Report on the Programming Language X10

Version 2.0.3

DRAFT — April 30, 2010

Vijay Saraswat and Bard Bloom

Please send comments to bardb@us.ibm.com

April 30, 2010

This report provides a description of the programming language X10. X10 is a class-based object-oriented 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 Bard Bloom, Ganesh Bikshandi, David Cunningham, Robert Fuhrer, David Grove, Sreedhar Kodali, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Olivier Tardieu.

Past members include Shivali Agarwal, David Bacon, Raj Barik, Bob Blainey, Philippe Charles, Perry Cheng, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Patrick Gallop, Christian Grothoff, Allan Kielstra, Sriram Krishnamoorthy, Bruce Lucas, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon, Sayantan Sur, Christoph von Praun, Pradeep Varma, Krishna Nandivada Venkata, 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, Filip Pizlo, Ram Rajamony, R. K. Shyamasundar, 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 1.7 of the Report, released in September 2008. It documents the language corresponding to Version 2.0 of the implementation. Version 1.7 of the report was co-authored by Nathaniel Nystrom. The design of structs in X10 was led by Olivier Tardieu and Nathaniel Nystrom.

Earlier implementations benefited from significant contributions by Raj Barik, Philippe Charles, David Cunningham, 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 require 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* (PGAS) 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 PGAS 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 *struct 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 capabilities for supporting operator overloading are also provided.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the PGAS model with *asynchrony* (yielding the *APGAS* programming model). 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 v2.0 builds on v1.7 to support the following features: *structs* (i.e., "headerless", inlinable objects), type rules for preventing escape of `this` from a constructor, the introduction of a global object model, permitting user-specified (immutable) fields to be replicated with the object reference. `value` classes are no longer supported; their functionality is accomplished by using structs or global fields and methods.

Several representative idioms for concurrency and communication 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 *struct* 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 *container-based* object-oriented language similar to Java and C++, and more recent language such as Scala. Programmers write X10 code by defining containers for data and behavior called *interfaces* (§8), *classes* (§9) and *structs* (§10). X10 provides inheritance and subtyping in fairly traditional ways.

Example 2.1.1 `Normed` describes entities with a `norm()` method. `Normed` is intended to be used for entities with a position in some coordinate system, and `norm()` gives the distance between the entity and the origin. A `Slider` is an object which can be moved around on a line; a `PlanePoint` is a fixed position in a plane. Both `Slider`s and `PlanePoint`s have a sensible `norm()` method, and implement `Normed`.

```
interface Normed {
  def norm():Double;
}
class Slider implements Normed {
  var x : Double = 0;
  public def norm() = Math.abs(x);
  public def move(dx:Double) { x += dx; }
```

```
}
struct PlanePoint implements Normed {
  val x : Double, y:Double;
  public def this(x:Double, y:Double) {
    this.x = x; this.y = y;
  }
  public def norm() = Math.sqrt(x*x+y*y);
}
```

□

Interfaces An X10 interface specifies a collection of abstract methods; `Normed` specifies just `norm()`. Classes and structs can be specified to *implement* interfaces, as `Slider` and `PlanePoint` implement `Normed` , and, when they do so, must provide all the methods that the interface demands.

Interfaces are purely abstract. Every value of type `Normed` must be an instance of some class like `Slider` or some struct like `PlanePoint` which implements `Normed`; no value can be `Normed` and nothing else.

Classes and Structs There are two kinds of concrete containers: *classes* (§9) and *structs* (§10). Concrete containers hold data in *fields*, and give concrete implementations of methods, as `Slider` and `PlainPoint` above.

Classes are organized in a single-inheritance tree: a class may have only a single parent class, though it may implement many interfaces and have many subclasses. Classes may have mutable fields, as `Slider` does.

In contrast, structs are headerless values, lacking the internal organs which give objects their intricate behavior. This makes them less powerful than objects (*e.g.*, structs cannot inherit methods, though objects can), but also cheaper (*e.g.*, they can be inlined, and they require less space than objects). Structs are immutable, though their fields may be immutably set to objects which are themselves mutable. They behave like objects in all ways consistent with these limitations; *e.g.*, while they cannot *inherit* methods, they can have them – as `PlanePoint` does.

X10 has no primitive classes per se. However, the standard library `x10.lang` supplies structs `Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Complex` and `String`. The user may defined additional arithmetic structs using the facilities of the language.

Functions. X10 provides functions (§11) to allow code to be used as values. Functions are first-class data: they can be stored in lists, passed between activities, and so on. `square`, below, is a function which squares an `Int`. `of4` takes an `Int`-to-`Int` function and applies it to the number `4`. So, `fourSquared` computes `of4(square)`, which is `square(4)`, which is 16, in a fairly complicated way.

```
val square = (i:Int) => i*i;
val of4 = (f: (Int)=>Int) => f(4);
val fourSquared = of4(square);
```

They are used extensively in X10 programs. For example, the normal way to construct a `Rail[Int]` – that is, a fixed-length array of numbers, like an `int[]` in Java – is to pass two arguments to a factory method: the first argument being the length of the rail, and the second being a function which computes the initial value of the $\texttt{i}^{th}$ element. The following code constructs a rail initialized to the squares of 0,1,...,9: `r(0) == 0`, `r(5)==25`, etc.

```
val r : Rail[Int] = Rail.make[Int](10, square);
```

Constrained Types X10 containers may declare *properties*, which are fields bound immutably at the creation of the container. The static analysis system understands properties, and can work with them logically.

For example, an implementation of matrices `Mat` might have the numbers of rows and columns as properties. A little bit of care in definitions allows the definition of a + operation that works on matrices of the same shape, and * that works on matrices with appropriately matching shapes The following code typechecks, but an attempt to compute `axb1 + bxc` or `bxc * axb1` would result in a compile-time type error:

```
static def example(a:Int, b:Int, c:Int) {
  val axb1 : Mat(a,b) = makeMat(a,b);
  val axb2 : Mat(a,b) = makeMat(a,b);
  val bxc  : Mat(b,c) = makeMat(b,c);
  val axc  : Mat(a,c) = (axb1 +axb2) * bxc;
}
```

The "little bit of care" shows off many of the features of constrained types. The `(rows:Int, cols:Int)` in the class definition declares two properties, `rows`

and `cols`.[1]

A constrained type looks like `Mat{self.rows==r && self.cols==c}`: a type name, followed by a Boolean expression in braces. The special variable `self` refers to the matrix whose number of rows and columns is being checked. The `type` declaration on the second line makes `Mat(2,3)` be a synonym for `Mat{self.rows==r && self.cols==c}`, allowing for compact types in many places.

Functions can return constrained types. The `makeMat(r,c)` method returns a `Mat(r,c)` – a matrix whose shape is given by the arguments to the method. For the sake of brevity in the example, it returns `null`; in real code, it would actually produce a matrix – which must be statically provable to have the right shape. In particular, constructors can have constrained return types to provide specific information about the constructed values.

The arguments of methods can have type constraints as well. The `operator this +` line lets `A+B` add two matrices. The type of the second argument `y` is constrained to have the same number of rows and columns as the first argument `this`. Attempts to add mismatched matrices will be flagged as type errors at compilation.

At times it is more convenient to put the constraint on the method as a whole, as seen in the `operator this *` line. Unlike for +, there is no need to constrain both dimensions; we simply need to check that the columns of the left factor match the rows of the right. This constraint is written in `{...}` after the argument list. The shape of the result is computed from the shapes of the arguments.

And that is all that is necessary for a user-defined class of matrices to have shape-checking for matrix addition and multiplication. The `example` method compiles under those definitions.

```
abstract class Mat(rows:Int, cols:Int) {
 static type Mat(r:Int, c:Int) = Mat{self.rows==r&&self.cols==c};
 static def makeMat(r:Int,c:Int) : Mat(r,c) = null;
 abstract global operator this + (y:Mat(this.rows,this.cols))
                :Mat(this.rows, this.cols);
 abstract global operator this * (y:Mat) {this.cols == y.rows}
                :Mat(this.rows, y.cols);
```

[1]The class is officially declared abstract to allow for multiple implementations, like sparse and band matrices, but in fact is abstract to avoid having to write the actual definitions of + and *.

Generic types Containers may have type parameters, permitting the definition of *generic types*. Type parameters may be instantiated by any X10 type. It is thus possible to make a list of integers `List[Int]`, a list of non-zero integers `List[Int{self != 0}]`, or a list of people `List[Person]`. In the definition of `List`, `T` is a type parameter; it can be instantiated with any type.

```
class List[T] {
    var head: T;
    var tail: List[T]!;
    def this(h: T, t: List[T]!) { head = h; tail = t; }
    def add(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[List[String]]` is the type of lists whose elements are themselves lists of string, and so on.

2.2 The sequential core of X10

The sequential aspects of X10 are mostly familiar from C and its progeny. X10 enjoys the familiar control flow constructs: `if` statements, `while` loops, `for` loops, `switch` statements, `throw` to raise exceptions and `try...catch` to handle them, and so on.

X10 has both implicit coercions and explicit conversions, and both can be defined on user-defined types. Explicit conversions are written with the `as` operation: `n as Int`. The types can be constrained: `n as Int{self != 0}` converts `n` to a non-zero integer, and throws a runtime exception if its value as an integer is zero.

2.3 Places and activities

The full power of X10 starts to emerge with concurrency. 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* (§14). A place can be thought of as 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*(§15). 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, thought different aggregates may be distributed differently. Objects are garbage-collected when no longer useable; 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 it is running. It may atomically update one or more data items, but only in the current place. If it becomes necessary to read or modify an object at some other place `q`, the *place-shifting* operation `at(q)` can be used, to move part of the activity to `q`. It is easy to compute across multiple places, but the expensive operations (*e.g.*, those which require communication) are readily visible in the code.

Object and Places. Every object has a *home place*, `x.home`. `home` is a property, and the constrained type mechanisms mentioned previously can be used with it. For example, an object `x` with a mutable field `f` only allows mutations when `x.home==here`, which formalizes the concept above. If we know that `x` is located here but the typechecker doesn't, we can use a cast to tell it:

```
val xhere = x as FHolder!;
xhere.f = 12;
```

(`FHolder!` is an abbreviation for `FHolder{self.home==here}`, a `FHolder` object located here.) This will throw an exception if `x` is not located here.

Atomic blocks. X10 has a control construct `atomic S` where `S` is a statement with certain restrictions. `S` will be executed atomically, without interruption by other activities. This is a common primitive used in concurrent algorithms, though rarely provided in this degree of generality by concurrent programming languages.

More powerfully – and more expensively – X10 allows conditional atomic blocks, `when(B)S`, which are executed atomically at some point when `B` is true. Conditional atomic blocks are one of the strongest primitives used in concurrent algorithms, and one of the least-often available.

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 `P` executing statement `S`.

If the activity needs to return a value, the form `future (p) E` is often convenient. This spawns an activity at `p` evaluating `E`, and also returns a value called a *future* which is capable of accepting the value of `E` when it is ready. The caller can try to get the value from the future, which provides it immediately if it is ready and blocks to wait for it if it is not.

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 heterogeneous activities in an X10 computation. X10 allows multiple barriers in a form that supports determinate, deadlock-free parallel computation, via the `Clock` type.

A single `Clock` represents a computation that occurs in phases. At any given time, an activity is *registered* with zero or more clocks. The X10 statement `next` tells all of an activity's registered clocks that the activity has finished the current phase, and causes it to wait for the next phase. Other operations allow waiting on a single clock, starting new clocks or new activities registered on an extant clock, and so on.

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.

2.5 Arrays, regions and distributions

X10 provides `DistArray`s, *distributed arrays*, which spread data across many places. An underlying `Dist` object provides the *distribution*, telling which elements of the `DistArray` go in which place. `Dist` uses subsidiary `Region` objects to abstract over the shape and even the dimensionality of arrays. Specialized X10 control statements such as `ateach` provide efficient parallel iteration over distributed arrays.

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, by encouraging working with local data, and limiting the ability of events at one place to delay those at another. For example, 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 whose 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* (§19). 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.[2] 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.

[2] 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`

`global`			
`goto`	`has`	`here`	`if`
`implements`	`import`	`instanceof`	`interface`
`native`	`new`	`next`	`nonblocking`
`or`	`package`	`pinned`	`private`
`protected`	`property`	`public`	`return`
`safe`	`self`	`sequential`	`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 Briefly, X10 v2.0 uses fairly standard syntax for its literals: integers, unsigned integers, floating point numbers, booleans, characters, strings, and `null`. The most exotic points are (1) unsigned numbers are marked by a `u` and cannot have a sign; (2) `true` and `false` are the literals for the booleans; and (3) floating point numbers are `Double` unless marked with an `f` for `Float`.

Less briefly, we use the following abbreviations:

$$
\begin{array}{rcl}
\delta & = & \text{one or more decimal digits} \\
\delta_8 & = & \text{one or more octal digits} \\
\delta_{16} & = & \text{one or more hexadecimal digits, using } \texttt{a-f} \text{ for 10-15} \\
\iota & = & \delta \mid \texttt{0}\delta_8 \mid \texttt{0x}\delta_{16} \mid \texttt{0X}\delta_{16} \\
\sigma & = & \text{optional + or -} \\
\beta & = & \delta \mid \delta. \mid \delta.\delta \mid .\delta \\
\xi & = & (\texttt{e} \mid \texttt{E})\sigma\delta \\
\phi & = & \beta\xi
\end{array}
$$

- `true` and `false` are the `Boolean` literals.
- `null` is a literal for the null value. It has type `Any{self==null}`.
- `Int` literals have the form $\sigma\iota$; *e.g.*, `123`, `-321` are decimal `Int`s, `0123` and `-0321` are octal `Int`s, and `0x123`, `-0X321`, `0xBED`, and `0XEBEC` are hexadecimal `Int`s.

- `Long` literals have the form $\sigma\iota$`l` or $\sigma\iota$`L`. *E.g.*, `1234567890L` and `0xBAGEL` are `Long` literals.
- `UInt` literals have the form ι`u` or ι`U`. *E.g.*, `123u`, `0123u`, and `0xBEAU` are `UInt` literals.
- `ULong` literals have the form ι`ul` or ι`lu`, or capital versions of those. For example, `123ul`, `0124567012ul`, `0xFLU`, `OXba1eful`, and `0xDecafC0ffeefUL` are `ULong` literals.
- `Float` literals have the form $\sigma\phi$`f` or $\sigma\phi$`F`. Note that the floating-point marker letter `f` is required: unmarked floating-point-looking literals are `Double`. *E.g.*, `1f`, `6.023E+32f`, `6.626068E-34F` are `Float` literals.
- `Double` literals have the form $\sigma\phi$[1], $\sigma\phi$`D`, and $\sigma\phi$`d`. *E.g.*, `0.0`, `0e100`, `229792458d`, and `314159265e-8` are `Double` literals.
- `Char` literals have one of the following forms:
  - `'`*c*`'` where *c* is any printing ASCII character other than \ or ', representing the character *c* itself; *e.g.*, `'!'`;
  - `'\b'`, representing backspace;
  - `'\t'`, representing tab;
  - `'\n'`, representing newline;
  - `'\f'`, representing form feed;
  - `'\r'`, representing return;
  - `'\''`, representing single-quote;
  - `'\"'`, representing double-quote;
  - `'\\'`, representing backslash;
  - `'\`*dd*`'`, where *dd* is one or more octal digits, representing the one-byte character numbered *dd*; it is an error if *dd*> 255.
- `String` literals consist of a double-quote ", followed by zero or more of the contents of a `Char` literal, followed by another double quote. *E.g.*, `"hi!"`, `""`.
- There are no literals of type `Byte` or `UByte`.

[1] Except that literals like `1` which match both ι and ϕ are counted as integers, not `Double`; `Double`s require a decimal point, an exponent, or the `d` marker.

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. Types limit the values that variables can hold and specify the places at which these values lie.

X10 supports three kinds of runtime entities, *objects*, *structs*, and *functions*. Objects are instances of *classes* (§9). They may contain mutable fields and must stay resident in the place in which they were created. Objects are said to be *boxed* in that variables of a class type are implemented through a single memory location that contains a reference to the memory containing the declared state of the object (and other meta-information such as the list of methods of the object). Thus objects are represented through an extra level of indirection. A consequence of this flexibility is that every class type contains the value `null` corresponding to the invalid reference. `null` is often useful as a default value. Further, two objects may be compared for equality (==) in constant time by simply containing references to the memory used to represent the objects.

Structs are instances of *struct types* (§10). They are immutable and may be freely copied from place to place. Further, they may be allocated inline, *i.e.*, using only as much memory as necessary to hold the fields of the struct (and any additional memory necessary to satisfy alignment constraints for data layout).

Functions are instances of *function types*— §11) and are created using function literals `(x1:T1,...,xn:Tn){c}:T => e`. Functions contain no user-visible mutable or immutable state; their representation, however, contains enough memory to hold the values of the variables in the environment that are referenced in the body of the function (`e`). Functions may be freely copied from place to place and may be repeatedly applied to a set of arguments, provided that the precondition `c` is satisfied.

These runtime entities are classified into different groups using *types*. Types are used in variable declarations, explicit coercions and conversions, object creation,

array creation, class literals, static state and method accessors, and `instanceof` expressions.[1]

X10 has a unified type system. The top of the type hierarchy is the interface `x10.lang.Any`. This specifies the following signature:

