to an instance of any reference type, including boxed value types.

```
package x10.lang;
public interface Object {
  def toString(): String {...}
  def equals(o: Object): Boolean {...}
  def hashCode(): Int {...}
}
```

The method `equals` and `hashCode` are useful in hash tables, and are defined as in Java. The default implementation of `equals` is stable equality, §12.23. This method may be overridden in a (value or reference) subclass.

4.11.2 The class `Ref`

The class `x10.lang.Ref` is the superclass of all reference classes. A variable of this type can hold a reference to an instance of any reference type, including boxed value types.

4.11.3 The class `Value`

The value class `x10.lang.Value` is the superclass of by all value classes. A variable of this type can hold an instance of a value type.

4.11.4 The class `String`

All strings in X10 are instances of the value class `x10.lang.String`. A string object is immutable, and has a concatenation operator (+) available on it.

4.11.5 Primitive value classes

Several value types are provided that encapsulate abstractions (such as fixed point and floating point arithmetic) commonly implemented in hardware by modern computers.

```
package x10.lang;

public value Boolean { }
```

```
public abstract value Number { }
public abstract value Integer extends Number { }

public value Byte extends Integer  { }
public value Short extends Integer { }
public value Int extends Integer   { }
public value Char extends Integer  { }
public value Long extends Integer  { }

public value Float extends Number  { }
public value Double extends Number { }
```

A program may contain literals that stand for specific instances of these classes. The syntax for literals is the same as for Java (§3).

Binary and unary operations produce new instances of these classes.

Values of one numeric class may be coerced into values of another numeric class via the `as` operation.

Future Extensions. *X10 may provide mechanisms in the future to permit the programmer to specify how a specific value class is to be mapped to special hardware operations (e.g., along the lines of [1]). Similarly, mechanisms may be provided to permit the user to specify new syntax for literals.*

4.11.6 Array types

Arrays in X10 are instances of the class `x10.lang.Array`. Because of the importance of arrays in X10, the language supports more concise syntax for accessing array elements and performing operations on arrays.

The array type `Array[T]` is the type of all reference arrays of base type `T`. Such an array can take on any distribution, over any region.

Both array classes implement the function type `(Point) => T`; the element of array `A` at point `p` may be accessed using the syntax `A(p)`. The `Array` class also implements the `Settable[Point,T]` interface permitting assignment to an array element using the syntax `A(p) = v`.

X10 also supports dependent types for arrays, e.g., `Array[Double]{rank==3}` is the type of all arrays of `Double` of rank `3`. The `Array` class has distribution, region, and rank properties. X10 v1.7 defines type definitions that allows a distribution, region, or rank to be specified with on the array type.

```
package x10.lang;
type Array[T](n: Int) = Array[T]{rank==n};
type Array[T](d: Dist) = Array[T]{dist==d};
type Array[T](r: Region) = Array[T]{region==r};
```

4.11.7 Rails

A *rail* is a one-dimensional, zero-based, local array. It is more primitive than the `Array` class. Rails are indexed by integers rather than multi-dimensional points. Rails have a single `length` property of type `Int`. Rails can be mutable or immutable and are defined by the following class definitions:

```
package x10.lang;
public value class ValRail[T](length: Int) extends (Int)=>T { }
public class Rail[T](length: Int) extends (Int)=>T, Settable[Int,T] { }
```

X10 supports shorthand syntax for rail construction (§12.29).

4.11.8 Future types

The interface `x10.lang.Future[T]` is the type of all `future` expressions. The type represents a value which when forced will return a value of type `T`. The interface makes available the following methods:

```
package x10.lang;
public interface Future[T] implements () => T {
  public def apply(): T;
  public def force(): T;
  public def forced(): Boolean;
}
```

4.12 Type inference

X10 v1.7 supports limited local type inference, permitting variable types and return types to be elided. It is a static error if an omitted type cannot be inferred or uniquely determined.

4.12.1 Variable declarations

The type of a variable declaration can be omitted if the declaration has an initializer. The inferred type of the variable is the computed type of the initializer.

4.12.2 Return types

The return type of a method can be omitted if the method has a body (i.e., is not `abstract` or `extern`). The inferred return type is the computed type of the body.

The return type of a closure can be omitted. The inferred return type is the computed type of the body.

The return type of a constructor can be omitted if the constructor has a body (i.e., is not `extern`). The inferred return type if the enclosing class type with properties bound to the arguments in the constructor's `property` statement, if any, or to the unconstrained class type.

The inferred type of a method or closure body is the least common ancestor of the types of the expressions in `return` statements in the body. If the method does not return a value, the inferred type is `Void`.

4.12.3 Type arguments

A call to a polymorphic method, closure, or constructor may omit the explicit type arguments. If the method has a type parameter `T`, the type argument corresponding to `T` is inferred to be the least common ancestor of the types of any formal parameters of type `T`.

Consider the following method:

```
def choose[T](a: T, b: T): T { ... }
```

Given `Set[T] <: Collection[T]` and `List[T] <: Collection[T]`, in the following snippet, the algorithm should infer the type `Collection` for `x`.

```
intSet: Set[Int];
stringList: List[String];
val x = choose(intSet, stringList);
```

And in this snippet, the algorithm should infer the type `Collection[Int]` for `y`.

```
intSet: Set[Int];
intList: List[Int];
val y = choose(intSet, intList);
```

Finally, in this snippet, the algorithm should infer the type `Collection{T <: Number}` for `z`.

```
intSet: Set[Int];
numList: List{T <: Number};
val z = choose(intSet, numList);
```

Now, consider the following example:

```
def union[T](a: Set[T], b: Set[T]) : Set[T];
```

The union method cannot be called with just arguments of type `Set`.

```
set1: Set;
set2: Set;
val a = union(set1, set2);
```

This is illegal because the type system cannot demonstrate that `set1.T` and `set2.T` are equal. The following, however, is acceptable:

```
set1: Set;
set2: Set[set1.T];
val a = union(set1, set2);
```

However, unlike `union` above, the following method can be called with an argument of type `Set` because there are no constraints on `T`:

```
def unmodifiableSet[T](set: Set[T]): Set[T];
```

Using desugared syntax, the method is equivalent to:

```
def unmodifiableSet[T](set: Set{self.T==T}): Set{self.T==T};
```

Any `Set` can be passed in: for an argument `e`, the method is instantiated on `e.T`. Note that if this method were defined as:

```
def unmodifiableSet(set: Set): Set;
```

then the connection between the element types of the argument and of the return types would be broken. However, one could use the following signature to keep the connection, without introducing a method type parameter.

```
def unmodifiableSet(set: Set): Set[set.T];
```

5 Variables

A variable is a storage location. All variables are initialized with a value and cannot be observed without a value.

Variables whose value may not be changed after initialization are called *final variables* (or sometimes *constants*). The programmer indicates that a variable is final by declaring it with the `val` keyword rather than the `var` keyword. Variables that are annotated neither `val` nor `var` are considered final.

A variable of a reference data type `T` where `T` is the name of a reference class (possibly with type arguments) always holds a reference to an instance of the class `T` or a class that is a subclass of `T`, or a `null` reference.

A variable of a reference rail type `Rail[T]` has as many variables as the size of the rail.

A variable of an interface type `I` always holds either a reference to a reference class implementing `I` (including possibly a boxed value class that implements `I`), or a `null` reference.

A variable of a value data type `T` where `T` is the name of a value class always holds a reference to an instance of `T` (or a class that is a subclass of `T`) or to an instance of `T` (or a class that is a subclass of `T`). No program can distinguish between the two cases.

A variable of an function type always holds an instance of a value class with implementing the function type.

A variable of a reference rail type `ValRail[T]` has as many variables as the size of the rail. Each of these variables is immutable and has the type `T`.

X10 supports seven kinds of variables: constant *class variables* (static variables), *instance variables* (the instance fields of a class), *array components*, *method parameters*, *constructor parameters*, *exception-handler parameters* and *local variables*.

5.1 Final variables

A final variable satisfies two conditions:

- it can be assigned to at most once,
- it must be assigned to before use.

X10 follows Java language rules in this respect [5, §4.5.4,8.3.1.2,16]. Briefly, the compiler must undertake a specific analysis to statically guarantee the two properties above.

Final local variables and fields are defined by the `val` keyword. Elements of value arrays are also final.

5.2 Initial values of variables

Every variable declared at a type must always contain a value of that type.

Every class variable must be initialized before it is read, through the execution of an explicit initializer or a static block. Every instance variable must be initialized before it is read, through the execution of an explicit initializer or a constructor. Non-final instance variables of reference type are assumed to have an initializer that sets the value to `null`. Non-final instance variables of value type are assumed to have an initializer that sets the value to the result of invoking the nullary constructor on the class. An initializer is required if the default initial value is not assignable to the variable's type.

Each method and constructor parameter is initialized to the corresponding argument value provided by the invoker of the method. An exception-handling parameter is initialized to the object thrown by the exception. A local variable must be explicitly given a value by initialization or assignment, in a way that the compiler can verify using the rules for definite assignment [5, § 16].

6 Objects

An object is an instance of a scalar class or an array type. It is created by using a allocation expression (§12.24) or an array creation (§16.4) expression, such as an array initializer. An object that is an instance of a reference (value) type is called a *reference* (*value*) *object*.

All reference classes subclass from `x10.lang.Ref`. This class has one property `location` of type `x10.lang.Place`. Thus all reference objects in X10 are located (have a place). In X10 v1.7 a reference object stays resident at the place at which it was created for its entire lifetime.

X10 has no operation to dispose of a reference. Instead, the collection of all objects across all places is globally garbage collected.

Unlike Java, X10 objects do not have any synchronization information (e.g., a lock) associated with them. Instead, programmers should use atomic blocks (§14.11) for mutual exclusion and clocks (§15) for sequencing multiple parallel operations.

A reference object may have many references, stored in fields of objects or components of arrays. A change to an object made through one reference is visible through another reference.

Note that the creation of a remote async activity (§14.2) `A` at `P` may cause the automatic creation of references to remote objects at `P`. (A reference to a remote object is called a *remote object reference*, to a local object a *local object reference*.) For instance `A` may be created with a reference to an object at `P` held in a variable referenced by the statement in `A`. Similarly the return of a value by a `future` may cause the automatic creation of a remote object reference, incurring some communication cost. An X10 implementation should try to ensure that the creation of a second or subsequent reference to the same remote object at a given place does not incur any (additional) communication cost.

A reference to an object may carry with it the values of final fields of the object. The implementation should try to ensure that the cost of communicating the values

of final fields of an object from the place where it is hosted to any other place is not incurred more than once for each target place.

X10 does not have an operation (such as Pascal's "dereference" operation) which returns an object given a reference to the object. Rather, most operations on object references are transparently performed on the bound object, as indicated below. The operations on objects and object references include:

- Field access (§12.4). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely from place to place). The implementation should try to ensure that the cost of copying the field from the place where the object was created to the referencing place will be incurred at most once per referencing place, according to the rule for final fields discussed above.

- Method invocation (§12.8). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely). The X10 implementation must attempt to ensure that the cost of copying enough relevant state of the value object to enable this method invocation to succeed is incurred at most once for each value object per place.

- Casting (§12.25). An activity can perform this operation on local or remote objects, and should not incur communication costs (to bring over type information) more than once per place.

- `instanceof` operator (§12.26). An activity can perform this operation on local or remote objects, and should not incur communication costs (to bring over type information) more than once per place.

- The stable equality operator == and `!=` (§12.23). An activity can perform these operations on local or remote objects, and should not incur communication costs (to bring over relevant information) more than once per place.

7 Names and packages

X10 supports Java's mechanisms for names and packages [5, §6,§7], including `public`, `protected`, `private` and package-specific access control.

TypeName	::=	*Identifier*
	|	*TypeName* `.` *Identifier*
	|	*PackageName* `.` *Identifier*
PackageName	::=	*Identifier*
	|	*PackageName* `.` *Identifier*

7.1 Naming conventions

While not enforced by the compiler, classes and interfaces in the X10 library support the following naming conventions. Names of types—including classes, type parameters, and types specified by type definitions—are in CamelCase and begin with an uppercase letter. For backward compatibility with languages such as C and Java, type definitions are provided to allow primitive value types such as `int` and `boolean` to be written in lowercase. Names of methods, fields, value properties, and packages are in camelCase and begin with a lowercase letter. Names of `const` fields are in all uppercase with words separated by an "_".

8 Interfaces

X10 v1.7 interfaces are essentially the same Java interfaces [5, §9]. An interface primarily specifies signatures for public methods. It may extend multiple interfaces.

Future version of X10 will introduce additional structure in interface definitions that will allow the programmer to state additional properties of classes that implement that interface. For instance a method may be declared `pure` to indicate that its evaluation cannot have any side-effects. A method may be declared `local` to indicate that its execution is confined purely to the current place (no communication with other places). Similarly, behavioral properties of the method as they relate to the usage of clocks of the current activity may be specified.

8.1 Interfaces with properties

X10 permits interfaces to have properties and specify an interface invariant. This is necessary so that programmers can build dependent types on top of interfaces and not just classes.

NormalInterfaceDeclaration ::= *InterfaceModifiers*$^?$ `interface` *Identifier*
TypeParameterList$^?$ *PropertyList*$^?$ *Constraint*$^?$
ExtendsInterfaces$^?$ *InterfaceBody*

The invariant associated with an interface is the conjunction of the invariants associated with its superinterfaces and the invariant defined at the interface.

STATIC SEMANTICS RULE: The compiler declares an error if this constraint is not consistent (§4.5.5).

Each interface implicitly defines a nullary getter method `def p(): T` for each property `p: T`.

STATIC SEMANTICS RULE: The compiler issues a warning if the programmer explicitly defines a method with this signature for an interface.

A class `C` (with properties) is said to implement an interface `I` if

- its properties contains all the properties of `I`,
- its class invariant $inv(\texttt{C})$ implies $inv(\texttt{I})$.

9 Classes

The *class declaration* has a list of type parameters, value properties, a constraint (the *class invariant*, a single superclass, one or more interfaces, and a class body containing the the definition of fields, methods, and member types. Each such declaration introduces a class type (§4.1).

NormalClassDeclaration ::= *ClassModifiers*? `class` *Identifier*
TypeParameterList? *PropertyList*? *Guard*?
Super? *Interfaces*? *ClassBody*

TypeParameterList ::= `[` *TypeParameters* `]`
TypeParameters ::= *TypeParameter* (`,` *Typearameter*)*
TypeParameter ::= *Variance*? *Annotation** *Identifier*
Variance ::= `+`
| `-`

PropertyList ::= `(` *Properties* `)`
Properties ::= *Property* (`,` *Property*)*
Property ::= *Annotation** `val`? *Identifier* `:` *Type*

Super ::= `extends` *ClassType*
Interfaces ::= `implements` *InterfaceType* (`,` *InterfaceType*)*

ClassBody ::= *ClassMember**
ClassMember ::= *ClassDeclaration*
| *InterfaceDeclaration*
| *FieldDeclaration*
| *MethodDeclaration*
| *ConstructorDeclaration*

A type parameter declaration is given by an optional variance tag and an identifier. A type parameter must be bound to a concrete type when an instance of the class is created.

A value property has a name and a type. Value properties are accessible in the same way as `public final` fields.

STATIC SEMANTICS RULE: It is a compile-time error for a class defining a value property `x: T` to have an ancestor class that defines a value property with the name `x`.

Each class `C` defining a property `x: T` implicitly has a field

```
public val x : T;
```

and a getter method

```
public final def x(): T { return x; }
```

Each interface `I` defining a property `x: T` implicitly has a getter method

```
public def x(): T;
```

STATIC SEMANTICS RULE: It is a compile-time error for a class or interface defining a property `x :T` to have an existing method with the signature `x(): T`.

Properties are used to build dependent types from classes, as described in §4.5.

The *Guard* in a class or interface declaration specifies an explicit condition on the properties of the type, and is discussed further in §9.3.

STATIC SEMANTICS RULE: Every constructor for a class defining properties $\texttt{x}_1\texttt{: T}_1\texttt{, } \ldots\texttt{, x}_n\texttt{: T}_n$ must ensure that each of the fields corresponding to the properties is definitely initialized (cf. requirement on initialization of final fields in Java) before the constructor returns.

Type parameters are used to define generic classes and interfaces, as described in §4.2.1.

Classes are structured in a single-inheritance code hierarchy, may implement multiple interfaces, may have static and instance fields, may have static and instance methods, may have constructors, may have static and instance initializers, may have static and instance inner classes and interfaces. X10 does not permit mutable static state, so the role of static methods and initializers is quite limited. Instead programmers should use singleton classes to carry mutable static state.

Method signatures may specify checked exceptions. Method definitions may be overridden by subclasses; the overriding definition may have a declared return type that is a subclass of the return type of the definition being overridden. Multiple methods with the same name but different signatures may be provided on a class (ad hoc polymorphism). The public/private/protected/package-protected access modification framework may be used.

Because of its different concurrency model, X10 does not support `transient` and `volatile` field modifiers.

9.1 Reference classes

A reference class is declared with the optional keyword `reference` preceding `class` in a class declaration. Reference class declarations may be used to construct reference types (§4.1). Reference classes may have mutable fields. Instances of a reference class are always created in a fixed place and in X10 v1.7 stay there for the lifetime of the object. (Future versions of X10 may support object migration.) Variables declared at a reference type always store a reference to the object, regardless of whether the object is local or remote.

9.2 Value classes

X10 singles out a certain set of classes for additional support. A class is said to be *stateless* if all of its fields are declared to be `final` (§5.1), otherwise it is *stateful*. (X10 has syntax for specifying an array class with final fields, unlike Java.) A *stateless (stateful) object* is an instance of a stateless (stateful) class.

X10 allows the programmer to signify that a class (and all its descendents) are stateless. Such a class is called a *value class*. The programmer specifies a value class by prefixing the modifier `value` before the keyword `class` in a class declaration. (A class not declared to be a value class will be called a *reference class*.) Each instance field of a value class is treated as `final`. It is legal (but neither required nor recommended) for fields in a value class to be declared final. For brevity, the X10 compiler allows the programmer to omit the keyword `class` after `value` in a value class declaration.

ValueClassDeclaration ::= *ClassModifiers*? `value` `class`? *Identifier*
TypePropertyList? *PropertyList*? *Guard*?
Super? *Interfaces*? *ValueClassBody*

The `Box` type constructor (§**??**) can be used to declare variables whose value may be `null` or a value type.

Stable equality for value types is defined through a deep walk, bottoming out in fields of reference types (§12.23).

STATIC SEMANTICS RULE: It is a compile-time error for a value class to inherit from a stateful class or for a reference class to inherit from a value class. All fields of a value class are implicitly declared `final`.

9.2.1 Representation

Since value objects do not contain any updatable locations, they can be freely copied from place to place. An implementation may use copying techniques even within a place to implement value types, rather than references. This is transparent to the programmer.

More explicitly, X10 guarantees that an implementation must always behave as if a variable of a reference type takes up as much space as needed to store a reference that is either null or is bound to an object allocated on the (appropriate) heap. However, X10 makes no such guarantees about the representation of a variable of value type. The implementation is free to behave as if the value is stored "inline", allocated on the heap (and a reference stored in the variable) or use any other scheme (such as structure-sharing) it may deem appropriate. Indeed, an implementation may even dynamically change the representation of an object of a value type, or dynamically use different representations for different instances (that is, implement automatic box/unboxing of values).

Implementations are strongly encouraged to implement value types as space-efficiently as possible (e.g., inlining them or passing them in registers, as appropriate). Implementations are expected to cache values of remote final value variables by default. If a value is large, the programmer may wish to consider spawning a remote activity (at the place the value was created) rather than referencing the containing variable (thus forcing it to be cached).

Example 9.2.1 A functional `LinkedList` program may be written as follows:

```
value LinkedList {
  val first: Object;
  val rest: LinkedList;
  public def this(first: Object) {
     this(first, null);
  }
  public def this(first: Object, rest: LinkedList) {
    this.first = first;
    this.rest = rest;
  }
  public def first(): Object {
    return first;
  }
  public def rest(): LinkedList {
    return rest;
  }
  public def append(l: LinkedList): LinkedList {
    return (this.rest == null)
        ? new LinkedList(this.first, l)
        : this.rest.append(l);
  }
}
```

Similarly, a `Complex` class may be implemented as follows:

```
value Complex {
  re: Double;
  im: Double;
  public def this(re: Double, im: Double) {
     this.re=re;
     this.im=im;
  }
  public def add(other: Complex): Complex {
    return new Complex(this.re+other.re,
                       this.im+other.im);
  }
  public def mult(other: Complex): Complex {
    return new Complex(this.re^2-other.re^2,
                       2*this.im*other.im);
```

```
  }
  ...
}
```

□

9.3 Type invariants

There is a general recipe for constructing a list of parameters or properties x_1: $T_1\{c_1\}$, ..., x_k: $T_k\{c_k\}$ that must satisfy a given (satisfiable) constraint c.

```
class Foo(x1: T1{x2: T2; ...; xk: Tk; c},
          x2: T2{x3: T3; ...; xk: Tk; c},
          ...
          xk: Tk{c}) {
  ...
}
```

The first type x_1: $T_1\{x_2$: T_2; ...; x_k: T_k; c} is consistent iff $\exists x_1$: T_1, x_2: T_2, ..., x_k: T_k. c is consistent. The second is consistent iff

$\forall x_1$: $T_1\{x_2$: T_2; ...; x_k: T_k; c}
$\exists x_2$: T_2. $\exists x_3$: T_3, ..., x_k: T_k. c

But this is always true. Similarly for the conditions for the other properties.

Thus logically every satisfiable constraint c on a list of parameters x_1, ..., x_k can be expressed using the dependent types of x_i, provided that the constraint language is rich enough to permit existential quantifiers.

Nevertheless we will find it convenient to permit the programmer to explicitly specify a depclause after the list of properties, thus:

```
class Point(i: Int, j: Int) { ... }
class Line(start: Point, end: Point){end != start}
  = { ... }
class Triangle (a: Line, b: Line, c: Line)
        {a.end == b.start && b.end == c.start &&
         c.end == a.start} = { ... }
class SolvableQuad(a: Int, b: Int, c: Int)
```

```
                    {a*x*x+b*x+c==0} = { ... }
class Circle (r: Int, x: Int, y: Int)
              {r > 0 && r*r==x*x+y*y} = { ... }
class NonEmptyList extends List{n > 0} {...}
```

Consider the definition of the class `Line`. This may be thought of as saying: the class `Line` has two fields, `start: Point` and `end: Point`. Further, every instance of `Line` must satisfy the constraint that `end != start`. Similarly for the other class definitions.

In the general case, the production for *NormalClassDeclaration* specifies that the list of properties may be followed by a *Guard*.

NormalClassDeclaration ::= *ClassModifiers*$^?$ `class` *Identifier* *TypeParameterList*$^?$ *PropertyList*$^?$ *Guard*$^?$ *Extends*$^?$ *Interfaces*$^?$ *ClassBody*

NormalInterfaceDeclaration ::= *InterfaceModifiers*$^?$ `interface` *Identifier* *TypeParameterList*$^?$ *PropertyList*$^?$ *Guard*$^?$ *ExtendsInterfaces*$^?$ *InterfaceBody*

All the properties in the list, together with inherited properties, may appear in the *Guard*. A guard `c` with value property list $\texttt{x}_1\texttt{: T}_1\texttt{, ..., x}_n\texttt{: T}_n$ for a class `C` is said to be consistent if each of the $\texttt{T}_i$ are consistent and the constraint

$$\exists \texttt{x}_1\texttt{: T}_1\texttt{, ..., x}_n\texttt{: T}_n\texttt{, self: C. c}$$

is valid (always true).

9.4 Class definitions

Consider a class definition

```
ClassModifiers?
class C(x1: P1, ..., xn: Pn) extends D{d}
   implements I1(:c1), ..., Ik{ck}
ClassBody
```

Each of the following static semantics rules must be satisfied:

STATIC SEMANTICS RULE (Int-implements): The type invariant inv(C) of C must entail c_i[this/self] for each i in $\{1, \ldots, k\}$

STATIC SEMANTICS RULE (Super-extends): The return type c of each constructor in *ClassBody* must entail d.

9.5 Constructor definitions

A constructor for a class C is guaranteed to return an object of the class on successful termination. This object must satisfy i(C), the class invariant associated with C (§4.5.4). However, often the objects returned by a constructor may satisfy *stronger* properties than the class invariant. X10's dependent type system permits these extra properties to be asserted with the constructor in the form of a constrained type (the "return type" of the constructor):

ConstructorDeclarator	::=	`def this` *TypeParameterList*$^?$ `(` *FormalParameterList*$^?$ `)` *ReturnType*$^?$ *Guard*$^?$ *Throws*$^?$
ReturnType	::=	`:` *Type*
Guard	::=	`""` *DepExpression* `""`
Throws	::=	`throws` *ExceptionType* `(` `,` *ExceptionType* `)`*
ExceptionType	::=	*ClassBaseType Annotation**

The parameter list for the constructor may specify a *guard* that is to be satisfied by the parameters to the list.

Example 9.5.1 Here is another example.

```
public class List[T](n: Int{n >= 0}) {
    protected head: Box[T];
    protected tail: List[T](n-1);
    public def this(o: T, t: List[T]) : List[T](t.n+1) = {
        n = t.n+1;
        tail = t;
        head = o;
    }
    public def this() : List[T](0) = {
        n = 0;
        head = null;
```

```
            tail = null;
        }
        ...
    }
```

The second constructor returns a `List` that is guaranteed to have `n==0`; the first constructor is guaranteed to return a List with `n>0` (in fact, `n==t.n+1`, where the argument to the constructor is `t`). This is recorded by the programmer in the constrained type associated with the constructor. □

STATIC SEMANTICS RULE (Super-invoke): Let `C` be a class with properties $\texttt{p}_1\texttt{: P}_1\texttt{, ..., p}_n\texttt{: P}_n$, invariant `c` extending the constrained type `D{d}` (where `D` is the name of a class).

For every constructor in `C` the compiler checks that the call to super invokes a constructor for `D` whose return type is strong enough to entail `d`. Specifically, if the call to super is of the form $\texttt{super(e}_1\texttt{, ..., e}_k\texttt{)}$ and the static type of each expression $\texttt{e}_i$ is $\texttt{S}_i$, and the invocation is statically resolved to a constructor $\texttt{def this(x}_1\texttt{: T}_1\texttt{, ..., x}_k\texttt{: T}_k\texttt{)\{c\}: D\{d}_1\texttt{\}}$ then it must be the case that

$$\texttt{x}_1\texttt{: S}_1\texttt{, ..., x}_i\texttt{: S}_i \vdash \texttt{x}_i\texttt{: T}_i \quad \texttt{(for } i \in \{1,\ldots,k\}\texttt{)}$$
$$\texttt{x}_1\texttt{: S}_1\texttt{, ..., x}_k\texttt{: S}_k \vdash \texttt{c}$$
$$\texttt{d}_1\texttt{[a/self] \&\& x}_1\texttt{: S}_1\texttt{ ... \&\& x}_k\texttt{: S}_k \vdash \texttt{d[a/self]}$$

where `a` is a constant that does not appear in $\texttt{x}_1\texttt{: S}_1 \wedge \texttt{...} \wedge \texttt{x}_k\texttt{: S}_k$.

STATIC SEMANTICS RULE (Constructor return): The compiler checks that every constructor for `C` ensures that the properties $\texttt{p}_1\texttt{,...,p}_n$ are initialized with values which satisfy `t(C)`, and its own return type `c'` as follows. In each constructor, the compiler checks that the static types $\texttt{T}_i$ of the expressions $\texttt{e}_i$ assigned to $\texttt{p}_i$ are such that the following is true:

$$\texttt{p}_1\texttt{: T}_1\texttt{, ..., p}_n\texttt{: T}_n \vdash \texttt{t(C)} \wedge \texttt{c'}$$

(Note that for the assignment of $\texttt{e}_i$ to $\texttt{p}_i$ to be type-correct it must be the case that $\texttt{p}_i\texttt{: T}_i \wedge \texttt{p}_i\texttt{: P}_i$.)

STATIC SEMANTICS RULE (Constructor invocation): The compiler must check that every invocation $\texttt{C(e}_1\texttt{, ..., e}_n\texttt{)}$ to a constructor is type correct: each argument $\texttt{e}_i$ must have a static type that is a subtype of the declared type $\texttt{T}_i$ for the ith argument of the constructor, and the conjunction of static types of the argument must entail the *Guard* in the parameter list of the constructor.

9.6 Field definitions

Not every instance of a class needs to have every field defined on the class. In Java-like languages this is ensured by conditionally setting fields to a default value, such as `null`, in those instances where the fields are not needed.

Consider the class `List` used earlier. Here all instances of `List` returned by the second constructor do not need the fields `value` and `tail`; their value is set to null.

X10 permits a much cleaner solution that does not require default values such as null to be stored in such fields. X10 permits fields to be *guarded* with a constraint. The field is accessible only if the *guard* constraint is satisified.

FieldDeclaration	::=	*FieldModifiers*$^?$ `val` *VariableDeclarators* `;`
	|	*FieldModifiers*$^?$ `var` *VariableDeclarators* `;`
VariableDeclarators	::=	*VariableDeclarator* (`,` *VariableDeclarator*)*
VariableDeclarator	::=	*Identifier* (*Constraint*)$^?$ (`:` *Type*)$^?$ (`=` *Expression*)$^?$

It is illegal for code to access a guarded field through a reference whose static type does not satisfy the associated guard, even implicitly (i.e., through an implicit `this`). Rather the source program should contain an explict cast, e.g., `me: C{c} = this as C{c}`.

STATIC SEMANTICS RULE: Let `f` be a field defined in class `C` with guard `c`. The compiler declares an error if field `f` is accessed through a reference `o` whose static type is not a subtype of `C{c}`.

Example 9.6.1 We may now rewrite the List example:

```
public class List(n: Int{n>=0}) {
  protected val head{n>0}: Object;
  protected val tail{n>0}: List(n-1);
  public def this(o: Object, t: List): List(t.n+1) {
     property(t.n+1);
     head=o;
     tail=t;
  }
  public def this(): List(0) {
     property(0);
  }
```

```
  ...
}
```

The fields `value` and `tail` do not exist for instances of the class `List(0)`. □

It is a compile-time error for a class to have two fields of the same name, even if their guards are different. A class `C` with a field named `f` is said to *hide* a field in a superclass named `f`.

STATIC SEMANTICS RULE: A class may not declare two fields with the same name.

To avoid an ambiguity, it is a static error for a class to declare a field with a function type (§4.6) with the same name and signature as a method of the same class.

9.6.1 Field hiding

A subclass that defines a field `f` hides any field `f` declared in a superclass, regardless of their types. The superclass field `f` may be accessed within the body of the subclass via the reference `super.f`.

9.7 Method definitions

X10 permits guarded method definitions, similar to guarded field definitions. Additionally, the parameter list for a method may contain a *Guard*.

MethodDeclaration	::=	*MethodHeader* `;`
	|	*MethodHeader* `=` *ClosureBody*
MethodHeader	::=	*MethodModifiers*$^?$ `def` *Identifier TypeParameters*$^?$ `(` *FormalParameterList*$^?$ `)` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

In the formal parameter list, variables may be declared with `val` or `var`. If neither is specified, the variable is `val`.