```
package x10.lang;
public interface Any {
  property def home():Place;
  property def at(p:Object):Boolean;
  property def at(p:Place):Boolean;
  global safe def toString():String;
  global safe def typeName():String;
  global safe def equals(Any):Boolean;
  global safe def hashCode():Int;
}
```

Properties are described in (§4.1.3): in brief they are immutable instance fields of objects and structs that can be used to construct types through constraints. Property methods (§4.1.3) permit more complex expressions to be defined using properties and other property methods. A method is `safe` if it has certain behavioral characteristics §9.6.3. It is `global` if it can be invoked at any place.

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).
- An interface declaration defines an *interface type* (§4.1.2).
- Classes and interface have *type parameters*. A class, interface, or struct or with one or more type parameters is a *generic class*, *generic interface*, or *generic struct*. (§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).

[1] In order to allow this version of the language to focus on the core new ideas, X10 v2.0 does not have user-definable class loaders, though there is no technical reason why they could not have been added.

- *Function type* constructors are used to define function types; functions and method selectors have function type (§4.5).
- A *constrained type* constrains the properties of a base type (§4.4).
- Types may be marked with user-defined annotations. *Annotated types* (§4.6) 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 ad-

dition, 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 classes extend the class `x10.lang.Object`. Classes are declared to extend a single superclass (except for `Object`, which extends no other class).

Variables of class type may contain the value `null`.

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 can also have static members: constant fields, type definitions, and member classes and interfaces.

An interface may extend multiple interfaces.

Classes may be declared to implement multiple interfaces. Semantically, the interface type is the set of all objects that are instances of 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 Properties

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

package not.x10.lang;

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

The properties of a class or interface may be constrained with a boolean expression. The type `Coords{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, §9.6,§11.2). 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. (Structs and interfaces may be generic as well.) 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; }
}
%~~siv
%~~neg
```

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

A covariant type parameter is permitted to appear only in covariant type positions, and a contravariant type parameter in contravariant positions.

- The return type of a method is a covariant position.
- The argument types of a method are contravariant positions.
- Whether a type argument position of a generic class, interface or struct type `C` is covariant or contravariant is determined by the `+` or `-` annotation at that position in the declaration of `C`.

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 contravariant, 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

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, given `import x10.util.*`:

```
type StringSet = Set[String];
type MapToList[K,V] = Map[K,List[V]];
type Int(x: Int) = Int{self==x};
type Dist(r: Int) = Dist{self.rank==r};
type Dist(r: Region) = Dist{self.region==r};
type Redund(n:Int, r:Region){r.rank==n} = Dist{rank==n && region==r};
```

As the two definitions of `Dist` 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 (§9.6.2), define distinct type constructors.

Type definitions may appear as (static) 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, 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[T1, ..., Tk]` is a class type, a constructor of `C` may be invoked with `new A(e1, ..., en)`, if the invocation `new C[T1, ..., Tk](e1, ..., en)` is legal and if the constructor return type is a subtype of `A`.

The collection of type definitions in `x10.lang.\_` is automatically imported in every compilation unit.

4.4 Constrained types

Given a type `T`, a *constrained type* `T{e}` may be constructed by constraining its properties with a boolean expression of a limited sort `e`. The values of `T{e}` are those values of type `T` for which `e` evaluates to `true`. *E.g.*, `Point` has a property `rank:Int`. If `p : Point`, `p` may have any `rank`. `Point{rank == 3}` is the point type constrained to only those values whose `rank` property is 3.

A common use of constrained types is to explain where objects are located. Every object has a `home` property. If `Person` is a type of people, then `Person{home==here}` is the type of people whose data is stored at the current location. As explained in §14, certain operations can only be performed at an object's home, so having this expressible as a type is crucial.

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

Constraints may refer to values in the local environment:

```
val n = 1;
var p : Point{rank == n};
```

Indeed, there is technically no need for a constraint to refer to the properties of its type; it can refer entirely to the environment, thus:

```
val m = 1;
val n = 2;
var p : Point{m != n};
```

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.

4.4.1 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*

In X10 v2.0 value constraints may be equalities (==), disequalities (!=) and conjunctions thereof. The terms over which these constraints are specified include literals and (accessible, immutable) variables and fields, property methods, and the special constants `here`, `self`, and `this`. Additionally, place types are permitted (§4.4.2).

When constraining a value of type `T`, `self` refers to the object of type `T` which is being constrained. For example, `Int{self == 4}` is the type of `Int`s which are equal to 4 – the best possible description of `4`, and a very difficult type to express without using `self`.

Type constraints may be subtyping and supertyping (`<:` and `:>`) expressions over types.

The static constraint checker approximates computational reality in some cases. For example, it assumes that built-in types are infinite. This is a good approximation for `Int`. It is a poor approximation for `Boolean`, as the checker believes that `a != b && a != c && b != c` is satisfiable over `Boolean`, which it is not. However, the checker is always correct when computing the truth or falsehood of a constraint.

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.

Acyclicity restriction

To ensure that type-checking is decidable, we require that property graphs be acyclic. That is, it should not be the case at runtime that a set of objects can be created such that the graph formed by taking objects as nodes and adding an edge from m to n if m has a property whose value is n has a cycle in it.

Currently this restriction is not checked by the compiler. Future versions of the compiler will check this restriction by introducing rules on escaping of `this` (§9.4) before the invocation of `property` calls.

4.4.2 Place constraints

An X10 computation spans multiple places (§14). Each place constains data and activities that operate on that data. X10 v2.0 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 ::= *Expression*

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

All X10 classes extend the class `x10.lang.Object`, which defines a property `home` of type `Place`.

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

The place `p` may be ommitted. It defaults to `this` for types in field declarations, and to `here` elsewhere.

4.4.3 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)

local `val`s, `val` method parameters or `val` constructor parameters visible at `T`, (d) `val` 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 `this` may be used in an expression at that point. *Viz.*, if `this` occurs in (a) a property declaration for a class, (b) an instance method, (c) an instance field, or (d) 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}`.

4.4.4 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 $S = \{\texttt{f}_1\texttt{: F}_1\texttt{, } \ldots\texttt{, f}_k\texttt{: F}_k\texttt{, f}_{k+1}\texttt{: F}_{k+1}\texttt{, } \ldots\texttt{, f}_n\texttt{: F}_n\}$ 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` (§9.1). 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 4.4.1 A class that represents a line has two distinct points:[2]

[2]We call them `Position` to avoid confusion with the built-in class `Point`

```
class Position(x: Int, y: Int) {
   def this(x:Int,y:Int){property(x,y);}
   }
class Line(start: Position,
           end: Position{self != start}) {}
```

□

Example 4.4.2 One can use dependent type to define other closed geometric figures as well.

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

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

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

Example 4.4.3 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})
 {
   def this(a:Line,
            b: Line{a.end == b.start},
            c: Line{b.end == c.start && c.end == a.start})
   {property(a,b,c);}
 }
```

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.

□

4.5 Function types

Function types are defined via the => type constructor. Closures (§11) and method selectors (§11.3) are of function type. The general form of a function type is:

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

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

The 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 may be omitted if they are not used. The guard specifies a condition that must hold for an application to be well-typed.

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

For every sequence of types `T1,..., Tn,T`, and `n` distinct variables `x1,...,xn` and constraint `c`, the expression `(x1:T1,...,xn:Tn){c}=>T` is a *function type*. It stands for the set of all functions `f` which can be applied in a place `p` to a list of values `(v1,...,vn)` provided that the constraint `c[v1,...,vn,p/x1,...,xn,here]` is true, and which returns a value of type `T[v1,...vn/x1,...,xn]`. When `c` is true, the clause `{c}` can be omitted. When `x1,...,xn` do not occur in `c` or `T`, they can be omitted. Thus the type `(T1,...,Tn)=>T` is actually shorthand for `(x1:T1,...,xn:Tn){true}=>T`, for some variables `x1,...,xn`.

Juxtaposition is used to express function application: the expression `f(a1,..,an)` expresses the application of a function `f` to the argument list `a1,...,an`.

Note that function invocation may throw unchecked exceptions.

A function type is covariant in its result type and contravariant in each of its argument types. That is, let `S1,...,Sn,S,T1,...Tn,T` be any types satisfying `Si <: Ti` and `S <: T`. Then `(x1:T1,...,xn:Tn){c}=>S` is a subtype of `(x1:S1,...,xn:Sn){c}=>T`.

A value `f` of a function type `(x1:T1,...,xn:Tn){c}=>T` also has all the methods of `Any` associated with it (see §11.5).

A function type `F=(x1:T1,...,xn:Tn){c}=>T` can be used as the declared type of local variables, parameters, loop variables, return types of methods and in `_ instanceof F` and `_ as F` expressions.

A class or struct definition may use a function type `F` in its implements clause; this declares an abstract method `def apply(x1:T1,...,xn:Tn){c}:T` on that

class. Similarly, an interface definition may specify a function type "F" in its "extends" clause. A class or struct implementing such an interface implicitly defines an abstract method `def apply(x1:T1,..,xn:Tn){c}:T`. Expressions of such a struct, class or interface type can be assigned to variables of type `F` and can be applied via juxtaposition to an argument list of the right type.

Thus, objects and structs in X10 may behave like functions.

A function type `F` is not a class type in that it does not extend any type or implement any interfaces, or support equality tests. `F` cannot be extended by any type. It is not an interface type in that it is not a subtype of `x10.lang.Object`. (Values of type `F` cannot be assigned to variables of type `x10.lang.Object`.) It is not a struct type in that it has no defined fields and hence no notion of structural equality.

`null` is a legal value for a function type.

4.6 Annotated types

Any X10 type may be annotated with zero or more user-defined *type annotations* (§18).

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 interface types $\texttt{A}_1, \ldots, \texttt{A}_n$ using the syntax $\texttt{@A}_1 \ \ldots \ \texttt{@A}_n$ `T`.

4.7 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.

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`. When this subtyping relationship holds, $[\![\texttt{C\{c\}}]\!]$ 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}`.

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.

4.8 Least common ancestor of types

To compute the type of conditional expressions (§12.20), and of rail constructors (§12.27), 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.Any`.

4.9 Coercions and conversions

X10 v2.0 supports the following coercions and conversions

4.9.1 Coercions

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast. A *conversion* may change object identity if the type being converted to is not the same as the type converted from. X10 permits user-defined conversions (§4.9.2).

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 `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

Effects of explicit numeric coercion Coercing a number of one type to another type gives the best approximation of the number in the result type, or a suitable disaster value if no approximation is good enough.

- Casting a number to a *wider* numeric type is safe and effective, and can be done by an implicit conversion as well as an explicit coercion. For example, `4 as Long` produces the `Long` value of 4.
- Casting a floating-point value to an integer value truncates the digits after the decimal point, thereby rounding the number towards zero. `54.321 as Int` is `54`, and `-54.321 as Int` is `-54`. If the floating-point value is too large to represent as that kind of integer, the coercion returns the largest or smallest value of that type instead: `1e110 as Int` is `Int.MAX_VALUE`, `2147483647`.

- Casting a `Double` to a `Float` normally truncates digits: `0.12345678901234567890 as Float` is `0.12345679f`. This can turn a nonzero `Double` into `0.0f`, the zero of type `Float`: `1e-100 as Float` is `0.0f`. Since `Double`s can be as large as about `1.79E308` and `Float`s can only be as large as about `3.4E38f`, a large `Double` will be converted to the special `Float` value of `Infinity`: `1e100 as Float` is `Infinity`.

- Integers are coerced to smaller integer types by truncating the high-order bits. If the value of the large integer fits into the smaller integer's range, this gives the same number in the smaller type: `12 as Byte` is the `Byte`-sized 12, `-12 as Byte` is -12. However, if the larger integer *doesn't* fit in the smaller type, the numeric value and even the sign can change: `254 as Byte` is `Byte`-sized `-2`.

4.9.2 Conversions

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
```

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.