The guard (specified by *Guard*) specifies a constraint `c` on the properties of the class `C` on which the method is being defined. The method exists only for those

instances of C which satisfy c. It is illegal for code to invoke the method on objects whose static type is not a subtype of C{c}.

STATIC SEMANTICS RULE: The compiler checks that every method invocation o.m(e_1, ..., e_n) for a method is type correct. Each each argument e_i must have a static type S_i that is a subtype of the declared type T_i for the ith argument of the method, and the conjunction of static types of the arguments must entail the guard in the parameter list of the method.

The compiler checks that in every method invocation o.m(e_1, ..., e_n) the static type of o, S, is a subtype of C{c}, where the method is defined in class C and the guard for m is equivalent to c.

Finally, if the declared return type of the method is D{d}, the return type computed for the call is D{a: S; x_1: S_1; ...; x_n: S_n; d[a/this]}, where a is a new variable that does not occur in d, S, S_1, ..., S_n, and x_1, ..., x_n are the formal parameters of the method.

Example 9.7.1 Consider the program:

```
public class List(n: Int{n>=0}) {
  protected val head{n>0}: Object;
  protected val tail{n>0}: List(n-1);
  public def this(o: Object, t: List): List(t.n+1) = {
     property(t.n+1);
     head=o;
     tail=t;
  }
  public def this(): List(0) = {
     property(0);
  }
  public def append(l: List): List{self.n==this.n+l.n} = {
      return (n==0)? l
         : new List(head, tail.append(l));
  }
  public def nth(k: Int{1 <= k && k <= n}){n > 0}: Object = {
      return k==1 ? head : tail.nth(k-1);
  }
}
```

The following code fragment

```
u: List{self.n==3} = ...
t: List{self.n==x} = ...;
s: List{self.n==x+3} = t.append(u);
```

will typecheck. The type of the expression `t.append(u)` is

```
List{a: List{self.n==x};
     l: List{self.n==3}; self.n==a.n+l.n}
```

which is equivalent to:

```
List{self.n==x+3}
```

□

The method body is either an expression, a block of statements, or a block ending with an expression.

9.7.1 Property methods

A method declared with the modifier `property` may be used in constraints. A property method declared in a class must have a body and must not be `void`. The body of the method must consist of only a single `return` statement or a single expression. It is a static error of the expression cannot be represented in the constraint system.

Property methods in classes are implicitly `final`; they cannot be overridden.

A property method definition may omit the formal parameters and the `def` keyword. That is, the following are equivalent:

```
property def rail(): boolean = rect && onePlace == here && zeroBased;
property rail: boolean = rect && onePlace == here && zeroBased;
```

9.7.2 Method overloading, overriding, hiding, shadowing and obscuring

The definitions of method overloading, overriding, hiding, shadowing and obscuring in X10 are the same as in Java, modulo the following considerations motivated by type parameters and dependent types.

Two or more methods of a class or interface may have the same name if they have a different number of type parameters, or they have value parameters of different types.

The definition of a method declaration $\texttt{m}_1$ "having the same signature as" a method declaration $\texttt{m}_2$ involves identity of types. Two X10 types are defined to be identical iff they are equivalent (§4.8). Two methods are said to have *the same signature* if (a) they have the same number of type parameters, (b) they have the same number of formal (value) parameters, and (c) for each formal parameter their types are equivalent. It is a compile-time error for there to be two methods with the same name and same signature in a class (either defined in that class or in a superclass).

STATIC SEMANTICS RULE: A class `C` may not have two declarations for a method named `m`—either defined at `C` or inherited:

```
def m[X1, ..., Xm](v1: T1{t1}, ..., vn: Tn{tn}){tc}: T {...}
def m[X1, ..., Xm](v1: S1{s1}, ..., vn: Sn{sn}){sc}: S {...}
```

if it is the case that the types `C{tc}`, $\texttt{T}_1\texttt{\{t}_1\texttt{\}}$, ..., $\texttt{T}_n\texttt{\{t}_n\texttt{\}}$ are equivalent to the types `C{sc}`, $\texttt{S}_1\texttt{\{t}_1\texttt{\}}$, ..., $\texttt{T}_n\texttt{\{t}_n\texttt{\}}$ respectively.

In addition, the guard of a overriding method must be no stronger than the guard of the overridden method. This ensures that any virtual call to the method satisfies the guard of the callee.

STATIC SEMANTICS RULE: If a class `C` overrides a method of a class or interface `B`, the guard of the method in `B` must entail the guard of the method in `C`.

A class `C` inherits from its direct superclass and superinterfaces all their methods visible according to the access restriction modifiers public/private/protected/(package) of the superclass/superinterfaces that are not hidden or overridden. A method $\texttt{M}_1$ in a class `C` overrides a method $\texttt{M}_2$ in a superclass `D` if $\texttt{M}_1$ and $\texttt{M}_2$ have the same signature. Methods are overriden on a signature-by-signature basis.

A method invocation $\texttt{o.m(e}_1\texttt{, ..., e}_n\texttt{)}$ is said to have the *static signature* <`T`, $\texttt{T}_1\texttt{, ..., T}_n$> where `T` is the static type of `o`, and $\texttt{T}_1, \ldots, \texttt{T}_n$ are the static types of $\texttt{e}_1, \ldots, \texttt{e}_n$, respectively. As in Java, it must be the case that the compiler can determine a single method defined on `T` with argument type $\texttt{T}_1, \ldots \texttt{T}_n$; otherwise, a compile-time error is declared. However, unlike Java, the X10 type `T` may be a dependent type `C{c}`. Therefore, given a class definition for `C` we must determine which methods of `C` are available at a type `C{c}`. But the answer to this question is clear: exactly those methods defined on `C` are available at the type `C{c}` whose guard `d` is implied by `c`.

Example 9.7.2 Consider the definitions:

```
class Point(i: Int, j: Int) {...}
class Line(s: Point, e: Point{self != i}) {
  // m1: Both points lie in the right half of the plane
  def draw(){s.i>= 0 && e.i >= 0} = {...}
  // m2: Both points lie on the y-axis
  def draw(){s.i== 0 && e.i == 0} = {...}
  // m3: Both points lie in the top half of the plane
  def draw(){s.j>= 0 && e.j >= 0} = {...}
  // m4: The general method
  def draw() = {...}
}
```

Three different implementations are given for the `draw` method, one for the case in which the line lies in the right half of the plane, one for the case that the line lies on the y-axis and the third for the case that the line lies in the top half of the plane.

Consider the invocation

```
m: Line{s.i < 0} = ...
m.draw();
```

This generates a compile time error because there is no applicable method definition.

Consider the invocation

```
m: Line{s.i>=0 && s.j>=0 && e.i>=0 && e.j>=0} = ...
m.draw();
```

This generates a compile time error because both `m1` and `m3` are applicable.

Consider the invocation

```
m: Line{s.i>=0 && s.j>=0 && e.i>=0} = ...
m.draw();
```

This does not generate any compile-time error since only m1 is applicable. □

In the last example, notice that at runtime `m1` will be invoked (assuming `m` contains an instance of the `Line` class, and not some subclass of `Line` that overrides this

method). This will be the case even if `m` satisfies at runtime the stronger conditions for `m2` (i.e., `s.i==0 && e.i==0`). That is, dynamic method lookup will not take into account the "strongest" constraint that the receiver may satisfy, i.e., its "strongest constrained type".

RATIONALE: The design decision that dynamic method lookup should ignore dependent type information was made to keep the design and the implementation simple and to ensure that serious errors such as method invocation errors are captured at compile-time.

Consider the above example and the invocation

```
m: Line = ...
m.draw();
```

Statically the compiler will not report an error because m4 is the only method that is applicable. However, if dynamic method lookup were to use constrained types then we would face the problem that if m is a line that lives in the upper right quadrant then both `m2` and `m3` are applicable and one does not override the other. Hence we must report an error dynamically.

As discussed above, the programmer can write code with `instanceof` and class casts that perform any application-appropriate discrimination.

9.7.3 Method annotations

`atomic` annotation

A method may be declared `atomic`.

MethodModifier ::= `atomic`

Such a method is treated as if the statement in its body is wrapped implicitly in an `atomic` statement.

`local` annotation

A method may be declared `local`.

MethodModifier ::= `local`

By declaring a method `local` the programmer asserts that while executing this method an activity will only access local memory.

The compiler implements the following rules to guarantee this property.

Let `o` be any expression occurring in the body of the method. Assume its static datatype is `F`.

- Local methods can only be overridden by local methods.
- If the body of the method contains any field access `o.e`, then the static placetype of `o` must be `here`.

  The programmer can always ensure that this condition is satisfied (albeit at the risk of introducing a runtime exception) by replacing each field access `o.e` with `(o as F!here).e`.

- If the body of the method contains any assignments to fields (e.g. `o.e Op= t`, or `Op o.e` or `o.e Op`) then the static placetype of `o` must be `here`.

  The programmer can always ensure that this condition is satisfied by replacing `o.e Op= t` by `o1.e Op=t` and preceding it (in the same basic block) with the local variable declaration `o1: F!here = o as F!here` (for some new local variable `o1`). Similarly for `Op o.e` and `o.e Op`.

- Recall that the static placetype of an array access `o(e)` is `o.dist(e)`. Therefore, any read/write array access `o(e)` must be guarded by the condition `o.dist(e) == here`. (Since `e` may have side-effects, the compiler must ensure that the place check uses the value returned by the same expression evaluation that is used to access the array element.)

- If the body of the method contains any method invocation $\texttt{o.m(t}_1\texttt{,...,t}_k\texttt{)}$ then the method invoked must be local. Additionally, the static place type of `o` must be `here`. As above, the programmer can always ensure the second condition is satisfied by writing such a method invocation as $\texttt{(o as F!here).m(t}_1\texttt{,...,t}_k\texttt{)}$.

Note that reads/writes to local variables or method parameters are always local, hence the compiler does not have to check any extra conditions.

A method declared `atomic` is automatically declared to be `local`.

9.8 Type definitions

With value arguments, type arguments, and constraints, the syntax for X10 types can often be verbose; X10 therefore provides *type definitions* to allow users to define new type constructors.

Type definitions have the following syntax:

TypeDefinition ::= `type` *Identifier* (`[` *TypeParameters* `]`)$^?$ (`(` *Formals* `)`)$^?$ *Constraint*$^?$ = *Type*

A type definition can be thought of as a type-valued function, mapping type parameters and value parameters to a concrete type. The following examples are legal type definitions:

```
type StringSet = Set[String];
type MapToList[K,V] = Map[K,List[V]];
type Nat = Int{self>=0};
type Int(x: Int) = Int{self==x};
type Int(lo: Int, hi: Int) = Int{lo <= self, self <= hi};
```

As the two definitions of `Int` demonstrate, type definitions may be overloaded: two type definitions with different numbers of type parameters or with different types of value parameters, according to the method overloading rules (§9.7.2), define distinct type constructors.

Type definitions may appear as a class or interface member or in a block statement.

Type definitions are applicative, not generative; that is, they define aliases for types but do not introduce new types. Thus, the following code is legal:

```
type A = Int;
type B = String;
type C = String;
a: A = 3;
b: B = new C("Hi");
c: C = b + ", Mom!";
```

If a type definition has no type parameters and no value parameters and is an alias for a class type, then a `new` expression may be used to create an instance of the class using the type definition's name. Given the following type definition:

```
type A = C[T1, ..., Tk]{c};
```

where `C[`$\mathtt{T}_1$`, ..., `$\mathtt{T}_k$`]` is a class type, a constructor of `C` may be invoked with `new A(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)`, if the invocation `new C[`$\mathtt{T}_1$`, ..., `$\mathtt{T}_k$`](`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)` is legal and if the constructor return type is a subtype of `A`.

10 Variable declarations

Variables declarations are used to declare formal parameters, fields, properties, and local variables.

VarDeclaratorWithType	::=	*VarDeclaratorId ResultType*
VarDeclarator	::=	*VarDeclaratorId ResultType*$^?$
VarDeclaratorId	::=	*Annotation** *Identifier*
	|	*Annotation** *Identifier* `(` *VarDeclaratorList* `)`
	|	*Annotation** `(` *VarDeclaratorList* `)`
VarDeclaratorList	::=	*VarDeclarator* `(` `,` *VarDeclarator* `)`*
ResultType	::=	`:` *Type*

Destructuring syntax. X10 permits a *destructuring* syntax for local variable declarations and formal parameters. At present, X10 v1.7 supports this feature only for variables of type `Point`; future versions of the language may support general pattern matching. Intuitively, this syntax allows a point to be "destructured" into its corresponding `Int` indices in a pattern-matching style. The kth declarator in a `Point` *VarDeclaratorList* is treated as a variable of type `Int` that is initialized with the value of the kth index of the point. The second form of the syntax permits the specification of only the index variables.

For example, the following code binds the `Int` variable `x` to `0` and `y` to `1`, and the variable `p` to the point object.

```
p(i,j): Point = new Point(0,1);
```

Formal parameters. Formal parameters are always declared with a type. Formals may be declared either final using the `val` or `var`; a declaration with neither

keyword is final. The variable name can be omitted if it is not to be used in the scope of the declaration.

```
Formal ::= FormalModifier* var VarDeclaratorWithType
         | FormalModifier* val VarDeclaratorWithType
         | FormalModifier* VarDeclaratorWithType
         | Type
FormalModifier ::= Annotation
                 | shared
```

Local variables. Local variable declarations may have optional initializer expressions. The initializer must be a subtype of the declared type of the variable. If the variable is final (i.e., is declared `val`) the type may be omitted and inferred from the initializer type (§4.12).

```
LocalDeclaration ::= LocalModifier* var LocalDeclaratorsWithType
                     ( , LocalDeclaratorsWithType )*
                   | LocalModifier* val LocalDeclarators
                     ( , LocalDeclarators )*
                   | LocalModifier* LocalDeclaratorsWithType
                     ( , LocalDeclaratorsWithType )*
LocalDeclarators ::= LocalDeclaratorsWithType
                 ::= LocalDeclaratorWithInit
LocalDeclaratorWithInit ::= VarDeclarator Init
LocalDeclaratorsWithType ::= VarDeclaratorId ( , VarDeclaratorId )* ResultType
LocalModifier ::= Annotation
                | shared
Init ::= = Expression
```

Fields. Fields are declared either `var` (non-final, non-static), `val` (final, non-static), or `const` (final, static); the default is `val`. Field declarations may have optional initializer expressions. The initializer must be a subtype of the declared type of the variable. For `var` fields, if the initializer is omitted, the constructor must initialize the field, or else the field is initialized with `null` if a reference type `0` if an `Int`, `0L` if a `Long`, `0.0F` if a `Float`, `0.0` if a `Double`, or `false` if a `Boolean`. It is a static error if the default value is not a member of the type (e.g., it is a static error to elide the initializer for `int{self==1}`).

If the variable is final, the type may be omitted and inferred from the initializer type (§4.12). Mutable fields must be declared with a type. A field declaration may have an optional *Guard*, restricting how the field may be accessed. The compiler is free to not allocate storage for the field if the guard of a field cannot be satisfied for a given containing object.

FieldDeclaration ::= *FieldModifier** `var` *FieldDeclaratorsWithType*
*(, FieldDeclaratorsWithType)**
| *FieldModifier** `const` *FieldDeclarators*
*(, FieldDeclarators)**
| *FieldModifier** `val` *FieldDeclarators*
*(, FieldDeclarators)**
| *FieldModifier** *FieldDeclaratorsWithType*
*(, FieldDeclaratorsWithType)**

FieldDeclarators ::= *FieldDeclaratorsWithType*
::= *FieldDeclaratorWithInit*

FieldDeclaratorId ::= *Identifier Guard*?

FieldDeclaratorWithInit ::= *FieldDeclaratorId Init*
| *FieldDeclaratorId ResultType Init*

FieldDeclaratorsWithType ::= *FieldDeclaratorId (, FieldDeclaratorId)** *ResultType*

FieldModifier ::= *Annotation*
| `static`

Properties. Property declarations are always declared with a type and are always final (either declared `val` or by default).

Property ::= *PropertyModifier** `val` *Identifier ResultType*
| *PropertyModifier** *Identifier ResultType*

PropertyModifier ::= *Annotation*

11 Statements

This chapter describes the statements in the sequential core of X10. Statements involving concurrency and distribution are described in §14.

11.1 Empty statement

Statement ::= `;`

The empty statement `;` does nothing. It is useful when a loop header is evaluated for its side effects. For example, the following code sums the elements of an array.

```
var sum: Int = 0;
for (i: Int = 0; i < a.length; i++, sum += a[i])
    ;
```

11.2 Local variable declaration

Statement ::= *LocalVariableDeclarationStatement*
LocalVariableDeclarationStatement ::= *LocalVariableDeclaration* `;`

The syntax of local variables declarations is described in §10.

Local variables may be declared only within a block statement (§11.3). The scope of a local variable declaration is the statement itself and the subsequent statements in the block.

11.3 Block statement

Statement ::= *BlockStatement*
BlockStatement ::= `{` *Statement** `}`

A block statement consists of a sequence of statements delimited by "`{`" and "`}`". Statements are evaluated in order. The scope of local variables introduced within the block is the remainder of the block following the variable declaration.

11.4 Expression statement

Statement ::= *ExpressionStatement*
ExpressionStatement ::= *StatementExpression* `;`
StatementExpression ::= *Assignment*
| *Allocation*
| *Call*

The expression statement evaluates an expression, ignoring the result. The expression must be either an assignment, an allocation, or a call.

11.5 Labeled statement

Statement ::= *LabeledStatement*
LabeledStatement ::= *Identifier* `:` *Statement*

Statements may be labeled. The label may be used as the target of a break or continue statement. The scope of a label is the statement labeled.

11.6 Break statement

Statement ::= *BreakStatement*
BreakStatement ::= `break` *Identifier*$^?$

An unlabeled break statement exits the currently enclosing loop or switch statement.

An labeled break statement exits the enclosing loop or switch statement with the given label.

It is illegal to break out of a loop not defined in the current method, constructor, initializer, or closure.

The following code searches for an element of a two-dimensional array and breaks out of the loop when found:

```
var found: Boolean = false;
for (i: Int = 0; i < a.length; i++)
    for (j: Int = 0; j < a(i).length; j++)
        if (a(i)(j) == v) {
            found = true;
            break;
        }
```

11.7 Continue statement

Statement ::= *ContinueStatement*
ContinueStatement ::= `continue` *Identifier*?

An unlabeled continue statement branches to the top of the currently enclosing loop.

An labeled break statement branches to the top of the enclosing loop with the given label.

It is illegal to continue a loop not defined in the current method, constructor, initializer, or closure.

11.8 If statement

Statement	::=	*IfThenStatement*
	|	*IfThenElseStatement*
IfThenStatement	::=	`if` (*Expression*) *Statement*
IfThenElseStatement	::=	`if` (*Expression*) *Statement* `else` *Statement*

An if statement comes in two forms: with and without an else clause.

The if-then statement evaluates a condition expression and evaluates the consequent expression if the condition is `true`. If the condition is `false`, the if-then statement completes normally.

The if-then-else statement evaluates a condition expression and evaluates the consequent expression if the condition is `true`; otherwise, the alternative statement is evaluated.

The condition must be of type `Boolean`.

11.9 Switch statement

Statement	::=	*SwitchStatement*
SwitchStatement	::=	`switch` (*Expression*) { *Case*$^+$ }
Case	::=	`case` *Expression* : *Statement**
	|	`default` : *Statement**

A switch statement evaluates an index expression and then branches to a case whose value equal to the value of the index expression. If no such case exists, the switch branches to the `default` case, if any.

Statements in each case branch evaluated in sequence. At the end of the branch, normal control-flow falls through to the next case, if any. To prevent fall-through, a case branch may be exited using a `break` statement.

The index expression must be of type `Int`.

Case labels must be of type `Int` and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

11.10 While statement

Statement ::= *WhileStatement*
WhileStatement ::= `while (` *Expression* `)` *Statement*

A while statement evaluates a condition and executes a loop body if `true`. If the loop body completes normally (either by reaching the end or via a `continue` statement with the loop header as target), the condition is reevaluated and the loop repeats if `true`. If the condition is `false`, the loop exits.

The condition must be of type `Boolean`.

11.11 Do–while statement

Statement ::= *DoWhileStatement*
DoWhileStatement ::= `do` *Statement* `while (` *Expression* `) ;`

A do-while statement executes a loop body, and then evaluates a condition expression. If `true`, the loop repeats. Otherwise, the loop exits.

The condition must be of type `Boolean`.

11.12 For statement

Statement ::= *ForStatement*
| *EnhancedForStatement*
ForStatement ::= `for (` *ForInit*$^?$ `;` *Expression*$^?$ `;` *ForUpdate*$^?$ `)` *Statement*
ForInit ::= *StatementExpression* `(` `,` *StatementExpression* `)`*
| *LocalVariableDeclaration*
EnhancedForStatement ::= `for (` *Formal* `in` *Expression* `)` *Statement*

X10 provides two forms of for statement: a basic for statement and an enhanced for statement.

A basic for statement consists of an initializer, a condition, an iterator, and a body. First, the initializer is evaluated. The initializer may introduce local variables that are in scope throughout the for statement. An empty initializer is permitted.

Next, the condition is evaluated. If `true`, the loop body is executed; otherwise, the loop exits. The condition may be omitted, in which case the condition is considered `true`. If the loop completes normally (either by reaching the end or via a `continue` statement with the loop header as target), the iterator is evaluated and then the condition is reevaluated and the loop repeats if `true`. If the condition is `false`, the loop exits.

The condition must be of type `Boolean`. The initializer and iterator are statements, not expressions and so do not have types.

An enhanced for statement is used to iterate over a collection. If the formal parameter is of type `T`, the collection expression must be of type `Iterable[T]`. Exploded syntax may be used for the formal parameter (§10). Each iteration of the loop binds the parameter to another element of the collection. If the parameter is final, it may not be assigned within the loop body.

In a common case, the the collection is intended to be of type `Region` and the formal parameter is of type `Point`. Expressions `e` of type `Dist` and `Array` are also accepted, and treated as if they were `e.region`. If the collection is a region, the `for` statement enumerates the points in the region in canonical order.

11.13 Throw statement

Statement ::= *ThrowStatement*
ThrowStatement ::= `throw` *Expression* `;`

The `throw` statement throws an exception. The exception must be a subclass of the value class `x10.lang.Throwable`.

Example 11.13.1 The following statement checks if an index is in range and throws an exception if not.

```
if (i < 0 || i > x.length)
    throw new IndexOutOfBoundsException();
```

□

11.14 Try–catch statement

Statement	::=	*TryStatement*
TryStatement	::=	`try` *BlockStatement Catch*$^{+}$ *Finally*$^{?}$
	|	`try` *BlockStatement Catch** *Finally*
Catch	::=	`catch (` *Formal* `)` *BlockStatement*
Finally	::=	`finally` *BlockStatement*

Exceptions are handled with a `try` statement. A `try` statement consists of a `try` block, zero or more `catch` blocks, and an optional `finally` block.

First, the `try` block is evaluated. If the block throws an exception, control transfers to the first matching `catch` block, if any. A `catch` matches if the value of the exception thrown is a subclass of the `catch` block's formal parameter type.

The `finally` block, if present, is evaluated on all normal and exceptional control-flow paths from the `try` block. If the `try` block completes normally or via a `return`, a `break`, or a `continue` statement, the `finally` block is evaluated, and then control resumes at the statement following the `try` statement, at the branch target, or at the caller as appropriate. If the `try` block completes exceptionally, the `finally` block is evaluated after the matching `catch` block, if any, and then the exception is rethrown.

11.15 Return statement

Statement	::=	*ReturnStatement*
ReturnStatement	::=	`return` *Expression* `;`
	|	`return ;`

Methods and closures may return values using a return statement. If the method's return type is expliclty declared `Void`, the method may return without a value; otherwise, it must return a value of the appropriate type.

12 Expressions

X10 supports a rich expression language similar to Java's. Evaluating an expression produces a value, which may be either an instance of a value class or an instance of a reference class. Expressions may also be `void`; that is, they produce no value. Expression evaluation may have side effects: assignment to a variable, allocation, method calls, or exceptional control-flow. Evaluation is strict and is performed left to right.

12.1 Literals

X10 supports the following literal expressions:

- An 32-bit integer literal is a value of type `x10.lang.Int`.
- An 64-bit long literal is a value of type `x10.lang.Long`.
- A 32-bit floating-point literal is a value of type `x10.lang.Float`.
- A 64-bit floating-point literal is a value of type `x10.lang.Double`.
- A character literal is a value of type `x10.lang.Char`.
- A string literal is a value of type `x10.lang.String`.
- The boolean literals `true` and `false` are of type `x10.lang.Boolean`.
- The `null` literal is of the null type, a subtype of all reference types.

12.2 this

ThisExpression ::= `this`
| *ClassName* `.` `this`

The expression `this` is a final local variable containing a reference to an instance of the lexically enclosing class. It may be used only within the body of an instance method, a constructor, or in the initializer of a instance field.

Within an inner class, `this` may be qualified with the name of a lexically enclosing class. In this case, it represents an instance of that enclosing class.

The type of a `this` expression is the innermost enclosing class, or the qualifying class, constrained by the class invariant and the method guard, if any.

The `this` expression may also be used within constraints in a class or interface header (the class invariant and `extends` and `implements` clauses). Here, the type of `this` is restricted so that only properties declared in the class header itself, and specifically not any members declared in the class body or in supertypes, are accessible through `this`.

12.3 Local variables

LocalExpression ::= *Identifier*

A local variable expression consists simply of the name of the local variable.

12.4 Field access

FieldExpression ::= *Expression* `.` *Identifier*
| `super` `.` *Identifier*
| *ClassName* `.` *Identifier*
| *ClassName* `.` `super` `.` *Identifier*

A field of an object instance may be accessed with a field access expression.

The type of the access is the declared type of the field with the actual target substituted for `this` in the type. If the actual target is not a final access path (§4.4.1), an anonymous path is substituted for `this`.

The field accessed is selected from the fields and value properties of the static type of the target and its superclasses.

If the field target is given by the keyword `super`, the target's type is the superclass of the enclosing class, as constrained by the superclass's class invariant, if any.

If the field target is `null`, a `NullPointerException` is thrown.

If the field target is a class name, a static field is selected.

It is illegal to access a field that is not visible from the current context. It is illegal to access a non-static field through a static field access expression.

12.5 Closures

X10 provides first-class, typed functions, including *closures*, *operator functions*, and *method selectors*.

ClosureExpression ::= *TypeParameters*$^?$ `(` *Formals*$^?$ `)`
Guard$^?$ *ReturnType*$^?$ *Throws*$^?$ `=>` *ClosureBody*

ClosureBody ::= *Expression*
| `{` *Statement** `}`
| `{` *Statement** *Expression* `}`

Closure expressions have zero or more type parameters, zero or more formal parameters, an optional return type and optional set of exceptions throws by the closure body. The closure body has the same syntax as a method body; it may be either an expression, a block of statements, or a block terminated by an expression to return. In particular, a value may be returned from the body of the closure using a return statement (§11.15). The type of a closure is a function type (§**??**).

As with methods, a closure may declare a guard to constrain the actual parameters with which it may be invoked. The guard may refer to the type parameters, formal parameters, and any final variables in scope at the closure expression.

The body of the closure is evaluated when the closure is invoked by a call expression (§12.8), not at the closure's place in the program text.

As with methods, a closure with return type `Void` cannot have a terminating expression. If the return type is omitted, it is inferred, as described in §4.12. It is a static error if the return type cannot be inferred.

Example 12.5.1 The following method takes a function parameter and uses it to test each element of the list, returning the first matching element.

```
def find[T](f: (T) => Boolean, xs: List[T]): T = {
  for (x: T in xs)
    if (f(x)) return x;
  null
}
```

The method may be invoked thus:

```
xs: List[Int] = ...;
x: Int = find((x: Int) => x>0, xs);
```

□

As with a normal method, the closure may have a `throws` clause. It is a static error if the body of the closure throws a checked exception that is not declared in the closure's `throws` clause.

12.5.1 Closures are objects

Closures, like all first-class functions in X10 are objects (§4.6.1). As objects, the closure body may refer to `this`, which is a reference to the current function, and use it to invoke the closure recursively.

Example 12.5.2 The following method uses a closure to compute the nth Fibonacci number.

```
def fib(n: Int): Int = {
    val f = (n: Int) : Point => {
        if (n < 1) return [1,0];
        val (r,r1): Point = this(n-1);
        [r+r1,r]
    };
```

```
    val (r,r1): Point = f(n);
    r
}
```

□

12.5.2 Outer variable access

In a closure `(x1: T1, ..., xn: Tn){c} => { s }` the types T_i, the guard `c` and the body `s` may access fields of enclosing classes and local variables and type parameters declared in an outer scope.

Recall that languages such as Java require that methods may access only those local variables declared in an enclosing scope ("outer variables") which are final. This is valuable in preventing accidental races between multiple closures reading and writing the same outer variable. At the same time, it is desirable to support the following common idiom of expression:

```
def allPositive(c: Collection): Boolean {
  shared var result: Boolean = true;
  c.applyToAll((x: Int) => { if (x < 0) atomic {result=false;}});
  return result;
}
```

This motivates the following rule:

STATIC SEMANTICS RULE: In an expression `(x1: T1, ..., xn: Tn) => e`, any outer local variable accessed by `e` must be final or must be declared as `shared` (§14.10).

The closure body may refer to instances of enclosing classes using the syntax `C.this`, where `C` is the name of the enclosing class.

NOTE: The main activity may run in parallel with any closures it creates. Hence even the read of an outer variable by the body of a closure may result in a race condition. Since closures are first-class, the analysis of whether a closure may execute in parallel with the activity that created it may be difficult.

12.6 Methods selectors

A method selector expression allows a method to be used as a first-class function.

MethodSelector ::= *Primary* `.` *MethodName* `.` *TypeParameters*$^?$ `(` *Formals*$^?$ `)`
| *TypeName* `.` *MethodName* `.` *TypeParameters*$^?$ `(` *Formals*$^?$ `)`

For a type `T`, a list of types (`T`$_1$, ..., `T`$_n$), a method name `m` and an expression `e` of type `T`, `e.m.(T`$_1$`, ..., T`$_n$`)` denotes the function, if any, bound to the instance method named `m` at type `T` whose argument type is `(T`$_1$`, ..., T`$_n$`)`, with `\xcd`this" in the method body bound to the value obtained by evaluating `e`. The return type of the function is specified in the method declaration.

Thus, the method selector

`e.m.[X`$_1$`, ..., X`$_m$`](T`$_1$`, ..., T`$_n$`)`

behaves as if it were the closure

`(X`$_1$`, ..., X`$_m$`](x`$_1$`: T`$_1$`, ..., x`$_n$`: T`$_n$`) => e.m[X`$_1$`, ..., X`$_m$`](x`$_1$`, ..., x`$_n$`)`

NOTE: Because of overloading, a method name is not sufficient to uniquely identify a function for a given class (in Java-like languages). One needs the argument type information as well. The selector syntax (dot) is used to distinguish `e.m()` (a method invocation on `e` of method named `m` with no arguments) from `e.m.()` (the function bound to the method).

A static method provides a binding from a name to a function that is independent of any instance of a class; rather it is associated with the class itself. The static function selector `T.m.(T`$_1$`, ..., T`$_n$`)` denotes the function bound to the static method named `m`, with argument types `(T`$_1$`, ..., T`$_n$`) for the type \xcd`T". The return type of the function is specified by the declaration of `T.m`.