User defined conversions. The user may define conversion operators from type `A` *to* a container type `B` by specifying a method on `B` as follows:

```
public static operator (r: A): T = ...
```

The return type `T` should be a subtype of `B`. The return type need not be specified explicitly; it will be computed in the usual fashion if it is not. However, it is good practice for the programmer to specify the return type for such operators explicitly.

For instance, the code for `x10.lang.Point` contains:

```
public static global safe operator (r: Rail[int])
   : Point(r.length) = make(r);
```

The compiler looks for such operators on the container type `B` when it encounters an expression of the form `r as B` (where `r` is of type `A`). If it finds such a method, it sets the type of the expression `r as B` to be the return type of the method. Thus the type of `r as B` is guaranteed to be some subtype of `B`.

Example 4.9.1 Consider the following code:

```
val p  = [2, 2, 2, 2, 2] as Point;
val q = [1, 1, 1, 1, 1] as Point;
val a = p - q;
```

This code fragment compiles successfully, given the above operator definition. The type of `p` is inferred to be `Point(5)` (i.e. the type `Point{self.rank==5}`. Similarly for `q`. Hence the application of the operator "-" is legal (it requires both arguments to have the same rank). The type of `a` is computed as `Point(5)`. □

4.10 Built-in types

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

4.10.1 The class `Object`

The class `x10.lang.Object` is the supertype of all classes. A variable of this type can hold a reference to any object. The code for this class (with annotations removed) is:

```
public class Object (home: Place)
     implements Any
{
    public native def this();
    public property def home() = home;
    public property def at(p:Place) = home==p;
    public property def at(r:Object) = home==r.home;
```

```
    public global safe native def toString() : String;
    public global safe native def typeName() : String;
    public global safe def equals(x:Any) = this == x;
    public global safe native def hashCode():Int;
}
```

4.10.2 The class `String`

Strings in X10 are instances of the class `x10.lang.String`, and are all immutable. Strings are one of the few types with literals, rather than simply constructors. String literals are the familiar "-delimited strings, like `"this"` and `"that"`.

Every X10 value has a `String` print representation, given by `whatever.toString()`. All values can be implicitly converted to strings by the concatenation operation +, which calls their `toString()` methods if they are not strings already. For example, `"one " + 2 + here` evaluates to something like `one 2(Place 0)`.

4.10.3 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 v2.0 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};
```

4.10.4 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 class ValRail[T](length: Int) implements (Int)=>T, Iterable[T] { }
public class Rail[T](length: Int) implements (Int)=>T, Settable[Int,T], Itera
```

X10 supports shorthand syntax for rail construction (§12.27).

4.10.5 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;
}
```

4.11 Type inference

X10 v2.0 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.

4.11.1 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.

4.11.2 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 is 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`.

4.11.3 Type arguments

A call to a polymorphic method 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]`, `List[T] <: Collection[T]`, and `SubClass <: SuperClass`, in the following snippet, the algorithm will infer the type `Collection[Any]` for `x`.

```
def m(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`.

```
def m(intSet: Set[Int], intList: List[Int]) {
  val y = choose(intSet, intList);
  ...
}
```

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

```
def m(intSet: Set[SubClass], numList: List{T <: SuperClass}) {
  val z = choose(intSet, numList);
  ...
}
```

5 Variables

A *variable* is an X10 identifier associated with a value within some context. Variable bindings have these essential properties:

- **Type:** What sorts of values can be bound to the identifier;
- **Scope:** The region of code in which the identifier is associated with the entity;
- **Lifetime:** The interval of time in which the identifier is associated with the entity.
- **Visibility:** Which parts of the program can read or manipulate the value through the variable.

X10 has many varieties of variables, used for a number of purposes. They will be described in more detail in this chapter.

- Class variables, also known as the static fields of a class, which hold their values for the lifetime of the class.
- Instance variables, which hold their values for the lifetime of an object;
- Array elements, which are not individually named and hold their values for the lifetime of an array;
- Formal parameters to methods, functions, and constructors, which hold their values for the duration of method (etc.) invocation;
- Local variables, which hold their values for the duration of execution of a block.

- Exception-handler parameters, which hold their values for the execution of the exception being handled.

A few other kinds of things are called variables for historical reasons; *e.g.*, type parameters are often called type variables, despite not being variables in this sense because they do not refer to X10 values. Other named entities, such as classes and methods, are not called variables. However, all name-to-whatever bindings enjoy similar concepts of scope and visibility.

In the following example, `n` is an instance variable, and `next` is a local variable defined within the method `bump`.[1]

```
class Counter {
  private var n : Int = 0;
  public def bump() : Int {
    val next = n+1;
    n = next;
    return next;
    }
}
```

Both variables have type `Int` (or perhaps something more specific). The scope of `n` is the body of `Counter`; the scope of `next` is the body of `bump`. The lifetime of `n` is the lifetime of the `Counter` object holding it; the lifetime of `next` is the duration of the call to `bump`. Neither variable can be seen from outside of its scope.

Variables whose value may not be changed after initialization are said to be *immutable*, or *constants* (§5.1), or simply `val` variables. Variables whose value may change are *mutable* or simply `var` variables. `var` variables are declared by the `var` keyword. `val` variables may be declared by the `val` keyword; when a variable declaration does not include either `var` or `val`, it is considered `val`.

```
val a : Int = 0;                // Full 'val' syntax
b : Int = 0;                    // 'val' implied
val c = 0;                      // Type inferred
var d : Int = 0;                // Full 'var' syntax
var e : Int;                    // Not initialized
var f : Int{self != 100} = 0;   // Constrained type
```

[1] This code is unnecessarily turgid for the sake of the example. One would generally write `public def bump() = ++n;`.

5.1 Immutable variables

Immutable variables can be given values (by initialization or assignment) at most once, and must be given values before they are used. Usually this is achieved by declaring and initializing the variable in a single statement.

```
val a : Int = 10;
val b = (a+1)*(a-1);
```

`a` and `b` cannot be assigned to further.

In other cases, the declaration and assignment are separate. One such case is how constructors give values to `val` fields of objects. The `Example` class has an immutable field `n`, which is given different values depending on which constructor was called. `n` can't be given its value by initialization when it is declared, since it is not knowable which constructor is called at that point.

```
class Example {
  val n : Int; // not initialized here
  def this() { n = 1; }
  def this(dummy:Boolean) { n = 2;}
}
```

Another common case of separating declaration and assignment is in function and method call. The formal parameters are bound to the corresponding actual parameters, but the binding does not happen until the function is called. In the code below, `x` is initialized to `3` in the first call and `4` in the second.

```
val sq = (x:Int) => x*x;
x10.io.Console.OUT.println("3 squared = " + sq(3));
x10.io.Console.OUT.println("4 squared = " + sq(4));
```

5.2 Initial values of variables

Every assignment, binding, or initialization to a variable of type `T{c}` must be an instance of type `T` satisfying the constraint `{c}`. Variables must be given a value before they are used. This may be done by initialization, which is the only way for immutable (`val`) variables and one option for mutable (`var`) ones:

```
val immut : Int = 3;
var mutab : Int = immut;
val use = immut + mutab;
```

Or, for mutable variables, it may be done by a later assignment.

```
var muta2 : Int;
muta2 = 4;
val use = muta2 * 10;
```

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. **(B: Revise this in light of initial values :B)** Mutable instance variables of class type are initialized to to `null`. Mutable instance variables of struct 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 of the variable's type is not assignable to the variable's type, *e.g.*, `Int` variables are initialized to zero, but that doesn't work for `val x:Int{x!=0}`.

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].

5.3 Destructuring syntax

X10 permits a *destructuring* syntax for local variable declarations and formal parameters. At present, X10 v2.0 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 `i` to `0` and `j` to `1`, and the variable `p` to the point object.

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

5.4 Formal parameters

Formal parameters are always declared with a type. 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`

`var`, `val`, and `shared` behave just as they do for local variables, §5.5.

5.5 Local variables

Local variable declarations may have initializer expressions: `var i:Int = 1;` introduces a variable `i` and initializes it to 1. The initializer must be a subtype of the declared type of the variable. If the variable is immutable (`val`) the type may be omitted and inferred from the initializer type (§4.11). Variables marked `shared` can be used by many activities at once; see §15.10.

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*

5.6 Fields

Fields are declared either `var` (mutable, non-static), `val` (immutable, non-static), or `const` (immutable, 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 immutable, the type may be omitted and inferred from the initializer type (§4.11). Mutable fields must be declared with a type.

FieldDeclaration ::= *FieldModifier** `var` *FieldDeclaratorsWithType*
*(, FieldDeclaratorsWithType)**
| *FieldModifier** `const` *FieldDeclarators*
*(, FieldDeclarators)**
| *FieldModifier** `val` *FieldDeclarators*
*(, FieldDeclarators)**
| *FieldModifier** *FieldDeclaratorsWithType*
*(, FieldDeclaratorsWithType)**
FieldDeclarators ::= *FieldDeclaratorsWithType*
::= *FieldDeclaratorWithInit*
FieldDeclaratorId ::= *Identifier*
FieldDeclaratorWithInit ::= *FieldDeclaratorId Init*
| *FieldDeclaratorId ResultType Init*
FieldDeclaratorsWithType ::= *FieldDeclaratorId (, FieldDeclaratorId)** *ResultType*
FieldModifier ::= *Annotation*
| `static`

5.7 Properties

Property declarations are always declared with a type and are always immutable (either explicitly declared `val` or implicitly by default).

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

6 Objects

6.1 Basic Design

An object is an instance of a scalar class or an array type. It is created by using an allocation expression (§12.22) or an array creation (§17.4) expression, such as an array initializer.

All classes subclass from `x10.lang.Object`. This class has one property `home` of type `x10.lang.Place`. Thus all objects in X10 are located (have a place).

In X10 v2.0 an object stays resident at the place at which it was created for its entire lifetime. However, the programmer may designate certain immutable field of an object as `global`. The value of these fields is accessible at every place the object can be referenced.

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 (§15.11) for mutual exclusion and clocks (§16) for sequencing multiple parallel operations.

An 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 (§15.2) `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 carries with it the values of `global val` fields of the object. The implementation should try to ensure that the cost of communicating the values of `val` 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 an object may perform this operation only if the object is local. (By contrast, an activity holding a reference to a struct may perform this operation regardless of the location of the struct, since structs 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 `val` fields discussed above.

- Method invocation (§12.6). A method may be marked `global`. A `global` method may be invoked at any place. It may access only the global fields of the object. The mutable fields of an object may be accessed only by activities operating in its home. Any activity may use an `at` statement to place-shift to the place of the object. Methods may also be marked `pinned`, `nonblocking`, `sequential`, `safe`, and `pure` (§9.6.3)..

- Casting (§12.23). 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 (§12.24). 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 equality operators == and != (§12.21). On creation, each object is associated with a globally unique identifier (guid). Two object references are == iff they refer to objects with the same guid.

X10 has a rather simple *distributed object model*.

The state of an object is partitioned into *global* state (a programmer defined subset of `val` fields, §9.5.2) and *non-global* state.

- Field definitions are marked with the qualifier `global` if they are intended to be included in the global state.
- If the `global` qualifier is omitted, the field is considered non-global.
- Properties and static fields are implicitly marked `global`.
- `var` fields cannot be marked `global`.

Similarly, the methods of an object may be qualified as `global`(§9.6.3) ; if they are not global they are said to be *non-global*. Global methods cannot be overridden by non-global methods.

Consider the execution of an `at (P) S` statement at a place `Q` different from `P`. Suppose `x` is an in-scope immutable local variable and contains a reference to an object `o` created at `Q`. Then within `S`, `x` is said to be a *remote reference* to `o`. References to `o` from place `Q` are said to be *local references*. X10 permits `global` fields to be read and `global` methods to be invoked through a remote reference.

Like local references, remote references are first-class entities: they may be passed as arguments to methods, returned from methods, stored in fields of objects.

Remote references may also be compared for equality (==). Two remote reference are equal if they are references to the same object. Equality is guaranteed to be a constant-time operation and not involve any communication.

When a remote reference to an object `o` located at place `P` is transmitted to `P` it automatically becomes a local reference to `o`. Therefore the situation in which a local reference can be compared to a remote reference simply cannot arise.

The X10 compiler ensures that non-global methods on `o` can only be invoked in a place where `here== o.home()`, i.e. the place where `o` was created.

Implementation notes Remote references to an object `o` are intended to be implemented by serializing the global state of `o` across the network, together with a globally unique id (guid). The data is deserialized at the receiver to create an implementation-level entity that is the remote reference. There is no requirement

that the implementation intern such entities; however the implementation must correctly implement equality (see below).

There is no requirement that a remote reference use only as much space as a local reference.

Local execution The semantics of `atomic` and `when` constructs requires that their bodies do not execute any `at` operations, implicitly or explicitly. Hence the compiler must establish that if a non-global method `m` is being invoked on a reference `o` in the body of such a construct, then `o` is a local reference. This can be done using place types (§4.4.2).

6.2 Examples

Assume the class declarations.

```
class C { }
class D {
   var f:C=null;
}
```

Then the following code behaves as stated.

```
val x = new C();
// C object o1 created, reference stored in x.
at (P) {
      // In the body x contains a remote reference to o
      val d = new D();
      d.f  = x; // remote reference stored in d.f
      Console.OUT.println(d.f == x);        // must print true
      Console.OUT.println(x == x);          // must print true
      at (Q) {
         // x continues to be a remote reference to o1.
         at (P) {
             Console.OUT.println(d.f == x);   // must print true
             Console.OUT.println(x == x);   // must print true
         }
```

```
        }
    }
```

Here is another example.

```
val x = new C();
// C object o created, reference stored in x.
// The type inferred for x is C!
at (P) {
    val x1 = x;
    // The type inferred for x1 is C, not C!;
    // the change is due to the place shift 'at(P)'
  at (x.home) {
        // x is now bound to o through a local reference. So is x1.
    Console.OUT.println(x1==x); // Must print true.
      // non-global methods can be invoked on x or x1 and will
      // execute locally on o
      // type of both x and x1 is C!.
  }
}
```

6.2.1 Programming Methodology

A programmer wishing to ensure that a `val` field is not serialized when the containing object is serialized (e.g. because it contains a large cache which makes sense only in the current place) must ensure the field is *not* marked global.

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*

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 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 v2.0 interfaces are essentially the same Java interfaces [5, §9]. An interface primarily specifies signatures for public methods. It may extend multiple interfaces.

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* *TypePropertyList*$^?$ *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 (§4.4.4).

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 an interface body contains an explicit definition for a method with this signature.

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, properties, a constraint (the *class invariant*), a single superclass, one or more interfaces, and a class body containing the the definition of fields, properties, 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* (`,` *TypeParameter*)*
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 property has a name and a type. Properties are accessible in the same way as `public val` fields.

STATIC SEMANTICS RULE: It is a compile-time error for a class defining a property `x: T` to have an ancestor class that defines a 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()=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 §4.4.

Properties are initialized by the invocation of a special `property` call in each constructor of the class:

```
property(e1,..., en);
```

The number and type of arguments to the `property` call must match the number and type of properties in the class declaration, in left to right lexical order. Each constructor is required to initialize its properties before normal termination.

The *Guard* in a class or interface declaration specifies an explicit condition on the properties of the type, and is discussed further in §9.1.

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.

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`/default-protected access modification framework may be used.

Class declarations may be used to construct class types (§4.1). Classes may have mutable fields. Instances of a class are always created in a fixed place and in X10 v2.0 stay there for the lifetime of the object. Variables declared at a class type always store a reference to the object, regardless of whether the object is local or remote.

9.1 Type invariants

There is a general recipe for constructing a list of parameters or properties $\texttt{x}_1\texttt{: T}_1\texttt{\{c}_1\texttt{\}, ..., x}_k\texttt{: T}_k\texttt{\{c}_k\texttt{\}}$ 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 $\texttt{x}_1\texttt{: T}_1\texttt{\{x}_2\texttt{: T}_2\texttt{; ...; x}_k\texttt{: T}_k\texttt{; c\}}$ is consistent iff $\exists\texttt{x}_1\texttt{: T}_1\texttt{, x}_2\texttt{: T}_2\texttt{, ..., x}_k\texttt{: T}_k\texttt{. c}$ is consistent. The second is consistent iff

$$\forall\texttt{x}_1\texttt{: T}_1\texttt{\{x}_2\texttt{: T}_2\texttt{; ...; x}_k\texttt{: T}_k\texttt{; c\}}$$
$$\exists\texttt{x}_2\texttt{: T}_2\texttt{. }\exists\texttt{x}_3\texttt{: T}_3\texttt{, ..., x}_k\texttt{: T}_k\texttt{. c}$$

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

Thus logically every satisfiable constraint `c` on a list of parameters $\texttt{x}_1, \ldots, \texttt{x}_k$ can be expressed using the dependent types of $\texttt{x}_i$, 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} { ... }
```

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 property list $\texttt{x}_1\texttt{: T}_1, \ldots, \texttt{x}_n\texttt{: T}_n$ for a class `C` is said to be consistent if each of the $\texttt{T}_i$ are consistent and the constraint

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

is valid (always true).

The guard is an invariant on all instances of the class or interface.

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 $\texttt{I}_1, \ldots, \texttt{I}_k$ and defines properties $\texttt{x}_1\texttt{: P}_1, \ldots, \texttt{x}_n\texttt{: P}_n$ and specifies a guard `c` is given by:

inv(I$_1$), ..., inv(I$_k$),
    self.x$_1$: P$_1$, ..., self.x$_n$: P$_n$, c

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

inv(D), inv(I$_1$), ..., inv(I$_k$),
    self.x$_1$: P$_1$, ..., self.x$_n$: P$_n$, 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(T[o/this]) $\wedge$ c[o/self].

9.2 implements and extends clauses

Consider a class definition

ClassModifiers$^?$
class C(x$_1$: P$_1$, ..., x$_n$: P$_n$) extends D{d}
   implements I$_1${c$_1$}, ..., I$_k${c$_k$}
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.

9.3 Constructor definitions

A constructor for a class C is guaranteed to return an object of the class on successful termination. This object must satisfy inv(C), the class invariant associated with C (§9.1). 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 9.3.1 Here is another example, constructed as a simplified version of `x10.lang.Region`. The `mockUnion` method has the type that a true `union` method would have.

```
class MyRegion(rank:Int) {
  static type MyRegion(n:Int)=MyRegion{self.rank==n};
  def this(r:Int):MyRegion(r) {
    property(r);
  }
  def this(diag:ValRail[Int]):MyRegion(diag.length){
    property(diag.length);
  }
  def mockUnion(r:MyRegion(rank)):MyRegion(rank) = this;
  def example() {
    val R1 : MyRegion(3)! = new MyRegion([4,4,4]);
    val R2 : MyRegion(3)! = new MyRegion([5,4,1]);
    val R3 = R1.mockUnion(R2); // inferred type MyRegion(3)
  }
}
```

The first constructor returns the empty region of rank `r`. The second constructor takes a `ValRail[Int]` of arbitrary length `n` and returns a `MyRegion(n)` (intended to represent the set of points in the rectangular parallelopiped between the origin and the `diag`.)

The code in `example` typechecks, and `R3`'s type is inferred as `MyRegion(3)`.

□

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 super($\texttt{e}_1$, ..., $\texttt{e}_k$) and the static type of each expression $\texttt{e}_i$ is $\texttt{S}_i$, and the invocation is statically resolved to a constructor def this($\texttt{x}_1$: $\texttt{T}_1$, ..., $\texttt{x}_k$: $\texttt{T}_k$){c}: D{$\texttt{d}_1$} then it must be the case that

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

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 C($\texttt{e}_1$, ..., $\texttt{e}_n$) 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.

9.4 proto qualifier on types

X10 ensures that every variable must have a value consistent with its type before it is read.

For local variables, this is ensured by using a pre-specified static analysis to ensure that every local variable is written into before it is read. Type-checking of assignment ensures the value written is consistent with the static type of the variable.

For fields, this is ensured by introducing a form of ownership types called *incomplete types* to address the *escaping-this* problem. To permit flexibility in writing constructors, X10 v1.7 permits `this` to be used in a constructor as a reference to the object currently being constructed. Unfortunately there are no restrictions on the usage of `this`. In particular, this reference can be permitted to escape: it may be stored in variables on the heap (thereby permitting concurrently executing activities to read the value of fields that may not yet have been initialized), passed as an argument to method invocations, or used as the target for a method invocation. Indeed, the method may be invoked in a super constructor, and may have been overridden at a subclass, guaranteeing that accesses to fields defined in the subclass are accesses to uninitialized variables. For instance an immutable field may be observed containing a value (the value the field was initialized with) which may be different from the value it will contain once the constructor has returned.

Incomplete types are designed with the following goals:

- Guarantee that fields are not read before they are initialized.
- Allow the creation of immutable cyclic object graphs.[1] This requires that it be possible to pass an object under construction into a constructor invocation.
- Allow appropriate user-defined methods can be called during object creation (so that the transformation between the values supplied as parameters to a constructor and the values actually placed in fields is determined by arbitrary user-defined code).
- Keep the design minimally invasive. Most programmers should not have to be concerned about this problem.
- Ensure that there is no runtime overhead.

These goals are met by introducing incomplete types through the type qualifier `proto`. Types of the form `proto T` are said to be *incomplete types*; types that do

[1] (Mutable graphs can be created without escaping `this` by initializing the backpointer to `null` and then changing it later.

not have the qualifier are said to be *complete*. Say that an object o is *confined* to a given activity A if it can be reached only from stack frames of A or from objects which are, recursively, confined to A. Thus confined objects cannot be accessed by activities other than A.

Incomplete types ensure that objects whose construtors have not exited are confined. Further, all references to such objects on the stack are contained in variables of incomplete types. The compiler does not permit the fields of variables of incomplete types to be read. Thus incomplete types permit the construction of graphs of objects while ensuring that these objects are confined and their fields are not read during construction.

The return value of a constructor for class C that takes no incomplete arguments is (a subtype of) C, that is, a complete type. It will point only to completed objects. It can now be assigned to any (type-consistent) field of any object, that is, it is now allowed to escape.

`proto` Rules

For every type T (where T is not a type variable), we introduce the type `proto T`.

There is no relationship between types T and `proto T` – neither is a subtype of the other.[2]

Incomplete types are permitted to occur only as types of method parameters or local variables or as return types for methods and constructors. They may not occur in (the source or target of) cast statements, `extends` or `implements` clauses, `catch` clauses, or as types of class fields.

Within the body of a class C the type of `this` in constructors, instance initializers and instance variable initializers is `proto C`.

Let v be a value of type `proto C`, for some class C.

No fields of v can be read. (This is the defining property of `proto` types.) However, v's (accessible) instance fields can be assigned.

[2]Clearly, a value of type `proto T` cannot be used anywhere that a T is needed, since its fields cannot be read. As discussed below, an incomplete value v can be assigned to a field f of an object o only if o is incomplete. This ensures that v cannot escape through this assignment. A completed value p cannot be substituted for o – it may permit v to escape through an assignment to its field. Therefore T cannot be a subtype of `proto T`.

`v` can be assigned to an instance field `o.f` only if `f` is of some type `S` such that `T <: S` and `o` has an incomplete type.

`v` can be assigned to local variables only if they are of some type `proto S` (such that `T <: S`).

Instance methods of class `C` may be qualified with `proto` (these methods are called *incomplete methods*). The type of `this` in incomplete methods is `proto C`. Incomplete methods can be overridden only by incomplete methods. Only incomplete methods can be invoked on `v`. Incomplete methods which do not take an argument of incomplete type can be invoked on completed values.

`v` can be passed as argument into a constructor or method call, or returned from a method. The return type of a method taking an argument at an incomplete type must be `void` or incomplete. The return type of a constructor taking an argument at a `proto` type must be incomplete.

A generic type parameter `T` can be instantiated with the type `proto S` (where `S` is not a type parameter itself), provided that the body of the entity being instantiated satisfies the conditions above for `proto S`.

During code generation, the type `proto T` is treated as if it were `T`. That is, there is no run-time cost to `proto` types.

The invariants maintained by the design are as follows. Say that an object field or stack variable (local variable) contains an incomplete value if a value of type `proto T` (for some `T`) was written into it.

- If an object `o` has a field containing an incomplete value `v`, then either `v`'s constructor has exited or `o` is confined. Further, every reference to `o` on the stack is held at an incomplete type.
- If a stack variable contains an incomplete value, then the variable's type is incomplete.

Say that a constructor invocation for a class `C` on the call stack is a *root* if it takes no incomplete arguments. Such a constructor invocation will return an object of type `C` whose fields may point to an arbitrary graph of newly created objects (objects created by the activity after the constructor invocation). Since the object returned is at type `C` – and not `proto C` – It may be assigned to any field of any object on the heap of type `D` such that `C <: D`. It is no longer confined. Thus the "magic moment" when an incomplete value becomes complete is when the last constructor for any incomplete value it references (including itself) returns.

Example

Example 9.4.1 This example shows how to create a fixed-size circular buffer. (Its pointer structure is immutable, though the contents of each field are mutable.)

```
class  CircularBuffer[A] {
  var a: A;
  val next: CircularBuffer[A];
  private def this(x: proto CircularBuffer[A]): proto CircularBuffer[A] {
    next = x;
  }
  def this(var n:Int) {
    var temp: proto CircularBuffer[A] = this;
    while (--n > 0)
      temp = new CircularBuffer[A](temp);
    next = temp;
  }
}
```

□

9.5 Field definitions

A class may have zero or more mutable or immutable fields. No two fields declared in a class may have the same name.

Fields may be marked `static`. Only one instance of such a field exists, and it may be accessed through the name of the class in which it is defined (§9.7). Fields not marked `static` are said to be *instance* fields. One copy of such a field exists for every instance of the class.

To avoid an ambiguity, it is a static error for a class to declare a field with a function type (§4.5) with the same name and signature as a method of the same class.

9.5.1 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`.

9.5.2 Field qualifiers

`global` qualifier

A field may be declared `global`.

FieldModifier ::= `global`

A `global` field must be immutable. It may be read from any place. Properties and static fields are implicitly marked `global`. Fields not marked `global` cannot be overridden by fields marked `global`.

9.6 Method definitions

X10 permits guarded method definitions.

MethodDeclaration ::= *MethodHeader* `;`
| *MethodHeader* `=` *ClosureBody*

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

A formal parameter may optionally have a `val` or `var` modifier (default: `val`). The body of the method is executed in an environment in which each formal parameter corresponds to a local variable and is initialized with the value of the actual parameter. The local variable is mutable if and only if the parameter is a `var` parameter.

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(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)` for a method is type correct. Each argument $\mathtt{e}_i$ must have a static type $\mathtt{S}_i$ that is a subtype of the declared type $\mathtt{T}_i$ 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(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)` 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. The method body is either an expression, a block of statements, or a block ending with an expression.

Example 9.6.1 The following program implements a subset of `Point`'s behavior.