Users of a function type do not care whether a function was defined directly (using the closure syntax), or obtained via (static or instance) function selectors.

NOTE: Design note: The function selector syntax is consistent with the reinterpretation of the usual method invocation syntax `e.m(e`$_1$`,..., e`$_n$`)` into a function specifier, `e.m`, applied to a tuple of arguments `(e`$_1$`,..., e`$_n$`)`. Note that the receiver is not treated as "an extra argument" to the function. That would break the above approach.

12.7 Operator functions

Every operator (e.g., +, -, *, /, ...) has a family of functions, one for each type on which the operator is defined. The function can be selected using the "." syntax:

OperatorFunction ::= *TypeName* `.` *Operator* `(` *Formals*$^?$ `)`
| *TypeName* `.` *Operator*

If an operator has more than one arity (e.g., unary and binary `-`), the appropriate version may be selected by giving the formal parameter types. The binary version is selected by default. For example, the following equivalences hold:

```
String.+              ≡ (x: String, y: String): String => x + y
Long.-                ≡ (x: Long, y: Long): Int => x - y
Float.-(Float,Float)  ≡ (x: Float, y: Float): Int => x - y
Int.-(Int)            ≡ (x: Int): Int => -x
Boolean.&             ≡ (x: Boolean, y: Boolean): Boolean => x & y
Boolean.!             ≡ (x: Boolean): Boolean => !x
Int.<(Int,Int)        ≡ (x: Int, y: Int): Boolean => x < y
Dist.|(Place)         ≡ (d: Dist, p: Place): Dist => d | p
```

Unary and binary promotion (§12.11) is not performed when invoking these operations; instead, the operands are coerced individually via implicit coercions (§4.10), as appropriate.

Additionally, for every expression `e` of a type `T` at which a binary operator `OP` is defined, the expression `e.OP` or `e.OP(T)` represents the function defined by:

```
(x: T): T => { e OP x }
```

Primary ::= *Expr* `.` *Operator* `(` *Formals*$^?$ `)`
| *Expr* `.` *Operator*

For every expression `e` of a type `T` at which a unary operator `OP` is defined, the expression `e.OP()` represents the function defined by:

```
(): T => { OP e }
```

For example, one may write an expression that adds one to each member of a list `xs` by:

```
xs.map(1.+)
```

12.8 Calls

MethodCall	::=	*TypeName* `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
	|	`super` `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
	|	*ClassName* `.` `super` `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
Call	::=	*Primary TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
TypeArguments	::=	`[` *Type* (`,` *Type*)* `]`

A *MethodCall* may be to either `static` or to instance methods. A *Call* may to either a method or a closure. The syntax is ambiguous; the target must be type-checked to determine if it is the name of a method or if it refers to a closure.

It is a static error if a call may resolve to both a closure call or to a method call.

A closure call `e(...)` is shorthand for a method call `e.apply(...)`.

Method selection rules are similar to that of Java. For a call with no explicit type arguments, a method with no parameters is considered more specific than a method with one or more type parameters that would have to be inferred.

Type arguments may be omitted and inferred, as described in §4.12.

It is a static error if a method's *Guard* is not satisfied by the caller.

12.9 Assignment

Expression	::=	*Assignment*
Assignment	::=	*SimpleAssignment*
	|	*OpAssignment*
SimpleAssignment	::=	*LeftHandSide* = *Expression*
OpAssignment	::=	*LeftHandSide* += *Expression*
	::=	*LeftHandSide* -= *Expression*
	::=	*LeftHandSide* *= *Expression*
	::=	*LeftHandSide* /= *Expression*
	::=	*LeftHandSide* %= *Expression*
	::=	*LeftHandSide* &= *Expression*
	::=	*LeftHandSide* |= *Expression*
	::=	*LeftHandSide* ^= *Expression*
	::=	*LeftHandSide* <<= *Expression*
	::=	*LeftHandSide* >>= *Expression*
	::=	*LeftHandSide* >>>= *Expression*
LeftHandSide	::=	*Identifier*
	|	*Primary* `.` *Identifier*
	|	*Primary* `(` *Expression* `)`

The assignment expression `x = e` assigns a value given by expression `e` to a mutable variable `x`. There are three forms of assignment: `x` may be a local variable, it may be a field `y.f`, or it may be the variable obtained by evaluating the expression `a(i)`. In the last case, `a` must be an instance of a class implementing the interface `x10.lang.Settable[S,T]`, where `S` and `T` are the types of `i` and `e`, respectively.

This interface is defined thus:

```
package x10.lang;
public interface Settable[S,T] {
    def set[S,T](i: S, v: T): T;
}
```

The assignment `a(i) = e` is equivalent to the call `a.set(i, e)`.

For a binary operator *op*, the *op*-assignment expression `x` *op*`= e` evaluates `x` to a memory location, then evaluates `e`, applies the operation `x \mathit{op} e`, and assigns the result into the location computed for `x`. The expression is equivalent to `x = x` *op* `e` except that any subexpressions of `x` are evaluated only once.

12.10 Increment and decrement

The operators ++ and -- increment and decrement a variable, respectively. The variable must be non-final and of numeric type.

When the operator is prefix, the variable is incremented or decremented by 1 and the result of the expression is the new value of the variable. When the operator is postfix, the variable is incremented or decremented by 1 and the result of the expression is the old value of the variable.

The new value of the variable v is identical to the result of the expressions v+1 or v-1, as appropriate.

12.11 Numeric promotion

The unary and binary operators promote their operands as follows. Values are sign extended and converted to instances of the promoted type.

- The unary promotion of `Byte`, `Short`, `Int` is `Int`.
- The unary promotion of `Long` is `Long`.
- The unary promotion of `Float` is `Float`.
- The unary promotion of `Double` is `Double`.
- The binary promotion of two types is the greater of the unary promotion of each type according to the following order: `Int`, `Long`, `Float`, `Double`.

12.12 Unary plus and unary minus

The unary + operator applies unary numeric promotion to its operand. The operand must be of numeric type.

The unary - operator applies unary numeric promotion to its operand and then subtracts the promoted operand from `0`. The operand must be of numeric type. The type of the result is promoted type.

12.13 Bitwise complement

The unary ~ operator applies unary numeric promotion to its operand and then evaluates to the bitwise complement of the promoted operand. The operand must be of integral type. The type of the result is promoted type.

12.14 Binary arithmetic operations

The binary arithmetic operations apply binary numeric promotion to their operands. The operands must be of numeric type. The type of the result is the promoted type. The + operator adds the promoted operands. The - operator subtracts the second operand from the first. The * operator multiplies the promoted operands. The / operator divides the first operand by the second. The % operator evaluates to the remainder of the division of the first operand by the second.

Floating point operations are determined by the IEEE 754 standard. The integer / and % throw a `DivideByZeroException` if the right operand is zero.

12.15 Binary shift operations

Unary promotion is performed on each operand separately. The operands must be of integral type. The type of the result is the promoted type of the left operand.

If the promoted type of the left operand is `Int`, the right operand is masked with `0x1f` using the bitwise AND (&) operator. If the promoted type of the left operand is `Long`, the right operand is masked with `0x3f` using the bitwise AND (&) operator.

The << operator left-shifts the left operand by the number of bits given by the right operand.

The >> operator right-shifts the left operand by the number of bits given by the right operand. The result is sign extended; that is, if the right operand is k, the most significant k bits of the result are set to the most significant bit of the operand.

The >>> operator right-shifts the left operand by the number of bits given by the right operand. The result is not sign extended; that is, if the right operand is k, the most significant k bits of the result are set to `0`.

12.16 Binary bitwise operations

The binary bitwise operations apply binary numeric promotion to their operands. The operands must be of integral type. The type of the result is the promoted type. The & operator performs the bitwise AND of the promoted operands. The | operator performs the bitwise inclusive OR of the promoted operands. The ^ operator performs the bitwise exclusive OR of the promoted operands.

12.17 String concatenation

The + operator is used for string concatenation as well as addition. If either operand is of static type `x10.lang.String`, the other operand is converted to a `String`, if needed, and the two strings are concatenated.

String conversion of a non-`null` value is performed by invoking the `toString()` method of the value. If the value is `null`, the value is converted to `"null"`.

The type of the result is `String`.

12.18 Logical negation

The operand of the unary `!` operator must be of type `x10.lang.Boolean`. The type of the result is `Boolean`. If the value of the operand is `true`, the result is `false`; if if the value of the operand is `false`, the result is `true`.

12.19 Boolean logical operations

Operands of the binary boolean logical operators must be of type `Boolean`. The type of the result is `Boolean`

The & operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`.

The | operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`.

12.20 Boolean conditional operations

Operands of the binary boolean conditional operators must be of type `Boolean`. The type of the result is `Boolean`

The `&&` operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`. Unlike the logical operator `&`, if the first operand is `false`, the second operand is not evaluated.

The `||` operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`. Unlike the logical operator `||`, if the first operand is `true`, the second operand is not evaluated.

12.21 Relational operations

The relational operations apply binary numeric promotion to their operands. The operands must be of numeric type. The type of the result is `Boolean`.

The < operator evaluates to `true` if the left operand is less than the right. The <= operator evaluates to `true` if the left operand is less than or equal to the right. The > operator evaluates to `true` if the left operand is greater than the right. The >= operator evaluates to `true` if the left operand is greater than or equal to the right.

Floating point comparison is determined by the IEEE 754 standard. Thus, if either operand is NaN, the result is `false`. Negative zero and positive zero are considered to be equal. All finite values are less than positive infinity and greater than negative infinity.

12.22 Conditional expressions

ConditionalExpression ::= *Expression* `?` *Expression* `:` *Expression*

A conditional expression evaluates its first subexpression (the condition); if `true` the second subexpression (the consequent) is evaluated; otherwise, the third subexpression (the alternative) is evaluated.

The type of the condition must be `Boolean`. The type of the conditional expression is the least common ancestor (§4.9) of the types of the consequent and the alternative.

12.23 Stable equality

EqualityExpression ::= *Expression* == *Expression*
| *Expression* != *Expression*

The == and != operators provide *stable equality*; that is, the result of the equality operation is not affected by the mutable state of the program.

Two operands may be compared with the infix predicate ==. The operation evaluates to `true` if and only if no action taken by any user program can distinguish between the two operands. In more detail, the rules are as follows.

If the operands both have reference type, then the operation evaluates to `true` if both are references to the same object (even if the object has no mutable fields).

If one operand evaluates to `null` then the predicate evaluates to `true` if and only if the other operand is also `null`.

If the operands both have value type, then they must be structurally equal; that is, they must be instances of the same value class or value array data type and all their fields or components must be ==.

If one operand is of reference type and the other is of value type, the result is `false`.

If the operands both have numeric type, binary promotion (§12.11) is performed on the operands before the comparison.

The predicate != returns `true` (`false`) on two arguments if and only if the operand == returns `false` (`true`) on the same operands.

The predicates == and != may not be overridden by the programmer.

12.24 Allocation

NewExpression ::= `new` *ClassName TypeArguments*? `(` *ArgumentList*? `)` *ClassBody*?
| `new` *InterfaceName TypeArguments*? `(` *ArgumentList*? `)` *ClassBody*

An allocation expression creates a new instance of a class and invokes a constructor of the class. The expression designates the class name and passes type and value arguments to the constructor.

The allocation expression may have an optional class body. In this case, an anonymous subclass of the given class is allocated. An anonymous class allocation may

also specify a single super-interface rather than a superclass; the superclass of the anonymous class is `x10.lang.Object`.

If the class is anonymous—that is, if a class body is provided—then the constructor is selected from the superclass. The constructor to invoke is selected using the same rules as for method invocation (§12.8).

The type of an allocation expression is the return type of the constructor invoked, with appropriate substitutions of actual arguments for formal parameters, as specified in §12.8.

It is illegal to allocate an instance of an `abstract` class. It is illegal to allocate an instance of a class or to invoke a constructor that is not visible at the allocation expression.

12.25 Casts

The cast operation may be used to cast an expression to a given type:

UnaryExpression ::= *CastExpression*
CastExpression ::= *UnaryExpression* `as` *Type*

The result of this operation is a value of the given type if the cast is permissible at run time.

The `as` operation converts the value to the given type or throws `x10.lang.ClassCastException` if the value cannot be converted. Object identity need not be preserved by the conversion. Type conversion is checked according to the rules of the Java language (e.g., [5, §5.5]). For constrained types, both the base type and the constraint are checked. If the value cannot be cast to the appropriate type, a `ClassCastException` is thrown.

12.26 `instanceof`

X10 permits types to be used in an in instanceof expression to determine whether an object is an instance of the given type:

RelationalExpression ::= *RelationalExpression* `instanceof` *Type*

In the above expression, *Type* is any type including constrained types and value types. At run time, the result of this operator is `true` if the *RelationalExpression* can be coerced to *Type* without a `TypeCastException` being thrown. Otherwise the result is `false`. This determination may involve checking that the constraint, if any, associated with the type is true for the given expression.

12.27 Subtyping expressions

SubtypingExpression ::= *Expression* `<:` *Expression*
| *Expression* `:>` *Expression*
| *Expression* `==` *Expression*

The subtyping expression T_1 `<:` T_2 evaluates to `true` T_1 is a subtype of T_2.

The expression T_1 `:>` T_2 evaluates to `true` T_2 is a subtype of T_1.

The expression T_1 `==` T_2 evaluates to `true` T_1 is a subtype of T_2 and if T_2 is a subtype of T_1.

Subtyping expressions are used in subtyping constraints for generic types.

12.28 Contains expressions

ContainsExpression ::= *Expression* `in` *Expression*

The expression `p in r` tests if a value `p` is in a collection `r`; it evaluates to `r.contains(p)`. The collection `r` must be of type `Collection[T]` and the value `p` must be of type `T`.

12.29 Rail constructors

RailConstructor ::= `[` *Expressions* `]`
Expressions ::= *Expression* (`,` *Expression*)*

The rail constructor `[`$\mathtt{a}_0$`, ..., `$\mathtt{a}_{k-1}$`]` creates an instance of `ValRail` with length k where the ith element is $\mathtt{a}_i$. The element type of the array (`T`) is bound to the least common ancestor of the types of the $\mathtt{a}_i$ (§4.9).

Since arrays are subtypes of `(Point) => T`, rail constructors can be passed into the `Array` and `ValArray` constructors as initializer functions.

Rail constructors of type `ValRail[Int]` and length `n` may be implicitly converted to type `Point{rank==n}`. Rail constructors of type `ValRail[Region]` and length `n` may be implicitly converted to type `Region{rank==n}`.

13 Places

An X10 place is a repository for data and activities. Each place is to be thought of as a locality boundary: the activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote places may take orders of magnitude longer.

X10 provides a built-in value class, `x10.lang.place`; all places are instances of this class. This class is `final` in X10 v1.7.

In X10 v1.7, the set of places available to a computation is determined at the time that the program is run and remains fixed through the run of the program. The number of places available may be determined by reading (`Place.MAX_PLACES`). (This number is specified from the command line/configuration information; see associated `README` documentation.)

All scalar objects created during program execution are located in one place, though they may be referenced from other places. Aggregate objects (arrays) may be distributed across multiple places using distributions.

The set of all places in a running instance of an X10 program may be obtained through the `const` field `Place.places`. (This set may be used to define distributions, for instance, §16.3.)

The set of all places is totally ordered. The first place may be obtained by reading `Place.FIRST_PLACE`. The initial activity for an X10 computation starts in this place (§14.5). For any place, the operation `next()` returns the next place in the total order (wrapping around at the end). Further details on the methods and fields available on this class may be obtained by consulting the API documentation.