```
final class Pointlike(rank: Int)  {
    public global val coords: ValRail[Int]{self.length == rank};
    public def this(v:ValRail[Int]) {
       property(v.length);
       coords = v as ValRail[Int]{self.length == rank};
    }
    public operator this + (that: Pointlike{self.rank == this.rank})
       : Pointlike{self.rank == this.rank}
       {
         val f : (Int)=>Int = (i:Int) =>
                 (this.coords)(i) + (that.coords)(i);
         val v = ValRail.make[Int](rank, f);
         return new Pointlike(v);
       }
    static def example() {
       val s <: Pointlike{self.rank==3} = new Pointlike([1,2,3]);
       val t <: Pointlike{self.rank==3} = new Pointlike([-1,-1,-1]);
       val u <: Pointlike{self.rank==3} = s + t;
    }
}
```

□

9.6.1 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 if the expression cannot be represented in the constraint system.

The expression may contain invocations of other properties. It is the responsibility of the programmer to ensure that the evaluation of a property terminates at compile-time, otherwise the type-checker will not terminate and the program will fail to compile.

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

A nullary 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;
```

and

```
property rail: Boolean = rect && onePlace == here && zeroBased;
```

Similarly, nullary property methods can be inspected in constraints without `()`. `w.rail`, with either definition above, is equivalent to `w.rail()`

9.6.2 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 formal parameters of different types. *E.g.*, the following is legal:

```
class Mful{
   def m() = 1;
   def m[T]() = 2;
   def m(x:Int) = 3;
   def m[T](x:Int) = 4;
}
```

X10 v2.0 does not permit overloading based on constraints. That is, the following is *not* legal, although either method definition individually is legal:

```
   def n(x:Int){x==1} = "one";
   def n(x:Int){x!=1} = "not";
```

The definition of a method declaration $\texttt{m}_1$ "having the same signature as" a method declaration $\texttt{m}_2$ involves identity of types.

The *constraint erasure* of a type T is defined as follows. The constraint erasure of (a) a class, interface or struct type T is T; (b) a type T{c} is the constraint erasure of T; (b) a type T[$\texttt{S}_1$,\ldots,$\texttt{S}_n$] is T'[$\texttt{S}_1$',\ldots,$\texttt{S}_n$'] where each primed type is the erasure of the corresponding unprimed type. 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 the constraint erasure of its 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, ..., vn: Tn){tc}: T {...}
def m[X1, ..., Xm](v1: S1, ..., vn: Sn){sc}: S {...}
```

if it is the case that the constraint erasures of the types $\texttt{T}_1$, ..., $\texttt{T}_n$ are equivalent to the constraint erasures of the types $\texttt{S}_1$, ..., $\texttt{T}_n$ 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 modifiers 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 o.m($\texttt{e}_1$, ..., $\texttt{e}_n$) is said to have the *static signature* <T, $\texttt{T}_1$, ..., $\texttt{T}_n$> where T is the static type of o, and $\texttt{T}_1$, ..., $\texttt{T}_n$ are the static types of $\texttt{e}_1$, ..., $\texttt{e}_n$, respectively. As in Java, it must be the case that the compiler can determine a single method defined on T with argument type $\texttt{T}_1$, ... $\texttt{T}_n$; 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.

9.6.3 Method qualifiers

`atomic` qualifier

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.

`global` qualifier

A method may be declared `global`.

MethodModifier ::= `global`

A `global` method can be invoked on an object `o` in any place. The body of such a method is type-checked without assuming that `here==this.home`. This permits `global` fields of `o` to be accessed, but not local fields. The programmer must insert an explicit `at(this)...` to get to the place where the object lives and access the field.

`global` methods can be overridden only by methods also marked `global`.

`pinned` qualifier

A method may be declared `pinned`.

MethodModifier ::= `pinned`

A `pinned` method may not contain any `at` statement or expression whose place argument is not statically equivalent to `here`. It must call only `pinned` methods. That is, a `pinned` method does not cause any communication.

`pinned` methods can be overridden only by methods marked `pinned`.

`nonblocking` qualifier

A method may be declared `nonblocking`.

MethodModifier ::= `nonblocking`

A `nonblocking` method may not contain any `when` statement whose condition is not statically equivalent to `true`. It must call only `nonblocking` methods. That is, a `nonblocking` method does not block.

`nonblocking` methods can be overridden only by methods marked `nonblocking`.

`sequential` qualifier

A method may be declared `sequential`.

MethodModifier ::= `sequential`

A `sequential` method may not contain any `async` statement. It must call only `sequential` methods. That is, a `sequential` method does not spawn any activity.

`sequential` methods can be overridden only by methods marked `sequential`.

`safe` qualifier

A method may be declared `safe`.

MethodModifier ::= `safe`

The `safe` annotation is considered shorthand for `pinned nonblocking sequential`.

9.7 Static initialization

The X10 runtime implements the following procedure to ensure reliable initialization of the static state of classes.

Execution commences with a single thread executing the *initialization* phase of an X10 computation at place `0`. This phase must complete successfully before the body of the `main` method is executed.

The initialization phase must be thought of as if it is implemented in the following fashion: (The implementation may do something more efficient as long as it is faithful to this semantics.)

```
Within the scope of a new finish
for every static field f of every class C
```

```
   (with type T and initializer e):
async {
  val l = e;
  ateach (Dist.makeUnique()) {
     assign l to the static f field of
         the local C class object;
     mark the f field of the local C
         class object as initialized;
  }
}
```

During this phase, any read of a static field `C.f` (where `f` is of type `T`) is replaced by a call to the method `C.read_f():T` defined on class `C` as follows

```
def read_f():T {
   await (initialized(C.f));
   return C.f;
}
```

If all these activities terminate normally, all static field have a legal value (per their type), and the finish terminates normally. If any activity throws an exception, the finish throws an exception. Since no user code is executing which can catch exceptions thrown by the finish, the exceptions are printed on the console, and computation aborts.

If the activities deadlock, the implementation deadlocks.

In all cases, the main method is executed only once all static fields have been initialized correctly.

Since static state is immutable and is replicated to all places via the initialization phase as described above, it can be accessed from any place.

9.8 User-Defined Operators

It is often convenient to have methods named by symbols rather than words. For example, suppose that we wish to define a `Poly` class of polynomials – for the sake of illustration, single-variable polynomials with `Int` coefficients. It would be very nice to be able to manipulate these polynomials by the usual operations: + to add,

* to multiply, - to subtract, and p(x) to compute the value of the polynomial at argument x. We would like to write code thus:

```
public static def main(Rail[String]) {
   val X = new Poly([0,1]);
   val t <: Poly = 7 * X + 6 * X * X * X;
   val u <: Poly = 3 + 5*X - 7*X*X;
   val v <: Poly = t * u - 1;
   for ( (i) in -3 .. 3) {
     x10.io.Console.OUT.println(
       "" + i + " X:" + X(i) + "  t:" + t(i)
       + "    u:" + u(i) + "  v:" + v(i)
       );
   }
}
```

Writing the same code with method calls, while possible, is far less elegant:

```
public static def uglymain() {
   val X = new UglyPoly([0,1]);
   val t <: UglyPoly = X.mult(7).plus(X.mult(X).mult(X).mult(6));
   val u <: UglyPoly = const(3).plus(X.mult(5)).minus(X.mult(X).mult(7));
   val v <: UglyPoly = t.mult(u).minus(1);
   for ( (i) in -3 .. 3) {
     x10.io.Console.OUT.println(
       "" + i + " X:" + X.apply(i) + "    t:" + t.apply(i)
        + "   u:" + u.apply(i) + "    v:" + v.apply(i)
       );
   }
}
```

The operator-using code can be written in X10, though a few variations are necessary to handle such exotic cases as 1+X.

9.8.1 Binary Operators

Defining the sum P+Q of two polynomials looks much like a method definition. It uses the `operator` keyword instead of `def`, and `this` appears in the definition in the place that a `Poly` would appear in a use of the operator. So, `operator this + (p:Poly!)` explains how to add `this` to a `Poly!` value.

```
class Poly {
  public global val coeff : ValRail[Int];
  public def this(coeff: ValRail[Int]) { this.coeff = coeff;}
  public global def degree() = coeff.length()-1;
  public global def  a(i:Int) = (i<0 || i>this.degree()) ? 0 : coeff(i);

  public operator this + (p:Poly!) =  new Poly(
     ValRail.make[Int](
        Math.max(this.coeff.length(), p.coeff.length()),
        (i:Int) => this.a(i) + p.a(i)
     ));
  // ...
```

The sum of a polynomial and an integer, `P+3`, looks like an overloaded method definition.

```
public operator (n : Int) + this = new Poly([n]) + this;
```

However, we want to allow the sum of an integer and a polynomial as well: `3+P`. It would be quite inconvenient to have to define this as a method on `Int`; changing `Int` is far outside of normal coding. So, we allow it as a method on `Poly` as well.

```
public operator this + (n : Int) = new Poly([n]) + this;
```

Furthermore, it is sometimes convenient to express a binary operation as a static method on a class. The definition for the sum of two `Poly`s could have been written:

```
public static operator (p:Poly!) + (q:Poly!) =  new Poly(
   ValRail.make[Int](
      Math.max(q.coeff.length(), p.coeff.length()),
      (i:Int) => q.a(i) + p.a(i)
   ));
```

This requires the following syntax:

MethodHeader ::= `operator` *TypeParameterList*$^?$ `this` *BinOp* `(` *FormalParameter* `)` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

MethodHeader ::= `operator` *TypeParameterList*$^?$ `(` *FormalParameter* `)` *BinOp* `this` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

MethodHeader ::= `operator` *TypeParameterList*$^?$ `(` *FormalParameter* `)` *BinOp* `(` *FormalParameter* *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

When X10 attempts to typecheck a binary operator expression like `P+Q`, it first typechecks `P` and `Q`. Then, it looks for operator declarations for + in the types of `P` and `Q`. If there are none, it is a static error. If there is precisely one, that one will be used. If there are several, X10 looks for a *best-matching* operation, *viz.* one which does not require the operands to be converted to another type. For example, `operator this + (n:Long)` and `operator this + (n:Int)` both apply to `p+1`, because 1 can be converted from an `Int` to a `Long`. However, the `Int` version will be chosen because it does not require a conversion. If even the best-matching operation is not uniquely determined, the compiler will report a static error.

The main difference between expressing a binary operation as an instance method (with a `this` in the definition) and a static one (no `this`) is that instance methods don't apply any conversions, while static methods attempt to convert both arguments. **(B: give an example :B)**

(B: List the operators which this works for, in precedence order :B)

9.8.2 Unary Operators

Unary operators are defined in a similar way, with `this` appearing in the `operator` definition where an actual value would occur in a unary expression. The operator to negate a polynomial is:

```
public operator - this = new Poly(
  ValRail.make[Int](coeff.length(), (i:Int) => -coeff(i))
  );
```

The syntax for unary operators is:

MethodHeader ::= `operator` *PrefixOp* `this` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

The rules for typechecking a unary operation are the same as for methods; the complexities of binary operations are not needed.

(B: List the operators which this works for, in precedence order :B)

9.8.3 Type Conversions

Explicit type conversions, `e as T{c}`, can be defined as operators on class `T`.

```
class Poly {
  public global val coeff : ValRail[Int];
  public def this(coeff: ValRail[Int]) { this.coeff = coeff;}
  public static operator (a:Int) as Poly! = new Poly([a]);
  public static def main(Rail[String]) {
     val three : Poly! = 3 as Poly!;
  }
}
```

You may define a type conversion to a constrained type, like `Poly!` in the previous example. If you convert to a more specific constraint, X10 will use the conversion, but insert a dynamic check to make sure that you have satisfied the more specific constraint. For example:

```
class Uni(n:Int) {
  public def this(n:Int) : Uni{self.n==n} = {property(n);}
  static operator (String) as Uni{self.n != 9} = new Uni(3);
  public static def main(Rail[String]) {
    val u = "" as Uni{self.n != 9 && self.n != 3};
  }
}
```

The string `""` is converted to `Uni{self.n != 9}` via the defined conversion operator, and that value is checked against the remaining constraints `{self.n != 3}` at runtime. (In this case it will fail.)

There may be many conversions from different types to `T`, but there may be at most one conversion from any given type to `T`.

(B: Syntax :B)

9.8.4 Implicit Type Coercions

You may also define *implicit* type coercions to `T{c}` as static operators in class `T`. The syntax for this is `static operator (x:U) : T{c} = e`. Implicit coercions are used automatically by the compiler. **(B: How does this work? One coercion, or a chain, and how about ambiguity? :B)**

For example, we can define an implicit coercion from `Int` to `Poly!`, and avoid having to define the sum of an integer and a polynomial as many special cases.

In the following example, we only define + on two polynomials (using a `static` operator, so that implicit coercions will be used – they would not be for an instance method operator). The calculation `1+x` coerces 1 to a polynomial and uses polynomial addition to add it to `x`.

```
public static safe operator (c : Int) : Poly! = new Poly([c]);

public static operator (p:Poly!) + (q:Poly!) = new Poly(
    ValRail.make[Int](
      Math.max(p.coeff.length(), q.coeff.length()),
      (i:Int) => p.a(i) + q.a(i)
   ));

public static def main(ValRail[String]) {
   val x = new Poly([0,1]);
   x10.io.Console.OUT.println("1+x=" + (1+x));
}
```

(B: Syntax :B)

9.8.5 `set` and `apply`

X10 allows types to implement the subscripting / function application operator, and indexed assignment. The `Array`-like classes take advantage of both of these in `a(i) = a(i) + 1`. Unlike unary and binary operators, subscripting and indexed assignment are done by methods, `apply` and `set` respectively.

`a(b,c,d)` is short for the method call `a.apply(b,c,d)`. Since it is possible to overload methods, the application syntax can be overloaded. For example, an ordered dictionary structure could allow subscripting by numbers with `def apply(i:Int)`, and by string-valued keys with `def apply(s:String)`.

`a(i)=b` is short for the method call `a.set(b,i)`, with one or more indices `i`. (This has a possibly surprising consequence for the order of evaluation: in `a(i)=b`, as in `a.set(b,i)`, `a` is evaluated first, then `b`, and finally `i`.) Again, it is possible to overload `set` to provide a variety of subscripting operations.

The `Oddvec` class of somewhat peculiar vectors illustrates this. `a()` returns a string representation of the oddvec, which probably should be done by `toString()` instead. `a(i)` picks out one of the three coordinates of `a`, which is sensible.

`a(i)=b` assigns to one of the coordinates. `a(i,j)=b` assigns different values to `a(i)` and `a(j)`, purely for the sake of the example.

```
class Oddvec {
  var v : Rail[Int]! = Rail.make[Int](3, (Int)=>0);
  public def apply() = "(" + v(0) + "," + v(1) + "," + v(2) + ")";
  public def apply(i:Int) = v(i);
  public def set(newval:Int, i:Int) = {v(i) = newval;}
  public def set(newval:Int, i:Int, j:Int) = {
       v(i) = newval; v(j) = newval+1;}
  // ...
```

10 Structs

An instance of a class `C` (an *object*) is represented in X10 as a contiguously allocated chunk of words in the heap, containing the fields of the object as well as one or more words used in method lookup (itable/vtable). Variables with base type `C` (or a supertype of `C`) are implemented as cells with enough memory to hold a *reference* to the object. The size of a reference (32 bits or 64 bits) depends on the underlying operating system.

For many high-performance programming idioms, the overhead of one extra level of indirection represented by an object is not acceptable. For instance, a programmer may wish to define a type `Complex` (consisting of two double fields) and require that instances of this type be represented precisely as these two fields. A variable or field of type complex should, therefore, contain enough space to store two doubles. An array of complex of size `N` should store `2*N` doubles. Method invocations should be resolved statically so that there is no need to store vtable/itable words with each instance. Parameters of type complex should be passed inline to a method as two doubles. If a method's return type is complex the method should return two doubles on the stack. Two values of this type should be equal precisely when the two doubles are equal (structural equality).

X10 supports the notion of *structs* which are precisely objects that can be implemented inline with a contiguous chunk of memory representing their fields, without any vtable/itable. Structs are introduced by struct definitions. struct definitions look very similar to class definitions, but have additional restrictions.

Implementation Note In X10 v2.0, structs are only implemented as inline headerless objects when using the C++ backend. In the Java backend, structs are mapped directly to Java classes and therefore have the same space and indirection characteristics as classes.

10.1 Struct declaration

X10 supports user-defined primitives (called *structs*). Like classes, structs define zero or more fields and zero or more methods, and may implement zero or more interfaces. A struct has the same modifiers as a class. However, structs are implicitly `val` and do *not* participate in any code inheritance relation. (This makes structs very easy to implement, without vtables.)

StructModifiers$^?$
`struct C[`$\mathtt{X}_1$`, ..., `$\mathtt{X}_n$`](`$\mathtt{p}_1$`:`$\mathtt{T}_1$`, ..., `$\mathtt{p}_n$`:`$\mathtt{T}_n$`){c}`
`  implements `$\mathtt{I}_1$`, ..., `$\mathtt{I}_k$` {`
StructBody
`}`

Each field and method in a struct is implicitly marked `global`.

The size of a variable of struct type `C` is the size of the fields defined at `C` (up to alignment considerations). No extra space is allocated for a vtable or an itable. This means that unlike classes, structs cannot be defined recursively. That is, a struct `S` cannot contain a field of type `S`, or a field of struct type `T` which, recursively, contains a field of type `S`.

- More precisely, we require that the set of *size equations* for all structs and classes must have a unique solution. A size equation for a struct `S` is defined as follows. Assume `S` has m fields of type $\mathtt{S}_i$ (for i in $0, \ldots, m-1$), and n fields of type (class) $\mathtt{C}_j$ (for j in $0, \ldots, n-1$). Then the size equation for `S` is

  $$\mathtt{size(S)} = \mathtt{size}(\mathtt{S}_0)+\ldots+\mathtt{size}(\mathtt{S}_{m-1})+\mathtt{size}(\mathtt{C}_0)+\ldots+\mathtt{size}(\mathtt{C}_{n-1})$$

  The size equation for a class `C` is just `size(C) = AddressSize`, where `AddressSize` is a compile-time parameter.

Values of a struct `C` type can be created by invoking a constructor defined in `C`, but without prefixing it with `new`.

Constrained types can be built on top of the base `C` in the same way as they can be built on top of a class `D`. In struct `C[T1,..., Tn]{c}`, the type of `self` in `c` is `C[T1,..., Tn]`.

10.2 Boxing of structs

If a struct `S` implements an interface `I` (e.g., `Any`), a value `v` of type `S` can be assigned to a variable of type `I`. The implementation creates an object `o` that is an instance of an anonymous class implementing `I` and containing `v`. The result of invoking a method of `I` on `o` is the same as invoking it on `v`. This operation is termed *auto-boxing*

In a generic class or struct obtained by instantiating a type parameter `T` with a struct `S`, variables declared at type `T` in the body of the class are not boxed. Rather they are implemented as if they were declared at type `S`.

10.3 Implementation of `Any` methods

Unlike objects, structs do not have global identity. Instead, two structs are equal (==) if and only if their corresponding fields are equal (==). This is the central property of structs.

All structs implement `x10.lang.Any`. All structs have the following methods implicitly defined on them:

```
property def home()=here;
property def at(Place)=true;
property def at(Object)=true;
```

It is an error for a programmer to attempt to define them.

Structs are required to implement the following methods:

```
public global safe def equals(Any):Boolean;
public global safe def hashCode():Int;
public global safe def typeName():String;
public global safe def toString():String;
```

These methods are defined automatically if they are not supplied by the programmer. A programmer who provides an explicit implementation of `equals(Any)` for a struct `S` should also consider supplying a definition for `equals(S):Boolean`. This will often yield better performance since the cost of an upcast to `Any` and then a downcast to `S` can be avoided.

Expressions of a struct type may be used in `instanceof` and `as` expressions.

10.4 "Primitives"

The package `x10.lang` provides the following structs. Most of the functionality of these structs is implemented natively.

```
boolean, char,
byte, short, int, long
float, double
ubyte, ushort, uint, ulong
```

10.5 Generic programming with structs

An unconstrained type variable `X` can be instantiated with `Object` or its subclasses or structs or functions.

Within a generic struct, all the operations of `Any` are available on a variable of type `X`. Additionally, variables of type `X` may be used with `==, !=`, in `instanceof`, and casts.

The programmer must be aware of the different interpretations of equality for structs and classes and ensure that the code is correctly written for both cases. If necessary the programmer can write code that distinguishes between the two cases (a type parameter `X` is instantiated to a struct or not) as follows:

```
val x:X = ...;
if (x instanceof Object) { // x is a real object
   val x2 = x as Object; // this cast will always succeed.
   ...
} else { // x is a struct
   ...
}
```

10.6 Programming Methodology

A programmer should by default organize his/her code in a class hierarchy, providing structs only in those well-thought situations where concrete types are appropriate.

10.6.1 Compatibility Note

A value class in X10 v1.7 can often be translated into a struct in X10 2.0. The crucial conditions to be checked manually are:

- A struct is of bounded size.
- Each method is global.
- The class is final.

If these conditions are not met, the value class should be converted into a class with `global` fields and methods.

10.6.2 Examples

An example illustrating pairing:

```
struct Pair[S,T]  {
  val x: S;
  val y: T;
  def this(x: S, y: T) {
    this.x=x;
    this.y=y;
  }
  def x()=x;
  def y()=y;
  public global safe def hashCode() = x.hashCode() + y.hashCode();
  public def equalsX[U](o:Pair[S,U]) = x==o.x;
  public def equalsY[U](o:Pair[U,T]) = y==o.y;
  public global safe def equals(that:Any) = this == that;
  public def equals(that:Pair[S,T]) = this==that;
}
```

The following types all make sense:

- `Pair[Complex, String]`: A struct with two fields, one inlined field of type `Complex` and another of type `String`.

- `Pair[Complex, Int]`: A struct with two fields, one inlined field of type `Complex` and another of type `Int`.

The definition of `x10.lang.Complex` provides a good example of the use of structs.

11 Functions

11.1 Overview

The runtime entities in X10 are of three kinds: *structs*, *objects*, and *functions*. This section is concerned with functions and their types – how they are created, and what operations can be performed on them.

Intuitively, a function is a piece of code which can be applied to a set of arguments to produce a value. The application may not terminate, or may terminate abruptly. Functions may throw checked and unchecked exceptions. The body of a function may be any X10 expression: hence a function evaluation may spawn multiple activities, read and write mutable locations, wait until memory locations contain a desired value, and execute over multiple places. In particular, function evaluation may be non-deterministic. When applied to the same input twice, a function may yield two different results.

It is a limitation of X10 v2.0 that functions do not support type arguments. This limitation may be removed in future versions of the language.

A *function literal* `(x1:T1,..,xn:Tn){c}:T=>e` creates a function of type `(x1:T1,...,xn:Tn){c}=>T` (§4.5). The body `e` of such an expression is type-checked in an environment in which `c` is true.

A function literal evaluates to a function entity ϕ. When ϕ is applied to a suitable list of actual parameters `a1`-`an`, it evaluates `e` with the formal parameters bound to the actual parameters. So, the following are equivalent, where `e` is an expression involving `x1` and `x2`[1]

[1] Strictly, there are a few other requirements; *e.g.*, `result` must be a `var` of type `T` defined outside the outer block, the variables `a1` and `a2` had better not appear in `e`, and everything in sight had better typecheck properly.

```
{
  val f = (x1:T1,x2:T2){true}:T => e;
  val a1 : T1 = arg1();
  val a2 : T2 = arg2();
  result = f(a1,a2);
}
```

and

```
{
  val a1 : T1 = arg1();
  val a2 : T2 = arg2();
  {
     val x1 : T1 = a1;
     val x2 : T2 = a2;
     result = e;
  }
}
```

The *method selector expression* `e.m.(x1:T1,...,xn:Tn)` (§11.3) permits the specification of the function underlying the method `m`, which takes arguments of type `(x1:T1,..., xn:Tn)`. Within this function, `this` is bound to the result of evaluating `e`.

Function types may be used in `implements` clauses of class definitions. Instances of such classes may be used as functions of the given type. Indeed, an object may behave like any (fixed) number of functions, since the class it is an instance of may implement any (fixed) number of function types.

11.2 Function Literals

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

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

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

Functions have zero or more formal parameters, an optional return type and optional set of exceptions throws by the body. The 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 function using a return statement (§13.15). The type of a function is a function type (§4.5). In some cases the return type `T` is also optional and defaults to the type of the body. If a formal `xi` does not occur in any `Tj`,`c`,`T` or `e`, the declaration `xi:Ti` may be replaced by just `Ti`.

As with methods, a function 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 `val`s in scope at the function expression.

The body of the function is evaluated when the function is invoked by a call expression (§12.6), not at the function's place in the program text.

As with methods, a function with return type `Void` cannot have a terminating expression. If the return type is omitted, it is inferred, as described in §4.11. It is a static error if the return type cannot be inferred.

Example 11.2.1 The following method takes a function parameter and uses it to test each element of the list, returning the first matching element. It returns `otherwise` if no element matches.

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

The method may be invoked thus:

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

□

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

11.2.1 Outer variable access

In a function `(`x_1`: `T_1`, ..., `x_n`: `T_n`){c} => { s }` the types T_i, the guard `c` and the body `s` may access many, though not all, sorts of variables from outer scopes. Specifically, they can access:

- All fields of the enclosing object and class;
- All type parameters;
- All `val` variables;
- `var` variables with the `shared` annotation.

Limitation: `shared` *is not currently supported.*

The function body may refer to instances of enclosing classes using the syntax `C.this`, where `C` is the name of the enclosing class. `this` refers to the instance of the immediately enclosing class, as usual.

For example, the following is legal. However, it would not be legal to add `e` or `h` to the sum; they are non-`shared` `var`s from the surrounding scope.

```
class Lambda {
   var a : Int = 0;
   val b = 0;
   def m(var c : Int, shared var d : Int,  val e : Int) {
      var f : Int = 0;
      shared var g : Int = 0;
      val h : Int = 0;
      val closure = (var i: Int, val j: Int) => {
          return a + b + d + e + g + h + i + j + this.a + Lambda.this.a;
      };
      return closure;
   }
}
```

Rationale: Non-`shared` `var`s like `e` and `h` are excluded in X10, as in many other languages, for practical implementation reasons. They are allocated on the stack, which is desirable for efficiency. However, the closure may exist for long after the stack frame containing `e` and `h` has been freed, so those storage locations are no

longer valid for those variables. `shared var`s are heap-allocated, which is less efficient but allows them to exist after `m` returns.

NOTE: `shared` does not guarantee **atomic** access to the shared variable. As with any code that might mutate shared data concurrently, be sure to protect references to mutable shared state with `atomic`. For example, the following code returns a pair of closures which operate on the same shared variable `a`, which are concurrency-safe—even if invoked many times simultaneously. Without `atomic`, it would no longer be concurrency-safe.