NOTE: Future versions of the language may permit user-definable places, and the ability to dynamically create places.

STATIC SEMANTICS RULE: Variables of type `Place` must be initialized and are implicitly `final`.

13.1 Place expressions

Any expression of type `Place` is called a place expression. Examples of place expressions are `this.location` (the place at which the current object lives), `here` (the place where the current activity is executing), etc.

Place expressions are used in the following contexts:

- As a target for an `async` activity or a future (§14.2).
- In a cast expression (§12.25).
- In an `instanceof` expression (§12.26).
- In stable equality comparisons, at type `Place`.

Like values of any other type, places may be passed as arguments to methods, returned from methods, stored in fields etc.

13.2 `here`

X10 supports a special indexical constant[1] `here`:

ExpressionName ::= `here`

The constant evaluates to the place at which the current activity is running. Unlike other place expressions, this constant cannot be used as the placetype of fields, since the type of a field should be independent of the activity accessing it.

Example. The code:

```
public class F {
  public def m(a: F) {
    val OldHere: place = here;
    async (a) {
      System.out.println("OldHere == here:"
                         + (OldHere == here));
```

[1] An indexical constant is one whose value depends on its context of use.

```
    }
  }
  public static void main(s: array[String]) {
    new F().m( (future(Place.FIRST_PLACE.next()) new F())() );
  }
}
```

will print out `true` iff the computation was configured to start with the number of places set to 1.

14 Activities

An X10 computation may have many concurrent *activities* "in flight" at any give time. We use the term activity to denote a dynamic execution instance of a piece of code (with references to data). An activity is intended to execute in parallel with other activities. An activity may be thought of as a very light-weight thread. In X10 v1.7, an activity may not be interrupted, suspended or resumed as the result of actions taken by any other activity.

An activity is spawned in a given place and stays in that place for its lifetime. An activity may be *running*, *blocked* on some condition or *terminated*. When the statement associated with an activity terminates normally, the activity terminates normally; when it terminates abruptly with some reason R, the activity terminates with the same reason (§14.1).

An activity may be long-running and may invoke recursive methods (thus may have a stack associated with it). On the other hand, an activity may be short-running, involving a fine-grained operation such as a single read or write.

An activity may asynchronously and in parallel launch activities at other places.

X10 distinguishes between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate locally when the activity has finished all its computation related to that statement. (For instance the creation of an asynchronous activity terminates locally when the activity has been created.) It is said to terminate globally when it has terminated locally and all activities that it may have spawned at any place (if any) have, recursively, terminated globally.

An X10 computation is initiated as a single activity from the command line. This activity is the *root activity* for the entire computation. The entire computation terminates when (and only when) this activity globally terminates. Thus X10 does not permit the creation of so called "daemon threads"—threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§14.5).

Future Extensions. *We may permit the initial activity to be a daemon activity to permit reactive computations, such as webservers, that may not terminate.*

14.1 The X10 rooted exception model

The rooted nature of X10 computations permits the definition of a *rooted* exception model. In multi-threaded programming languages there is a natural parent-child relationship between a thread and a thread that it spawns. Typically the parent thread continues execution in parallel with the child thread. Therefore the parent thread cannot serve to catch any exceptions thrown by the child thread.

The presence of a root activity permits X10 to adopt a different model. In any state of the computation, say that an activity A is *a root of* an activity B if A is an ancestor of B and A is suspended at a statement (such as the `finish` statement §14.4) awaiting the termination of B (and possibly other activities). For every X10 computation, the *root-of* relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the *root-of* tree when the activity terminates (based on the semantics of the statement being executed by the activity).[2] Thus, unlike concurrent languages such as Java, no exception is "thrown on the floor".

14.2 Spawning an activity

Asynchronous activities serve as a single abstraction for supporting a wide range of concurrency constructs such as message passing, threads, DMA, streaming, data prefetching. (In general, asynchronous operations are better suited for supporting scalability than synchronous operations.)

An activity is created by executing the statement:

[1] Note that depending on the state of the computation the activation path may traverse activities that are running, suspended or terminated.

[2] In X10 v1.7 the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

Statement	::=	*AsyncStatement*
AsyncStatement	::=	`async` *PlaceExpressionSingleList*$^?$ *Statement*
PlaceExpressionSingleList	::=	`(` *PlaceExpression* `)`
PlaceExpression	::=	*Expression*

The place expression `e` is expected to be of type `Place`, e.g., `here` or `d(p)` for some distribution `d` and point `p` (§13). If not, the compiler replaces `e` with `e.location` if `e` is of type `x10.lang.Ref`. Otherwise the compiler reports a type error.

Note specifically that the expression `a(i)` when used as a place expression may evaluate to `a(i).location`, which may not be the same place as `a.dist(i)`. The programmer must be careful to choose the right expression, appropriate for the statement. Accesses to `a(i)` within *Statement* should typically be guarded by the place expression `a.dist(i)`.

In many cases the compiler may infer the unique place at which the statement is to be executed by an analysis of the types of the variables occuring in the statement. (The place must be such that the statement can be executed safely, without generating a `BadPlaceException`.) In such cases the programmer may omit the place designator; the compiler will throw an error if it cannot determine the unique designated place.[3]

An activity A executes the statement `async (P) S` by launching a new activity B at the designated place, to execute the specified statement. The statement terminates locally as soon as B is launched. The activation path for B is that of A, augmented with information about the line number at which B was spawned. B terminates normally when S terminates normally. It terminates abruptly if S throws an (uncaught) exception. The exception is propagated to A if A is a root activity (see §14.4), otherwise through A to A's root activity. Note that while an activity is running, exceptions thrown by activities it has already generated may propagate through it up to its root activity.

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the pool of activities at the target place and will be executed in sequence or in parallel based on the local scheduler's decisions. If the programmer wishes to sequence their execution s/he must use X10 constructs, such as clocks and `finish` to obtain the desired effect. Further, the X10

[3] X10 v1.7 does not specify a particular algorithm; this will be fixed in future versions.

implementations are not required to have fair schedulers, though every implementation should make a best faith effort to ensure that every activity eventually gets a chance to make forward progress.

STATIC SEMANTICS RULE: The statement in the body of an `async` is subject to the restriction that it must be acceptable as the body of a `void` method for an anoymous inner class declared at that point in the code, which throws no checked exceptions. As such, it may reference variables in lexically enclosing scopes (including `clock` variables, §15) provided that such variables are (implicitly or explicitly) `final`.

14.3 Place changes

An activity may change place using the `at` statement or `at` expression:

Statement	::=	*AtStatement*
AtStatement	::=	`at` *PlaceExpressionSingleList Statement*
Expression	::=	*AtExpression*
AtExpression	::=	`at` *PlaceExpressionSingleList ClosureBody*

The statement `at (p) S` executes the statement `S` synchronously at place `p`. The expression `at (p) E` executes the statement `E` synchronously at place `p`, returning the result to the originating place.

14.4 Finish

The statement `finish S` converts global termination to local termination and introduces a root activity.

Statement	::=	*FinishStatement*
FinishStatement	::=	`finish` *Statement*

An activity A executes `finish S` by executing `S`. The execution of `S` may spawn other asynchronous activities (here or at other places). Uncaught exceptions thrown or propagated by any activity spawned by `S` are accumulated at `finish S`. `finish S` terminates locally when all activities spawned by `S` terminate globally (either

abruptly or normally). If S terminates normally, then `finish S` terminates normally and A continues execution with the next statement after `finish S`. If S terminates abruptly, then `finish S` terminates abruptly and throws a single exception formed from the collection of exceptions accumulated at `finish S`.

Thus a `finish S` statement serves as a collection point for uncaught exceptions generated during the execution of S.

Note that repeatedly `finish`ing a statement has no effect after the first `finish`: `finish finish S` is indistinguishable from `finish S`.

Interaction with clocks. `finish S` interacts with clocks (§15).

While executing S, an activity must not spawn any `clocked` asyncs. (Asyncs spawned during the execution of S may spawn clocked asyncs.) A `ClockUseException` is thrown if (and when) this condition is violated.

In X10 v1.7 this condition is checked dynamically; future versions of the language will introduce type qualifiers which permit this condition to be checked statically.

Future Extensions. *The semantics of* `finish S` *is conjunctive; it terminates when all the activities created during the execution of* S *(recursively) terminate. In many situations (e.g., nondeterministic search) it is natural to require a statement to terminate when any* one *of the activities it has spawned succeeds. The other activities may then be safely aborted. Future versions of the language may introduce a* `finishone S` *construct to support such speculative or nondeterministic computation.*

14.5 Initial activity

An X10 computation is initiated from the command line on the presentation of a classname C. The class must have a `public static def main(a: array[String])` method, otherwise an exception is thrown and the computation terminates. The single statement

```
finish async (place.FIRST_PLACE) {
  C.main(s);
}
```

is executed where `s` is an array of strings created from command line arguments. This single activity is the root activity for the entire computation. (See §13 for a discussion of placs.)

14.6 Foreach statements

Statement ::= *ForEachStatement*
ForEachStatement ::= `foreach (` *Formal* `in` *Expression* `)` *Statement*

The `foreach` statement is similar to the enhanced `for` statement (§11.12).

An activity executes a `foreach` statement in a similar fashion except that separate `async` activities are launched in parallel in the local place of each object returned by the iteration. The statement terminates locally when all the activities have been spawned. It never throws an exception, though exceptions thrown by the spawned activities are propagated through to the root activity.

In a common case, the the collection is intended to be of type `Region` and the formal parameter is of type `Point`. Expressions `e` of type `Dist` and `Array` are also accepted, and treated as if they were `e.region`.

14.7 Ateach statements

Statement ::= *AtEachStatement*
AtEachStatement ::= `ateach (` *Formal* `in` *Expression* `)` *Statement*

The `ateach` statement is similar to the `foreach` statement. The collection must be of type `Dist` and the formal parameter of type `Point`. Expressions `e` of type `Array` are also accepted, and treated as if they were `e.dist`. The compiler reports a type error in all other cases.

This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, `ateach(p(i1,...,ik): point in D) S` may be thought of as standing for:

```
foreach (p(i1,...,ik): point in D.region)
  async (D(p)) S
```

14.8 Futures

X10 provides syntactic support for *asynchronous expressions*, also known as futures:

Primary	::=	*FutureExpression*
FutureExpression	::=	`future` *PlaceExpressionSingleList*$^?$ *ClosureBody*

Intuitively such an expression evaluates its body asynchronously at the given place. The resulting value may be obtained from the future returned by this expression, by using the `force` operation.

In more detail, in an expression `future (Q) e`, the place expression `Q` is treated as in an `async` statement. `e` is an expression of some type `T`. `e` may reference only those variables in the enclosing lexical environment which are declared to be `final`.

If the type of `e` is `T` then the type of `future (Q) e` is `future[T]`. This type `Future[T]` is defined as if by:

```
package x10.lang;
public interface Future[T] implements () => T {
  def forced(): Boolean;
}
```

Evaluation of `future (Q) e` terminates locally with the creation of a value `f` of type `Future[T]`. This value may be stored in objects, passed as arguments to methods, returned from method invocation etc.

At any point, the method `forced` may be invoked on `f`. This method returns without blocking, with the value `true` if the asynchronous evaluation of `e` has terminated globally and with the value `false` if it has not.

`Future[T]` is a subtype of the function type `() => T`. Invoking—*forcing*—the future `f` blocks until the asynchronous evaluation of `e` has terminated globally. If the evaluation terminates successfully with value `v`, then the method invocation returns `v`. If the evaluation terminates abruptly with exception `z`, then the method throws exception `z`. Multiple invocations of the function (by this or any other activity) do not result in multiple evaluations of `e`. The results of the first evaluation are stored in the future `f` and used to respond to all queries.

Example 14.8.1 `promise: Future[T] = future (a.dist(3)) a(3);`
`value: T = promise();`

□

14.9 At expressions

Expression ::= `at (` *Expression* `)` *Expression*

An at expression evaluates an expression synchronously at a given place. The expression `at (p) e` is equivalent to `future (p) e).force()`.

14.10 Shared variables

A shared local variable is declared with the annotation `shared`. It may be thought of as being accessible by any spawned activity in its lexical scope. Final variables are implicitly shared. An implementation may consider boxing shared variables and making a reference to the boxed value available to any closures that use the variable.

14.11 Atomic blocks

Languages such as Java use low-level synchronization locks to allow multiple interacting threads to coordinate the mutation of shared data. X10 eschews locks in favor of a very simple high-level construct, the *atomic block*.

A programmer may use atomic blocks to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple activities running in the same place.

14.11.1 Unconditional atomic blocks

The simplest form of an atomic block is the *unconditional atomic block*:

Statement	::=	*AtomicStatement*
AtomicStatement	::=	`atomic` *Statement*
MethodModifier	::=	`atomic`

For the sake of efficient implementation X10 v1.7 requires that the atomic block be *analyzable*, that is, the set of locations that are read and written by the *Block-Statement* are bounded and determined statically.[4] The exact algorithm to be used by the compiler to perform this analysis will be specified in future versions of the language.

Such a statement is executed by an activity as if in a single step during which all other concurrent activities in the same place are suspended. If execution of the statement may throw an exception, it is the programmer's responsibility to wrap the atomic block within a `try/finally` clause and include undo code in the finally clause. Thus the `atomic` statement only guarantees atomicity on successful execution, not on a faulty execution.

We allow methods of an object to be annotated with `atomic`. Such a method is taken to stand for a method whose body is wrapped within an `atomic` statement.

Atomic blocks are closely related to non-blocking synchronization constructs [6], and can be used to implement non-blocking concurrent algorithms.

STATIC SEMANTICS RULE: In `atomic S`, `S` may include method calls, conditionals, etc. It may *not* include an `async` activity. It may *not* include any statement that may potentially block at runtime (e.g., `when`, `force` operations, `next` operations on clocks, `finish`).

Limitation: Not checked in the current implementation.

All locations accessed in an atomic block must reside `here` (§13.2). A `BadPlaceException` is thrown if (and when) this condition is violated.

All locations accessed in an atomic block must statically satisfy the *locality condition*: they must belong to the place of the current activity. The compiler checks for this condition by checking whether the statement could be the body of a `void` method annotated with `local` at that point in the code (§9.7.3).

Consequences. Note an important property of an (unconditonal) atomic block:

```
atomic s1; atomic s2  =  atomic s1; s2        (14.1)
```

Further, an atomic block will eventually terminate successfully or thrown an exception; it may not introduce a deadlock.

[4] A static bound is a constant that depends only on the program text, and is independent of any runtime parameters.