```
def counters() {
    shared var a : Int = 0;
     return [
        () => {atomic a ++;},
        () => {atomic return a;}
        ];
 }
```

11.3 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*$^{?}$ `)`

The *method selector expression* `e.m.(T1,...,Tn)` is type correct only if it is the case that the static type of `e` is a class or struct or interface with a method `m(x1:T1,...xn:Tn){c}:T` defined on it (for some `x1,...,xn,c,T`). At runtime the evaluation of this expression evaluates `e` to a value `v` and creates a function `f` which, when applied to an argument list `(a1,...,an)` (of the right type) yields the value obtained by evaluating `v.m(a1,...,an)`.

Thus, the method selector

`e.m.[`X_1`, ..., `X_m`](`T_1`, ..., `T_n`)`

behaves as if it were the function

```
((v:T)=>
  [X_1, ..., X_m](x_1: T_1, ..., x_n: T_n) => v.m[X_1, ..., X_m](x_1, ..., x_n))
(e)
```

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 `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 function 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.

11.4 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): Long => x - y
Float.-(Float,Float)  ≡ (x: Float, y: Float): Float => 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 (§12.9) is not performed when invoking these operations; instead, the operands are coerced individually via implicit coercions (§4.9), as appropriate.

Planned 11.4.1 The following is not implemented in version 2.0.3:

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

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

□

11.5 Functions as objects of type `Any`

Two functions `f` and `g` are equal ("==") if both are instances of classes and the same object, or if both were obtained by the same evaluation of a function literal.[2] Further, it is guaranteed that if two functions are equal then they refer to the same locations in the environment and represent the "same code" (so their executions in an identical environment are comparable).

Every function type implements all the methods of `Any`. For a value `f` of a function type `(x1:T1,...xn:Tn)=>T`, the expression `f.equals(g)` is of type `Boolean`. It succeeds if and only if `f==g` succeeds. Similarly, the expression `f.hashCode()` is of type `Int` and returns an implementation defined hash code which is guaranteed to be the same for two values that are `equal`. The expression `f.toString()` returns an implementation-dependent string. Two strings returned on different evaluations are `equal` to each other. Similarly, `f.typeName()`

[2] A literal may occur in program text within a loop, and hence may be evaluated multiple times.

returns an implementation-dependent string. Two strings returned on different evaluations are `equal` to each other.

The method `f.home()` always returns `here`. The call `f.at(p)` for `p` a value of type `Place` always returns `true`. The call `f.at(o)` for `o` a value of type `Object` always returns `true`

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 reference class, struct, or function. 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

The syntax for literals is given in §3.

The type that X10 gives a literal includes its value. *E.g.*, `1` is of type `Int{self==1}`, and `true` is of type `Boolean{self==true}`.

12.2 `this`

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

The expression `this` is a local `val` 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.

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. This form is used to access fields of the parent class shadowed by same-named fields of the current class.

If the field target is `Cls.super`, then the target's type is `Cls`, which must be an ancestor class of the enclosing class. This (admittedly obscure) form is used to access fields of an ancestor class which are shadowed by same-named fields of some more recent ancestor.

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 Function Literals

Please see §11 for a description of `FunctionExpressions`.

12.6 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 §4.11.

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

12.7 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` *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.

12.8 Increment and decrement

The operators ++ and -- increment and decrement a variable, respectively. The variable must be `var` 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.

12.9 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`.

12.10 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.

12.11 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.

12.12 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.

12.13 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`. This operation is deprecated, and may be removed in a later version of the language.

12.14 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.

12.15 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`.

12.16 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`.

12.17 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`.

12.18 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.

12.19 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.

12.20 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 (§4.8) of the types of the consequent and the alternative.

12.21 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.

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 struct type, then they must be structurally equal; that is, they must be instances of the same struct and all their fields or components must be ==.

The definition of equality for function types is specified in §11.5.

If the operands both have numeric type, binary promotion (§12.9) 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.

12.22 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 (§12.6).

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 §12.6.

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.

12.23 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, and either a compile-time error or a runtime exception (`x10.lang.TypeCastException`) if it is not.

When evaluating `E as T{c}`, first the value of `E` is converted to type `T` (which may fail), and then the constraint `{c}` is checked.

- If `T` is a primitive type, then `E`'s value is converted to type `T` according to the rules of §4.9.1.
- If `T` is a class, then the first half of the cast succeeds if the run-time value of `E` is an instance of class `T`, or of a subclass
- If `T` is an interface, then the first half of the cast succeeds if the run-time value of `E` is an instance of a class implementing `T`.
- If `T` is a struct type, then the first half of the cast succeeds if the run-time value of `E` is an instance of `T`.

- If `T` is a function type, then the first half of the cast succeeds if the run-time value of `X` is a function of precisely that type.

If the first half of the cast succeeds, the second half – the constraint `{c}` – must be checked. In general this will be done at runtime, though in special cases it can be checked at compile time. For example, `n as Int{self != w}` succeeds if `n != w` — even if `w` is a value read from input, and thus not determined at compile time.

The compiler may forbid casts that it knows cannot possibly work. If there is no way for the value of `E` to be of type `T{c}`, then `E as T{c}` can result in a static error, rather than a runtime error. For example, `1 as Int{self==2}` may fail to compile, because the compiler knows that `1`, which has type `Int{self==1}`, cannot possibly be of type `Int{self==2}`.

12.24 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. At run time, the result of this operator is `true` if the *RelationalExpression* can be coerced to *Type* without a `ClassCastException` 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.

12.25 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 $\texttt{T}_1$ `==` $\texttt{T}_2$ evaluates to `true` $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ and if $\texttt{T}_2$ is a subtype of $\texttt{T}_1$.

Subtyping expressions are used in subtyping constraints for generic types.

12.26 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`.

12.27 Rail constructors

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

The rail constructor $[\texttt{a}_0, \ldots, \texttt{a}_{k-1}]$ creates an instance of `ValRail` with length k where the ith element is $\texttt{a}_i$. The element type of the array (`T`) is bound to the least common ancestor of the types of the $\texttt{a}_i$ (§4.8).

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 Statements

This chapter describes the statements in the sequential core of X10. Statements involving concurrency and distribution are described in §15.

13.1 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 a rail.

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

13.2 Local variable declaration

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

The syntax of local variables declarations is described in §5.

Local variables may be declared only within a block statement (§13.3). The scope of a local variable declaration is the statement itself and the subsequent statements in the block.

13.3 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.

13.4 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.

13.5 Labeled statement

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

Loop statements (`for`, `while`, `do`, `ateach`, `foreach`) 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.

13.6 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;
outer: for (var i: Int = 0; i < a.length; i++)
    for (var j: Int = 0; j < a(i).length; j++)
        if (a(i)(j) == v) {
            found = true;
            break outer;
        }
```

13.7 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.

13.8 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, which must be of type `Boolean`. If the condition is `true`, it evaluates the then-clause. If the condition is `false`, the if-then statement completes normally.

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

As is traditional in languages derived from Algol, the if-statement is ambiguous. That is,

```
if (B1) if (B2) S1 else S2
```

could be intended to mean either

```
if (B1) { if (B2) S1 else S2 }
```

or

```
if (B1) {if (B2) S1} else S2
```

X10, as is traditional, attaches an `else` clause to the most recent `if` that doesn't have one. This example is interpreted as `if (B1) { if (B2) S1 else S2 }`.

13.9 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`, `Byte`, `Short`, or `Char` and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

13.10 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`.

13.11 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`.

13.12 For statement

Statement	::=	*ForStatement*
	|	*EnhancedForStatement*
ForStatement	::=	`for` `(` *ForInit*? `;` *Expression*? `;` *ForUpdate*? `)` *Statement*
ForInit	::=	*StatementExpression* (`,` *StatementExpression*)*
	|	*LocalVariableDeclaration*
ForUpdate	::=	*StatementExpression* (`,` *StatementExpression*)*
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 provides for arbitrary iteration in a somewhat more organized fashion than a `while`. `for(init; test; step)body` is equivalent to:

```
{
   init;
   while(test) {
      body;
      step;
   }
}
```

`init` is performed before the loop, and is traditionally used to declare and/or initialize the loop variables. It may be a single variable binding statement, such as `var i:Int = 0` or `var i:Int=0, j:Int=100`. (Note that a single variable binding statement may bind multiple variables.) Variables introduced by `init` may appear anywhere in the `for` statement, but not outside of it. Or, it may be a sequence of expression statements, such as `i=0, j=100`, operating on already-defined variables. If omitted, `init` does nothing.

`test` is a Boolean-valued expression; an iteration of the loop will only proceed if `test` is true at the beginning of the loop, after `init` on the first iteration or or `step` on later ones. If omitted, `test` defaults to `true`, giving a loop that will run until stopped by some other means such as `break`, `return`, or `throw`.

`step` is performed after the loop body, between one iteration and the next. It traditionally updates the loop variables from one iteration to the next: *e.g.*, `i++` and `i++,j--`. If omitted, `step` does nothing.

`body` is a statement, often a code block, which is performed whenever `test` is true. If omitted, `body` does nothing.

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]`. Destructuring syntax may be used for the formal parameter (§5). Each iteration of the loop binds the parameter to another element of the collection. The formal paramter must be immutable.

In a common case, 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.

13.13 Throw statement

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

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

Example 13.13.1 The following statement checks if an index is in range and throws an exception if not.

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

□

13.14 Try–catch statement

Statement	::=	*TryStatement*
TryStatement	::=	`try` *BlockStatement Catch*$^+$
	|	`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.

13.15 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 must return without a value; otherwise, it must return a value of the appropriate type.

14 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 struct, `x10.lang.Place`; all places are instances of this struct.

In X10 v2.0, 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, §17.3.)

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 (§15.5). 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 `val`.

14.1 Place expressions

Any expression of type `Place` is called a place expression. Examples of place expressions are `this.home` (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 (§15.2).
- In a cast expression (§12.23).
- In an `instanceof` expression (§12.24).
- 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.

14.2 `here`

X10 supports a special indexical constant[1] `here`, which evaluates to the place at which the current activity is running.

Expression ::= `here`

Example 14.2.1 For example, the following method looks through a collection of `Thing`s for ones which belong in the current place `here`, and deals with the things which do. Note that every object `thing` has a property `thing.home` giving its home location.

```
public static def dealWithLocal(things: Rail[Thing]) {
   for(thing in things) {
       if (thing.home == here)
          dealWith(thing);
   }
}
```

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

□

here is frequently used in constraints, quite often of the form obh.home == here. Such constraints are necessary to check that a non-global method can be called on ob:

```
val ob : Thing{self.home == here} = new Thing();
ob.nonGlobalMethod();
```

This idiom is so common and useful that the constraint T{self.home==here} can be abbreviated as T!:

```
val ob : Thing! = new Thing();
ob.nonGlobalMethod();
```

15 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 v2.0, an activity may not be interrupted, blocked 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 (§15.1).

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* (§15.5).

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

15.1 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 blocked at a statement (such as the `finish` statement §15.4) 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".

15.2 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, blocked or terminated.

[2] In X10 v2.0 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` (§14). If not, the compiler replaces `e` with `e.home` if `e` is of type `x10.lang.Object`. 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).home`, 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 occurring 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 §15.4), 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 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.

[3] X10 v2.0 does not specify a particular algorithm; this will be fixed in future versions.

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 anonymous 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, §16) provided that such variables are (implicitly or explicitly) `val`.

15.3 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. *E.g.*, if `obj` is an object with a non-`global` method `meth()`, the general way to call the method from anywhere is

```
at(x.home) x.meth();
```

Or, if you want to capture the result of the method call, use the `at`-expression:

```
val res = at(x.home)x.meth();
```

`at(p)S` does *not* start a new activity. It should be thought of as transporting the current activity to `p`, running `S` there, and then transporting it back. If you want to start a new activity, use `async`; if you want to start a new activity at `p`, use `at(p) async S`.

As a consequence of this, `S` may contain constructs which only make sense within a single activity. For example,

```
for(x in things)
  if (at(x.home) x.nice())
      return x;
```

returns the first nice thing in a collection. If we had used `async at(x.home)`, this would not be allowed; you can't `return` from an `async`.

15.4 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, `x10.lang.MultipleExceptions` 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 (§16).

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 v2.0 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.*

15.5 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 §14 for a discussion of places.)

15.6 Foreach statements

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

The `foreach` statement is a parallel version of the enhanced `for` statement (§13.12). `for(x in C)S` executes `S` *sequentially*, with everything happening `here`. `foreach(x in C)S` executes `S` for each iteration of the loop *in parallel*, located at `x.home`. It is thus equivalent to:

```
foreach (x in C)
  async at (x.home) S
```

As a common and useful special case, `C` may be a `Dist` or an `Array`. For both of these, `foreach(x in C)S` is treated just like `foreach(x in C.region)S`. `x` ranges over the `Point`s of the region. Each activity that `foreach` starts is located at `here` – the same place that the `foreach` statement itself is executing. (If you want to start an activity at the place where the array element `C(p)` is located, use `ateach` (§15.7) instead of `foreach`.)

Exceptions thrown by `S`, like other exceptions in `async`s, are propagated to the root activity of the `foreach`.

15.7 Ateach statements

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

The `ateach` statement is similar to the `foreach` statement, but it spawns activites at each place of a distribution. In `ateach(p in D) S`, `D` must be either of type `Dist` or of type `Array[T]`, and `p` will be of type `Point`.

This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, if `D` is a `Dist`, `ateach(p in D) S` could be implemented as:

```
foreach (p in D.region)
  async (D(p)) S(p)
```

However, the compiler may implement it more efficiently to avoid extraneous communications. In particular, it is recommended that `ateach(p in D)S` be implemented as the following code, which coordinates with each place of `D` just once, rather than once per element of `D` at that place:

```
foreach (p in D.places()) at (p) {
    foreach (pt in D|here) {
        S(p);
    }
}
```

If `e` is an `DistArray[T]`, then `ateach (p in e)S` is identical to `ateach(p in e.dist)S`; the iteration is over the array's underlying distribution. The code below is a common and generally efficient way to work with the elements of a distributed array:

```
ateach(p in A)
  dealWith(A(p));
```

15.8 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 `val`.

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 {
  global def forced(): Boolean;
  global def force(): T;
}
```

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.

```
val promise: Future[T] = future (a.dist(pt)) a(pt);
val value: T = promise();
```

15.9 At expressions

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

An `at` expression evaluates an expression synchronously at the given place and returns its value. Note that expression evaluation may spawn asynchronous activities. The `at` expression will return without waiting for those activities to terminate. That is, `at` does not have built-in `finish` semantics.

15.10 Shared variables

Compiler Limitation: Shared variables are not currently implemented.

A *shared local variable* is declared with the annotation `shared`. It can be accessed within any control construct in its scope, including `async`, `at`, `future` and closures.

Note that the lifetime of some of these constructs may outlast the lifetime of the scope – requiring the implementation to allocate them outside the current stack frame.

15.11 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.

15.11.1 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 v2.0 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 blocked. 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 (such as creation of a `future`).

It may *not* include any statement that may potentially block at runtime (e.g., `when`, `force` operations, `next` operations on clocks, `finish`).

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 `safe` at that point in the code (§9.6.3).

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

```
atomic {s1; atomic s2}  =  atomic {s1; s2}        (15.1)
```

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:

```
var target:Object = null;
public atomic def CAS(old: Object, new: Object): Boolean {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

15.11.2 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 intermmittently. 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 (as for the `sequential` qualifier on methods) and to have a statically determinable upper bound on their execution (as for the `nonblocking` qualifier on methods). 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 15.11.1 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 {
  var datum: Object = null;
  var 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;
    }
  }
}
```

□

16 Clocks

Many concurrent algorithms proceed in phases: in phase k, several activities work independently, but synchronize together before proceeding on to phase $k + 1$. X10 supports this communication structure (and many variations on it) with a generalization of barriers **(B: cite something :B)** called *clocks*. Clocks are structured to allow barrier synchronization without introducing either deadlocks or race conditions.

The following minimalist example of clocked code has two worker activities A and B, and three phases. In the first phase, each worker activity says its name followed by 1; in the second phase, by a 2, and in the third, by a 3. So, if `say` prints its argument, `A-1 B-1 A-2 B-2 A-3 B-3` would be a legitimate run of the program, but `A-1 A-2 B-1 B-2 A-3 B-3` (with `A-2` before `B-1`) would not.

The program creates a clock `cl` to manage the phases. Each worker does the work of its first phase, and then executes `next;` to signal that it is finished with that work. `next;` is blocking, and causes the worker to wait until all workers have finished with the phase – as measured by the clock `cl` to which they are both registered. Then they do the second phase, and another `next;` to make sure that neither proceeds to the third phase until both are ready. This example uses `finish` to wait for both workers to finish. The parent thread is also registered on the clock just as the workers are, and executes `next;next;` to run through the phases.