Example

The following class method implements a (generic) compare and swap (CAS) operation:

```
// target defined in lexically enclosing environment.
public atomic def CAS(old: Object, new: Object): Boolean {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

14.11.2 Conditional atomic blocks

Conditional atomic blocks are of the form:

Statement	::=	*WhenStatement*
WhenStatement	::=	`when (` *Expression* `)` *Statement*
	|	*WhenStatement* `or (` *Expression* `)` *Statement*

In such a statement the one or more expressions are called *guards* and must be `Boolean` expressions. The statements are the corresponding *guarded statements*. The first pair of expression and statement is called the *main clause* and the additional pairs are called *auxiliary clauses*. A statement must have a main clause and may have no auxiliary clauses.

An activity executing such a statement suspends until such time as any one of the guards is true in the current state. In that state, the statement corresponding to the first guard that is true is executed. The checking of the guards and the execution of the corresponding guarded statement is done atomically.

X10 does not guarantee that a conditional atomic block will execute if its condition holds only intermmitently. For, based on the vagaries of the scheduler, the precise instant at which a condition holds may be missed. Therefore the programmer is advised to ensure that conditions being tested by conditional atomic blocks are eventually stable, i.e., they will continue to hold until the block executes (the action in the body of the block may cause the condition to not hold any more).

RATIONALE: The guarantee provided by `wait/notify` in Java is no stronger. Indeed conditional atomic blocks may be thought of as a replacement for Java's wait/notify functionality.

We note two common abbreviations. The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends. Second, `when (c) {;}` may be abbreviated to `await(c);`—it simply indicates that the thread must await the occurrence of a certain condition before proceeding. Finally note that a `when` statement with multiple branches is behaviorally identical to a `when` statement with a single branch that checks the disjunction of the condition of each branch, and whose body contains an `if/then/else` checking each of the branch conditions.

STATIC SEMANTICS RULE: For the sake of efficient implementation certain restrictions are placed on the guards and statements in a conditional atomic block.

Guards are required not to have side-effects, not to spawn asynchronous activities and to have a statically determinable upper bound on their execution. These conditions are expected to be checked statically by the compiler.

The body of a `when` statement must satisfy the conditions for the body of an `atomic` block.

Note that this implies that guarded statements are required to be *flat*, that is, they may not contain conditional atomic blocks. (The implementation of nested conditional atomic blocks may require sophisticated operational techniques such as rollbacks.)

Sample usage. There are many ways to ensure that a guard is eventually stable. Typically the set of activities are divided into those that may enable a condition and those that are blocked on the condition. Then it is sufficient to require that the threads that may enable a condition do not disable it once it is enabled. Instead the condition may be disabled in a guarded statement guarded by the condition. This will ensure forward progress, given the weak-fairness guarantee.

Example 14.11.1 The following class shows how to implement a bounded buffer of size 1 in X10 for repeated communication between a sender and a receiver.

```
class OneBuffer {
  datum: Object = null;
  filled: Boolean = false;
  public def send(v: Object) {
```

```
      when (!filled) {
        this.datum = v;
        this.filled = true;
      }
    }
    public def receive(): Object {
      when (filled) {
        v: Object = datum;
        datum = null;
        filled = false;
        return v;
      }
    }
  }
```

□

15 Clocks

The standard library for X10, `x10.lang`, defines a final value class `Clock` intended for repeated quiescence detection of arbitrary, data-dependent collection of activities. Clocks are a generalization of *barriers*. They permit dynamically created activities to register and deregister. An activity may be registered with multiple clocks at the same time. In particular, nested clocks are permitted: an activity may create a nested clock and within one phase of the outer clock schedule activities to run to completion on the nested clock. Nevertheless, the design of clocks ensures that deadlock cannot be introduced by using clock operations, and that clock operations do not introduce any races.

This chapter describes the syntax and semantics of clocks and statements in the language that have parameters of type `Clock`.

The key invariants associated with clocks are as follows. At any stage of the computation, a clock has zero or more *registered* activities. An activity may perform operations only on those clocks it is registered with (these clocks constitute its *clock set*). An activity is registered with one or more clocks when it is created. During its lifetime the only additional clocks it is registered with are exactly those that it creates. In particular it is not possible for an activity to register itself with a clock it discovers by reading a data-structure.

An activity may perform the following operations on a clock `c`. It may *unregister* with `c` by executing `c.drop();`. After this, it may perform no further actions on `c` for its lifetime. It may *check* to see if it is unregistered on a clock. It may *register* a newly forked activity with `c`. It may *resume* the clock by executing `c.resume();`. This indicates to `c` that it has finished posting all statements it wishes to perform in the current phase. Finally, it may *block* (by executing `next;`) on all the clocks that it is registered with. (This operation implicitly `resume`'s all clocks for the activity.) It will resume from this statement only when all these clocks are ready to advance to the next phase.

A clock becomes ready to advance to the next phase when every activity registered with the clock has executed at least one `resume` operation on that clock and all statements posted for completion in the current phase have been completed.

Though clocks introduce a blocking statement (`next`) an important property of X10 is that clocks cannot introduce deadlocks. That is, the system cannot reach a quiescent state (in which no activity is progressing) from which it is unable to progress. For, before blocking each activity resumes all clocks it is registered with. Thus if a configuration were to be stuck (that is, no activity can progress) all clocks will have been resumed. But this implies that all activities blocked on `next` may continue and the configuration is not stuck. The only other possibility is that an activity may be stuck on `finish`. But the interaction rule between `finish` and clocks (§14.4) guarantees that this cannot cause a cycle in the wait-for graph. A more rigorous proof may be found in [9].

15.1 Clock operations

The special statements introduced for clock operations are listed below.

Statement	::=	*ClockedStatement*
ClockedStatement	::=	`clocked (` *ClockList* `)` *Statement*
NextStatement	::=	*next* `;`

Note that `x10.lang.Clock` provides several useful methods on clocks (e.g. `drop`).

15.1.1 Creating new clocks

Clocks are created using the nullary constructor for `x10.lang.Clock`:

```
timeSynchronizer: Clock = Clock.make();
```

Clocks are created in the place global heap and hence outlive the lifetime of the creating activity. Clocks are instances of value classes, hence may be freely copied from place to place.

The current activity is automatically registered with the newly created clock. It may deregister using the `drop` method on clocks (see the documentation of `x10.lang.Clock`). All activities are automatically deregistered from all clocks they are registered with on termination (normal or abrupt).

15.1.2 Registering new activities on clocks

The programmer may specify which clocks a new activity is to be registered with using the `clocked` clause.

An activity may transmit only those clocks that is registered with and has not quiesced on (§15.1.3). A `ClockUseException` is thrown if (and when) this condition is violated.

An activity may check that it is registered on a clock `c` by executing:

```
c.registered()
```

This call returns the `Boolean` value `true` iff the activity is registered on `c`; otherwise it returns `false`.

NOTE: X10 does not contain a "register" statement that would allow an activity to discover a clock in a datastructure and register itself on it. Therefore, while clocks may be stored in a datastructure by one activity and read from that by another, the new activity cannot "use" the clock unless it is already registered with it.

15.1.3 Resuming clocks

X10 permits *split phase* clocks. An activity may wish to indicate that it has completed whatever work it wishes to perform in the current phase of a clock `c` it is registered with, without suspending all activity. It may do so by executing the method invocation:

```
c.resume();
```

An activity may invoke this method only on a clock it is registered with, and has not yet dropped (§15.1.5). A `ClockUseException` is thrown if (and when) this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase. Otherwise execution of this statement indicates that the activity will not transmit `c` to an `async` (through a `clocked` clause), until it terminates, drops `c` or executes a `next`.

STATIC SEMANTICS RULE: The compiler should issue an error if any activity has a potentially live execution path from a `resume` statement on a clock `c` to a async spawn statement (which registers the new activity on `c`) unless the path goes through a `next` statement. (A `c.drop()` following a `c.resume()` is legal, as is `c.resume()` following a `c.resume()`.)

15.1.4 Advancing clocks

An activity may execute the statement

```
next;
```

Execution of this statement blocks until all the clocks that the activity is registered with (if any) have advanced. (The activity implicitly issues a `resume` on all clocks it is registered with before suspending.)

An X10 computation is said to be *quiescent* on a clock `c` if each activity registered with `c` has resumed `c`. Note that once a computation is quiescent on `c`, it will remain quiescent on `c` forever (unless the system takes some action), since no other activity can become registered with `c`. That is, quiescence on a clock is a *stable property*.

Once the implementation has detected quiecence on `c`, the system marks all activities registered with `c` as being able to progress on `c`. An activity blocked on `next` resumes execution once it is marked for progress by all the clocks it is registered with.

15.1.5 Dropping clocks

An activity may drop a clock by executing:

```
c.drop();
```

The activity is no longer considered registered with this clock. A `ClockUseException` is thrown if the activity has already dropped `c`.

15.1.6 Program equivalences

From the discussion above it should be clear that the following equivalences hold:

$$\texttt{c.resume(); next;} = \texttt{next;} \quad (15.1)$$

$$\texttt{c.resume(); d.resume();} = \texttt{d.resume(); c.resume();} \quad (15.2)$$

$$\texttt{c.resume(); c.resume();} = \texttt{c.resume();} \quad (15.3)$$

Note that `next; next;` is not the same as `next;`. The first will wait for clocks to advance twice, and the second once.

16 Arrays

An array is a mapping from a region (set of points) to a range data type distributed over one or more places. Multiple arrays may be declared with the same underlying distribution. The distribution underlying an array `a` may be obtained through the field `a.dist`.

16.1 Points

Arrays are indexed by points–n-dimensional tuples of integers, implemented by the value class `x10.lang.Point`. X10 specifies a simple syntax for the construction of points. A rail constructor (§12.29) of type `ValRail[Int]` of length n can be implicitly coerced to a `Point` of rank n. For example, the following code initializes `p` to a point of rank two using a rail constructor:

```
p: Point = [1,2];
```

The `Point` constructor can take a rail constructor as argument. The assignment above can be written, without implicit coercion, as:

```
p: Point = new Point([1,2]);
```

Points implement the function type `(Int) => Int`; thus, the `i`the element of a point `p` may be accessed as `p(i)`. If `i` is out of range, an `ArrayIndexOutOfBoundsException` is thrown.

16.2 Regions

A region is a set of points all having a common rank. X10 provides a built-in value class, `x10.lang.Region`, to allow the creation of new regions and to perform

operations on regions.

Each region R has a constant rank, R.rank, which is a non-negative integer. The literal [] represents the *empty region* and has rank 0.

Here are several examples of region declarations:

```
Null: Region = new Region();  // Empty 0-dimensional region
R1: Region = 1..100; // 1-dim region with extent 1..100
R2: Region = [1..100]; // same as R1
R3: Region = (0..99) * (-1..MAX_HEIGHT);
R4: Region = [0..99, -1..MAX_HEIGHT]; // same as R3
R5: Region = Region.factory.upperTriangular(N);
R6: Region = Region.factory.banded(N, K);
R7: Region = R4 && R5; // intersection of two regions
R8: Region = R4 || R5; // union of two regions
```

The expression $a_1..a_2$ is shorthand for the rectangular, rank-1 region consisting of the points $\{[a_1], \ldots, [a_2]\}$. Each subexpression of a_i must be of type Int. If a_1 is greater than a_2, the region is empty.

A region may be constructed by converting from a rail of regions or from a rail of points. Regions are typically initialized using the rail constructor syntax (§12.29) (e.g., R4 above). The region constructed from a rail of points represents the region containing just those points. The region constructed from a rail of regions represents the Cartesian product of each of the arguments.

Various built-in regions are provided through factory methods on Region. For instance:

- Region.upperTriangular(N) returns a region corresponding to the non-zero indices in an upper-triangular N x N matrix.
- Region.lowerTriangular(N) returns a region corresponding to the non-zero indices in a lower-triangular N x N matrix.
- Region.banded(N, K) returns a region corresponding to the non-zero indices in a banded N x N matrix where the width of the band is K

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of a region R=(1..2)*(1..2) are ordered as

```
(1,1), (1,2), (2,1), (2,2)
```

Sequential iteration statements such as `for` (§11.12) iterate over the points in a region in the canonical order.

A region is said to be *convex* if it is of the form `(`$\mathtt{T}_1$ `*` $\cdots$ `*` $\mathtt{T}_k$`)` for some set of enumerations $\mathtt{T}_i$. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them. (Note that `||` may produce non-convex regions from convex regions, e.g., `[1,1] || [3,3]` is a non-convex region.)

For each region `R`, the *convex closure* of `R` is the smallest convex region enclosing `R`. For each integer `i` less than `R.rank`, the term `R(i)` represents the enumeration in the `i`th dimension of the convex closure of `R`. It may be used in a type expression wherever an enumeration may be used.

16.2.1 Operations on regions

Various non side-effecting operators (i.e., pure functions) are provided on regions. These allow the programmer to express sparse as well as dense regions.

Let `R` be a region. A subset of `R` is also called a *sub-region*.

Let $\mathtt{R}_1$ and $\mathtt{R}_2$ be two regions.

$\mathtt{R}_1$ `&&` $\mathtt{R}_2$ is the intersection of $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `||` $\mathtt{R}_2$ is the union of the $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `-` $\mathtt{R}_2$ is the set difference of $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `*` $\mathtt{R}_2$ is the Cartesian product of $\mathtt{R}_1$ and $\mathtt{R}_2$, formed by pairing each point in $\mathtt{R}_1$ with every the point in $\mathtt{R}_2$. Thus, `(1..2) * (3..4)` is the region consisting of the points {`(1,3)`, `(1,4)`, `(2,3)`, `(2,4)`}.

Two regions are equal (==) if they represent the same set of points.

16.3 Distributions

A *distribution* is a mapping from a region to a set of places. X10 provides a built-in value class, `x10.lang.Dist`, to allow the creation of new distributions and to perform operations on distributions. This class is `final` in X10 v1.7; future versions of the language may permit user-definable distributions. Since

distributions play a dual role (values as well as types), variables of type `Dist` must be initialized and are implicitly `final`.

The *rank* of a distribution is the rank of the underlying region.

```
R: Region = 1..100;
D: Dist = Dist.block(R);
D: Dist = Dist.cyclic(R);
D: Dist = R -> here;
D: Dist = Dist.random(R);
```

Let `D` be a distribution. `D.region` denotes the underlying region. `D.places` is the set of places constituting the range of `D` (viewed as a function). Given a point `p`, the expression `D(p)` represents the application of `D` to `p`, that is, the place that `p` is mapped to by `D`. The evaluation of the expression `D(p)` throws an `ArrayIndexOutofBoundsException` if `p` does not lie in the underlying region.

When operated on as a distribution, a region `R` implicitly behaves as the distribution mapping each item in `R` to `here` (i.e., `R->here`, see below). Conversely, when used in a context expecting a region, a distribution `D` should be thought of as standing for `D.region`.

16.3.1 Operations returning distributions

Let `R` be a region, `Q` a set of places {`p1`, ..., `pk`} (enumerated in canonical order), and `P` a place. All the operations described below may be performed on `Dist.factory`.

Unique distribution The distribution `unique(Q)` is the unique distribution from the region `1:k` to `Q` mapping each point `i` to `pi`.

Constant distributions. The distribution `R->P` maps every point in `R` to `P`.

Block distributions. The distribution `block(R, Q)` distributes the elements of `R` (in order) over the set of places `Q` in blocks as follows. Let p equal `|R| div N` and q equal `|R| mod N`, where `N` is the size of `Q`, and `|R|` is the size of `R`. The first q places get successive blocks of size $(p + 1)$ and the remaining places get blocks of size p.

The distribution `block(R)` is the same distribution as `block(R, Place.places)`.

Cyclic distributions. The distribution `cyclic(R, Q)` distributes the points in `R` cyclically across places in `Q` in order.

The distribution `cyclic(R)` is the same distribution as `cyclic(R, Place.places)`.

Thus the distribution `cyclic(Place.MAX_PLACES)` provides a 1–1 mapping from the region `Place.MAX_PLACES` to the set of all places and is the same as the distribution `unique(Place.places)`.

Block cyclic distributions. The distribution `blockCyclic(R, N, Q)` distributes the elements of `R` cyclically over the set of places `Q` in blocks of size `N`.

Arbitrary distributions. The distribution `arbitrary(R,Q)` arbitrarily allocates points in `R` to `Q`. As above, `arbitrary(R)` is the same distribution as `arbitrary(R, Place.places)`.

Domain Restriction. If `D` is a distribution and `R` is a sub-region of `D.region`, then `D | R` represents the restriction of `D` to `R`. The compiler throws an error if it cannot determine that `R` is a sub-region of `D.region`.

Range Restriction. If `D` is a distribution and `P` a place expression, the term `D | P` denotes the sub-distribution of `D` defined over all the points in the region of `D` mapped to `P`.

Note that `D | here` does not necessarily contain adjacent points in `D.region`. For instance, if `D` is a cyclic distribution, `D | here` will typically contain points that are `P` apart, where `P` is the number of places. An implementation may find a way to still represent them in contiguous memory, e.g., using a complex arithmetic function to map from the region index to an index into the array.

16.3.2 User-defined distributions

Future versions of X10 may provide user-defined distributions, in a way that supports static reasoning.

16.3.3 Operations on distributions

A *sub-distribution* of D is any distribution E defined on some subset of the region of D, which agrees with D on all points in its region. We also say that D is a *super-distribution* of E. A distribution D_1 *is larger than* D_2 if D_1 is a super-distribution of D_2.

Let D_1 and D_2 be two distributions.

Intersection of distributions. D_1 && D_2, the intersection of D_1 and D_2, is the largest common sub-distribution of D_1 and D_2.

Asymmetric union of distributions. D_1.overlay(D_2, the asymmetric union of D_1 and D_2, is the distribution whose region is the union of the regions of D_1 and D_2, and whose value at each point p in its region is D_2(p) if p lies in D.region otherwise it is D_1(p). (D_1 provides the defaults.)

Disjoint union of distributions. D_1 || D_2, the disjoint union of D_1 and D_2, is defined only if the regions of D_1 and D_2 are disjoint. Its value is D_1.overlay(D_2) (or equivalently D_2.overlay(D_1). (It is the least super-distribution of D_1 and D_2.)

Difference of distributions. D_1 - D_2 is the largest sub-distribution of D_1 whose region is disjoint from that of D_2.

16.3.4 Example