```
val cl = Clock.make();
finish{
  async clocked(cl) {// Activity A
    say("A-1");
    next;
    say("A-2");
    next;
    say("A-3");
```

```
  }// Activity A

  async clocked(cl) {// Activity B
    say("B-1");
    next;
    say("B-2");
    next;
    say("B-3");
  }// Activity B
  next;next;
}
say("All done");
```

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.

The primary operations that an activity `a` may perform on a clock `c` that it is registered upon are:

- It may *register* a newly-created activity on `c`: `async clocked(c){S}`.
- It may *unregister* itself from `c`, with `c.drop()`. After doing so, it can no longer use most primary operations on `c`.
- It may *resume* the clock, with `c.resume()`, indicating that it has finished with the current phase associated with `c` and is ready to move on to the next one.
- It may *block* on all the clocks it is registered with simultaneously, by the command `next;`, which indicates that it has finished the current phase according to all of them; this statement `resume()`s all of them. `a` will only continue executing when all of these clocks have been `resume()`d by all activities registered upon them.

- Other miscellaneous operations are available as well; see the `Clock` API.

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 (§15.4) guarantees that this cannot cause a cycle in the wait-for graph. A more rigorous proof may be found in [9].

16.1 Clock operations

There are two language constructs for working with clocks. `async clocked(cl) S` starts a new activity registered on one or more clocks. `next;` blocks the current activity until all the activities sharing clocks with it are ready to proceed. Clocks are objects, and have a number of useful methods on them as well.

16.1.1 Creating new clocks

Clocks are created using a factory method on `x10.lang.Clock`:

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

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).

16.1.2 Registering new activities on clocks

The statement

```
async clocked (c1, c2, c3) S
```

starts a new activity, initially registered with clocks `c1`, `c2`, and `c3`, and running `S`. The activity running this code must be registered on those clocks. Furthermore, it cannot be quiescent on any of them (see §16.1.3), because that would introduce a race condition. Violations of these conditions are punished by the throwing of a `ClockUseException`.

An activity may check that it is registered on a clock `c` by the predicate `c.registered()`

NOTE: X10 does not contain a "register" operation that would allow an activity to discover a clock in a datastructure and register itself on it. Therefore, while a clock `c` may be stored in a data structure by one activity `a` and read from it by another activity `b`, `b` cannot do much with `c` unless it is already registered with it. In particular, it cannot register itself on `c`, and, lacking that registration, cannot register a sub-activity on it with `clocked(c) async S`.

16.1.3 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 `c.resume()`.

An activity may invoke `resume()` only on a clock it is registered with, and has not yet dropped (§16.1.5). A `ClockUseException` is thrown if this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase. Otherwise, `c.resume()` 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: It is a static 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()`.)

16.1.4 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 quiescence 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.

16.1.5 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`.

16.2 Program equivalences

From the discussion above it should be clear that the following equivalences hold:

```
      c.resume(); next;  =  next;                        (16.1)
c.resume(); d.resume();  =  d.resume(); c.resume();      (16.2)
c.resume(); c.resume();  =  c.resume();                  (16.3)
```

Note that `next; next;` is not the same as `next;`. The first will wait for clocks to advance twice, and the second once.

17 Arrays

This chapter provides an overview of the class `Array`, and its supporting classes `Point`, `Region` and `Dist`. All these classes are in the `x10.lang` package. For more details, please consider the API documentation.

An array is a mapping from a distribution to a range data type. Multiple arrays may be declared with the same underlying distribution.

Each array has a field `a.dist` which may be used to obtain the underlying distribution.

17.1 Points

Arrays are indexed by points–n-dimensional tuples of integers, implemented by the class `x10.lang.Point`. X10 specifies a simple syntax for the construction of points. A rail constructor (§12.27) 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:

CHECK: *Huh? Why do we say rail constructor above? As opposed to just any old valrail?*

```
p: Point(2) = [1,2];
```

The `Point` factory method `make` can take a rail constructor as argument. The assignment above can be written, without implicit coercion, as:

```
p: Point = Point.make([1,2]);
```

Points implement the function type `(Int) => Int`; thus, the `i`th element of a point `p` may be accessed as `p(i)`.

17.2 Regions

A region is a set of points. X10 provides a built-in class, `x10.lang.Region`, to allow the creation of new regions and to perform operations on regions.

Each region `R` has a constant integer rank, `R.rank`.

Here are several examples of region declarations:

```
val MAX_HEIGHT=20;
val Null = Region.makeUnit();  // Empty 0-dimensional region
val N = 10;
val K = 2;
val R1 = 1..100; // 1-dim region with extent 1..100
val R2 = [1..100] as Region(1); // same as R1
val R3 = (0..99) * (-1..MAX_HEIGHT);
val R4 = [0..99, -1..MAX_HEIGHT] as Region(2); // same as R3
val R5 = Region.makeUpperTriangular(N);
val R7 = R4 && R5; // intersection of two regions
val R8 = R4 || R5; // union of two regions
```

The expression $\texttt{a}_1\texttt{..a}_2$ is shorthand for the rectangular, rank-1 region consisting of the points $\{\texttt{[a}_1\texttt{]}, \ldots, \texttt{[a}_2\texttt{]}\}$. Each subexpression of $\texttt{a}_i$ must be of type `Int`. If $\texttt{a}_1$ is greater than $\texttt{a}_2$, the region is empty.

A region may be constructed by converting from a rail of regions or a rail of points, typically using a rail constructor (§12.27) (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. X10 v2.0 does not (yet) support hierarchical regions.

Various built-in regions are provided through factory methods on `Region`. For instance:

- `Region.makeUpperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `Region.makeLowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular `N x N` matrix.

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` (§13.12) iterate over the points in a region in the canonical order.

A region is said to be *rectangular* if it is of the form `(`$\mathtt{T}_1$ `*` $\cdots$ `*` $\mathtt{T}_k$`)` for some set of regions $\mathtt{T}_i$. 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 (that is, it is *convex*). (Note that `||` may produce non-convex regions from convex regions, e.g., `[1,1] || [3,3]` is a non-convex region. The operation `R.boundingBox()` gives the smallest rectangular region containing `R`.)

17.2.1 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 whose type establishes that they are of the same rank. Let `S` be a region of unrelated rank.

$\mathtt{R}_1$ `&&` $\mathtt{R}_2$ 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 $\mathtt{R}_2$.

$\mathtt{R}_1$ `*` `S` is the Cartesian product of $\mathtt{R}_1$ and `S`, formed by pairing each point in $\mathtt{R}_1$ with every the point in `S`. Thus, `([1..2,3..4] as Region 2) * (5..6)` is the region of rank `3` containing the points `(x,y,z)` where `x` is `1` or `2`, `y` is `3` or `4`, and `z` is `5` or `6`.

For a region `R` and point `p` of the same rank `R+p` and `R-p` represent the translation of the region with `p`. That is, point `q` is in `R` if and only if point `q+p` is in `R+p`. (And similarly for `R-p`.)

For more details on the available methods on `Region`, please consult the API documentation.

17.3 Distributions

A *distribution* is a mapping from a region to a set of places. X10 provides a built-in class, `x10.lang.Dist`, to allow the creation of new distributions and to perform

operations on distributions. This class is `final` in X10 v2.0; future versions of the language may permit user-definable distributions.

The *rank* of a distribution is the rank of the underlying region.

```
R: Region = 1..100;
D: Dist = Dist.makeBlock(R);
D: Dist = Dist.makeCyclic(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`.

17.3.1 Operations returning distributions

Let `R` be a region, `Q` a set of places $\{$`p1`, ..., `pk`$\}$ (enumerated in canonical order), and `P` a place.

Unique distribution The distribution `Dist.makeUnique(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`, as does `Dist.makeConstant(R,P)`.

Block distributions. The distribution `Dist.makeBlock(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 `Dist.makeBlock(R)` is the same distribution as `Dist.makeBlock(R, Place.places)`.

Cyclic distributions. The distribution `Dist.makeCyclic(R, Q)` distributes the points in `R` cyclically across places in `Q` in order.

The distribution `Dist.makeCyclic(R)` is the same distribution as `Dist.makeCyclic(R, Place.places)`.

Thus the distribution `Dist.makeCyclic(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 `Dist.makeCyclic(Place.places)`.

Block cyclic distributions. The distribution `Dist.makeBlockCyclic(R, N, Q)` distributes the elements of `R` cyclically over the set of places `Q` in blocks of size `N`.

Arbitrary distributions. The distribution `Dist.makeArbitrary(R,Q)` arbitrarily allocates points in `R` to `Q`. As above, `Dist.makeArbitrary(R)` is the same distribution as `Dist.makeArbitrary(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.

17.3.2 User-defined distributions

Future versions of X10 may provide user-defined distributions, in a way that supports static reasoning.

17.3.3 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_2`.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.

17.3.4 Example

```
def dotProduct(a: Array[T](D), b: Array[T](D)): Array[Double](D) =
  (new Array[T]([1:D.places],
      (Point) => (new Array[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.

17.4 Array initializer

Arrays are instantiated by invoking one of the make factory methods of the Array class.

An array creation must take either an Int as an argument or a Dist. In the first case an array is created over the distribution [0:N-1]->here; in the second over the given distribution.

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](1..1000->here, ((i):Point) => i);
val data2 : Array[Int]
    = Array.make[Int]([1..1000,1..1000]->here, ((i,j):Point) => i*j);
```

The first declaration stores in data a reference to a mutable 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 reference to a mutable 2-d array over [1..1000, 1..1000] initialized with i*j at point [i,j].

Other examples:

```
val D1:Dist(1) = ...; /* An expression that creates a Dist */
val D2:Dist(2) = ...; /* An expression that creates a Dist */

val data : Array[Int]
    = Array.make[Int](1000, ((i):Point) => i*i);

val data2 : Array[Float]
```

```
      = Array.make[Float](D1, ((i):Point) => i*i as Float);

val result : Array[Float]
   = Array.make[Float](D2, ((i,j):Point) => i+j as Float);;
```

17.5 Operations on arrays

In the following let `a` be an array with distribution `D` and base type `T`.

17.5.1 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`.

17.5.2 Constant promotion

For a distribution `D` and a val `v` of type `T` the expression `new Array[T](D, (p: Point) => v)` denotes the mutable array with distribution `D` and base type `T` initialized with `v` at every point.

17.5.3 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 (§17.3) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

17.5.4 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)`.

17.5.5 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.

18 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.

18.1 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 (var i: Int = 0; i < n; i++) { ... }

// An annotated empty statement ;
@Assert(x < y);
```

18.2 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.

18.3 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();
```

19 Linking with native code

On some platforms, X10 v2.0 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 via the `unsafe` annotation.

```
new unsafe Array[T](dist)
```

Unsafe arrays can be of any dimension. However, X10 v2.0 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 19.0.1 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] = new unsafe Array[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] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, London, UK, 2005. Springer-Verlag.

[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.

A Change Log

A.1 Changes from X10 v2.0

- `Any` is now the top of the type hierarchy (every object, struct and function has a type that is a subtype of `Any`). `Any` defines `home`, `at`, `toString`, `typeName`, `equals` and `hashCode`. `Any` also defines the methods of `Equals`, so `Equals` is not needed any more.
- Revised discussion of incomplete types (§9.4).
- The manual has been revised and brought into line with the current implementation.

A.2 Changes from X10 v1.7

The language has changed in the following way:

- **Type system changes**: There are now three kinds of entities in an X10 computation: objects, structs and functions. Their associated types are class types, struct types and function types.

  Class and struct types are called *container types* in that they specify a collection of fields and methods. Container types have a name and a signature (the collection of members accessible on that type). Collection types support primitive equality == and may support user-defined equality if they implement the `x10.lang.Equals` interface.

  Container types (and interface types) may be further qualified with constraints.

A function type specifies a set of arguments and their type, the result type, and (optionally) a guard. A function application type-checks if the arguments are of the given type and the guard is satisfied, and the return value is of the given type. A function type does not permit == checks. Closure literals create instances of the corresponding function type.

Container types may implement interfaces and zero or more function types.

All types support a basic set of operations that return a string representation, a type name, and specify the home place of the entity.

The type system is not unitary. However, any type may be used to instantiate a generic type.

There is no longer any notion of `value` classes. `value` classes must be re-written into structs or (reference) classes.

- **Global object model**: Objects are instances of classes. Each object is associated with a globally unique identifier. Two objects are considered identical == if their ids are identical. Classes may specify `global` fields and methods. These can be accessed at any place. (`global` fields must be immutable.)

- **Proto types.** For the decidability of dependent type checking it is necessary that the property graph is acyclic. This is ensured by enforcing rules on the leakage of `this` in constructors. The rules are flexible enough to permit cycles to be created with normal fields, but not with properties.

- Place types. Place types are now implemented. This means that non-global methods can be invoked on a variable, only if the variable's type is either a struct type or a function type, or a class type whose constraint specifies that the object is located in the current place.

  There is still no support for statically checking array access bounds, or performing place checks on array accesses.

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.