```
def dotProduct(a: Array[T](D), b: Array[T](D)): Array[Double](D) =
  (Array.make[T]([1:D.places],
      (Point) => (Array.make[T](D | here,
                    (i): Point) => a(i)*b(i)).sum())).sum();
```

This code returns the inner product of two T vectors defined over the same (otherwise unknown) distribution. The result is the sum reduction of an array of T with one element at each place in the range of D. The value of this array at each point is the sum reduction of the array formed by multiplying the corresponding elements of a and b in the local sub-array at the current place.

16.4 Array initializer

Arrays are instantiated by invoking a factory method for the class `Array`.

An array creation operation may also specify an initializer function. The function is applied in parallel at all points in the domain of the distribution. The array construction operation terminates locally only when the array has been fully created and initialized (at all places in the range of the distribution).

For instance:

```
val data : Array[Int]
    = Array.make[Int](1000->here, Point(i) => i);
val data2 : Array[Int]
    = Array.make[Int]([1:1000,1:1000]->here, Point(i,j) => i*j);
```

The first declaration stores in `data` a reference to a array with `1000` elements each of which is located in the same place as the array. Each array component is initialized to `i`.

The second declaration stores in `data2` a 2-d array over `[1:1000, 1:1000]` initialized with `i*j` at point `[i,j]`. It uses a more abbreviated form to specify the array initializer function.

Other examples:

```
val data : Array[Int]
    = Array.make[Int](1000, ((i): Point) => i*i);
val d : Array[Float](D)
    = Array.make[Float](D, ((i): Point) => 10.0*i);
val result : Array[Float](D)
    = Array.make[Float](D, ((i,j): Point) => i+j);
```

16.5 Operations on arrays

In the following let `a` be an array with distribution `D` and base type `T`. `a` may be mutable or immutable, unless indicated otherwise.

16.5.1 Element operations

The value of `a` at a point `p` in its region of definition is obtained by using the indexing operation `a(p)`. This operation may be used on the left hand side of an assignment operation to update the value. The operator assignments `a(i) op= e` are also available in X10.

For array variables, the right-hand-side of an assignment must have the same distribution `D` as an array being assigned. This assignment involves control communication between the sites hosting `D`. Each site performs the assignment(s) of array components locally. The assignment terminates when assignment has terminated at all sites hosting `D`.

16.5.2 Constant promotion

For a distribution `D` and a constant or final variable `v` of type `T` the expression `Array.make[T](D, (p: Point) => v)` denotes the mutable array with distribution `D` and base type `T` initialized with `v` at every point.

16.5.3 Restriction of an array

Let `D1` be a sub-distribution of `D`. Then `a | D1` represents the sub-array of `a` with the distribution `D1`.

Recall that a rich set of operators are available on distributions (§16.3) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

16.5.4 Assembling an array

Let `a1,a2` be arrays of the same base type `T` defined over distributions `D1` and `D2` respectively. Assume that both arrays are value or reference arrays.

Assembling arrays over disjoint regions If `D1` and `D2` are disjoint then the expression `a1 || a2` denotes the unique array of base type `T` defined over the distribution `D1 || D2` such that its value at point `p` is `a1(p)` if `p` lies in `D1` and `a2(p)` otherwise. This array is a reference (value) array if `a1` is.

Overlaying an array on another The expression `a1.overlay(a2)` (read: the array `a1` *overlaid with* `a2`) represents an array whose underlying region is the union of that of `a1` and `a2` and whose distribution maps each point `p` in this region to `D2(p)` if that is defined and to `D1(p)` otherwise. The value `a1.overlay(a2)(p)` is `a2(p)` if it is defined and `a1(p)` otherwise.

This array is a reference (value) array if `a1` is.

The expression `a1.update(a2)` updates the array `a1` in place with the result of `a1.overlay(a2)`.

16.5.5 Global operations

Pointwise operations The unary `lift` operation applies a function to each element of an array, returning a new array with the same distribution. The `lift` operation is implemented by the following method in `Array[T]`:

```
def lift[S](f: (T) => S): Array[S](dist);
```

The binary `lift` operation takes a binary function and another array over the same distribution and applies the function pointwise to corresponding elements of the two arrays, returning a new array with the same distribution. The `lift` operation is implemented by the following method in `Array[T]`:

```
def lift[S,R](f: (T,S) => R, Array[S](dist)): Array[R](dist);
```

Reductions Let `f` be a function of type `(T,T)=>T`. Let `a` be a value or reference array over base type `T`. Let `unit` be a value of type `T`. Then the operation `a.reduce(f, unit)` returns a value of type `T` obtained by performing `f` on all points in `a` in some order, and in parallel. The function `f` must be associative and commutative. The value `unit` should satisfy `f(unit,x)` == `x` == `f(x,unit)`.

This operation involves communication between the places over which the array is distributed. The X10 implementation guarantees that only one value of type `T` is communicated from a place as part of this reduction process.

Scans Let `f` be a reduction operator defined on type `T`. Let `a` be a value or reference array over base type `T` and distribution `D`. Then the operation `a||f()` returns an array of base type `T` and distribution `D` whose ith element (in canonical

order) is obtained by performing the reduction `f` on the first i elements of `a` (in canonical order).

This operation involves communication between the places over which the array is distributed. The X10 implementation will endeavour to minimize the communication between places to implement this operation.

Other operations on arrays may be found in `x10.lang.Array` and other related classes.

17 Annotations and compiler plugins

X10 provides an an annotation system and compiler plugin system for to allow the compiler to be extended with new static analyses and new transformations.

Annotations are interface types that decorate the abstract syntax tree of an X10 program. The X10 type-checker ensures that an annotation is a legal interface type. In X10, interfaces may declare both methods and properties. Therefore, like any interface type, an annotation may instantiate one or more of its interface's properties. Unlike with Java annotations, property initializers need not be compile-time constants; however, a given compiler plugin may do additional checks to constrain the allowable initializer expressions. The X10 type-checker does not check that all properties of an annotation are initialized, although this could be enforced by a compiler plugin.

17.1 Annotation syntax

The annotation syntax consists of an "`@`" followed by an interface type.

Annotation ::= `@` *InterfaceBaseType Constraints*$^{?}$

Annotations can be applied to most syntactic constructs in the language including class declarations, constructors, methods, field declarations, local variable declarations and formal parameters, statements, expressions, and types. Multiple occurrences of the same annotation (i.e., multiple annotations with the same interface type) on the same entity are permitted.

ClassModifier	::=	*Annotation*
InterfaceModifier	::=	*Annotation*
FieldModifier	::=	*Annotation*
MethodModifier	::=	*Annotation*
VariableModifier	::=	*Annotation*
ConstructorModifier	::=	*Annotation*
AbstractMethodModifier	::=	*Annotation*
ConstantModifier	::=	*Annotation*
Type	::=	*AnnotatedType*
AnnotatedType	::=	$Annotation^{+}$ *Type*
Statement	::=	*AnnotatedStatement*
AnnotatedStatement	::=	$Annotation^{+}$ *Statement*
Expression	::=	*AnnotatedExpression*
AnnotatedExpression	::=	$Annotation^{+}$ *Expression*

Recall that interface types may have dependent parameters.

The following examples illustrate the syntax:

- Declaration annotations:

```
// class annotation
@Value
class Cons { ... }

// method annotation
@PreCondition(0 <= i && i < this.size)
public def get(i: Int): Object { ... }

// constructor annotation
@Where(x != null)
def this(x: T) { ... }

// constructor return type annotation
def this(x: T): C@Initialized { ... }

// variable annotation
@Unique x: A;
```

- Type annotations:

```
List@Nonempty

Int@Range(1,4)

Array[Array[Double]]@Size(n * n)
```

- Expression annotations:

```
m() : @RemoteCall
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(0)
@MaxIterations(n)
for (i: Int = 0; i < n; i++) { ... }

// An annotated empty statement ;
@Assert(x < y);
```

17.2 Annotation declarations

Annotations are declared as interfaces. They must be subtypes of the interface `x10.lang.annotation.Annotation`. Annotations on types, expressions, statements, classes, fields, methods, constructors, and local variable declarations (or formal parameters) must extend `ExpressionAnnotation`, `StatementAnnotation`, `ClassAnnotation`, `FieldAnnotation`, `MethodAnnotation`, `ConstructorAnnotation`, and `VariableAnnotation`, respectively.

17.3 Compiler plugins

After the base X10 semantic checking is completed, compiler plugins are loaded and run. Plugins may perform any number of compiler passes to implement additional semantic checking and code transformations, including transformations

using the abstract syntax of the annotations themselves. Plugins should output valid X10 abstract syntax trees.

Plugins are implemented in Java as Polyglot [8] passes applied to the AST after normal base X10 type checking. Plugins to run are specified on the command-line. The order of execution is determined by the Polyglot pass scheduler.

To run compiler plugins, add the command-line option:

```
-PLUGINS=P1,P2,...,Pn
```

where `P1`, `P2`, ..., `Pn` are classes that implement the `CompilerPlugin` interface:

```
package polyglot.ext.x10.plugin;

import polyglot.ext.x10.ExtensionInfo;
import polyglot.frontend.Job;
import polyglot.frontend.goals.Goal;

public interface CompilerPlugin {
  public Goal
    register(ExtensionInfo extInfo, Job job);
}
```

The `Goal` object returned by the `register` method specifies dependencies on other passes. Documentation for Polyglot can be found at:

```
http://www.cs.cornell.edu/Projects/polyglot
```

Most plugins should implement either `SimpleOnePassPlugin` or `SimpleVisitorPlugin`.

The compiler loads plugin classes from the x10c classpath.

Plugins are given access to a Polyglot AST and type system. Annotations are represented in the AST as `Node`s with the following interface:

```
package polyglot.ext.x10.ast;

public interface AnnotationNode extends Node {
  X10ClassType annotation();
}
```

Annotations for a `Node` object `n` can be accessed through the node's extension object as follows:

```
List<AnnotationNode> annotations =
  ((X10Ext) n.ext()).annotations();
List<X10ClassType> annotationTypes =
  ((X10Ext) n.ext()).annotationInterfaces();
```

In the type system, `X10TypeObject` has the following method for accessing annotations:

```
List<X10ClassType> annotations();
```

18 Linking with native code

On some platforms, X10 v1.7 supports a simple facility to permit the efficient intra-thread communication of an array of primitive type to code written in the language C. The array must be a "local" array. The primary intent of this design is to permit the reuse of native code that efficiently implements some numeric array/matrix calculation.

Future language releases are expected to support similar bindings to FORTRAN, and to support parallel native processing of distributed X10 arrays.

The interface consists of two parts. First, an array intended to be communicated to native code must be created as an unsafe array:

```
Array.makeUnsafe[T](dist)
```

Unsafe arrays can be of any dimension. However, X10 v1.7 requires that unsafe arrays be of a primitive type, and local (i.e., with an underlying distribution that maps all elements in its region to `here`).

Unsafe arrays are allocated in a special array of memory that permits their efficient transmission to natively linked code.

Second, the X10 programmer may specify that certain methods are to be implemented natively by using the modifier `extern`:

MethodModifier ::= `extern`

Such a method must be declared `static` and may not have a method body.[1] Primitive types in the method argument are translated to their corresponding JNI type (e.g., `Float` is translated to `jfloat`, `Double` to `jdouble`, etc.). The only non-primitive type permitted in an `extern` method is an unsafe array. This is passed at type `jlong` as an eight-byte address into the unsafe region that contains the data for the array. Note that `jlong` is not the same as `long` on 32-bit machines.

[1] This restriction is likely to be lifted in the future.

Since only the starting address of an array is passed, if the array is multidimensional, the user must explicitly communicate (or have a guarantee of) the rank of the passed array, and must either typecast or explicitly code the address calculation. Note that all X10 arrays are created in row-major order, and so any native routine must also access them in the same order.

For each class `C` that contains an `extern` method, the X10 compiler generates a text file `C_x10stub.c`. This file contains generated `C` stub functions that are called from the `extern` routines. The name of the stub function is derived from the name of the `extern` method. If the method is `C.process()`, the stub function will be `Java_C_C_process()`. The name is suffixed with the signature of the method if the method is overloaded.

The programmer must write C code to implement the native method, using the methods in the `C` stub file to call the actual native method. The programmer must compile these files and link them into a dynamically linked library (DLL). Note that the `jni.h` header file must be in the include path. The programmer must ensure this library is loaded by the program before the method is called, e.g., by adding a `x10.lang.System.loadLibrary` call (in a static initializer of the X10 class).

Example 18.0.1 The following class illustrates the use of `unsafe` and native linking.

```
public class IntArrayExternUnsafe {
  public static extern
      def process(yy: unsafe Rail[Int], size: Int);
  static { System.loadLibrary("IntArrayExternUnsafe"); }
  public static def main(args: Rail[String]) {
     val b = (new IntArrayExternUnsafe()).run();
     System.out.println("++++++ Test "
                        +(b?"succeeded.":"failed."));
     System.exit(b?0:1);
  }
  public def run() : Boolean {
    val high = 10;
    val d = (0..high) -> here;
    val y: Array[Int] = Array.makeUnsafe[Int](d);
    for (val (j) in y.region) {
        y(j) = j;
```

```
    process(y,high);
    for (val (j) in y.region) {
      val expected = j+100;
      if(y(j) != expected) {
        System.out.println("y("+j+")="
                           +y(j)+" != "+expected);
        return false;
       }
    }
    return true;
  }
}
```

The programmer may then write the C code thus:

```
void IntArrayExternUnsafe_process(jlong yy, signed int size) {
  int i;
  int* array = (int*) (long) yy;
  for (i = 0; i < size; i++) {
    array[i] += 100;
  }
}
/* automatically generated in C_x10stub.c */
void
Java_IntArrayExternUnsafe_IntArrayExternUnsafe_process
(JNIEnv *env, jobject obj, jlong yy, jint size) {
   IntArrayExternUnsafe_process(yy, size);
}
```

This code may be linked with the stub file (or textually placed in it). The programmer must then compile and link the C code and ensure that the DLL is on the appropriate classpath.

□

References

[1] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[2] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.

[3] William Carlson, Tarek El-Ghazawi, Bob Numrich, and Kathy Yelick. Programming in the Partitioned Global Address Space Model, 2003. Presentation at SC 2003, http://www.gwu.edu/ upc/tutorials.html.

[4] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[5] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[7] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[8] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in LNCS, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[9] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur '05, to appear*, 2005.

[10] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[11] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

The X10 language has been developed as part of the IBM PERCS Project, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.