X10 Language Specification
Version 2.6.2

Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove

Please send comments to x10-core@lists.sourceforge.net

January 4, 2019

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 David Grove, Ben Herta, Louis Mandel, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, Olivier Tardieu.

Past members include Shivali Agarwal, Bowen Alpern, David Bacon, Raj Barik, Ganesh Bikshandi, Bob Blainey, Bard Bloom, Philippe Charles, Perry Cheng, David Cunningham, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Stephen Fink, Robert Fuhrer, Patrick Gallop, Christian Grothoff, Hiroshi Horii, Kiyokuni Kawachiya, Allan Kielstra, Sreedhar Kodali, Sriram Krishnamoorthy, Yan Li, Bruce Lucas, Yuki Makino, Nathaniel Nystrom, Igor Peshansky, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon, Toshio Suganuma, Sayantan Sur, Toyotaro Suzumura, Christoph von Praun, Leena Unnikrishnan, Pradeep Varma, Krishna Nandivada Venkata, Jan Vitek, Hai Chuan Wang, Tong Wen, Salikh Zakirov, and Yoav Zibin.

For extended discussions and support we would like to thank: Gheorghe Almasi, Robert Blackmore, Rob O'Callahan, Calin Cascaval, Norman Cohen, Elmootaz Elnozahy, John Field, Kevin Gildea, Sara Salem Hamouda, Michihiro Horie, Arun Iyengar, Chulho Kim, Orren Krieger, Doug Lea, John McCalpin, Paul McKenney, Hiroki Murata, 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], the Scala language specification [10], and ZPL [4].

This document specifies the language corresponding to Version 2.6.2 of the implementation. The redesign and reimplementation of arrays and rails was done by Dave Grove and Olivier Tardieu. 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

The era of the mighty single-processor computer is over. Now, when more computing power is needed, one does not buy a faster uniprocessor—one buys another processor just like those one already has, or another hundred, or another million, and connects them with a high-speed communication network. Or, perhaps, one rents them instead, with a cloud computer. This gives one whatever number of computer cycles that one can desire and afford.

The problem, then, is how to use those computer cycles effectively. One must understand how to divide up the available work into chunks that can be executed simultaneously without introducing undesirable indeterminacy, cycles of "deadly embrace" which jam up processors or causing processors to spin uselessly waiting for conditions that may never materialize.

One response to this problem has been to move to a *fragmented memory model*. Multiple processors are programmed largely as if they were uniprocessors, but are made to interact via a relatively language-neutral message-passing format such as MPI [12]. 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 Single Program Multiple Data (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, 16]. 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 processor, 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 synchronized.

X10 is the first of the second generation of PGAS languages. It is a modern object-oriented programming language that introduces new constructs that significantly simplify scale out programming. 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, such as big data analytics.

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, [9, 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) and function literals, 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 overloading operator 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 can be dynamically created. Activities are 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. A distributed termination construct `finish` enables code to execute after all activities in the given statement have terminated, thus ensuring that all their side-effects have already taken place. An activity may shift to another place to execute a statement block. X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale. Multiple memory locations in multiple places cannot be accessed atomically. *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. `DistArray`s, distributed arrays, may be distributed across multiple places and support parallel collective operations. A novel exception flow model ensures that exceptions thrown by asynchronous activities can be caught at a suitable parent activity. Asynchronous initialization of variables is supported. Linking with native code is supported.

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 languages such as Scala. Programmers write X10 code by defining containers for data and behavior called *classes* (§8) and *structs* (§9), often abstracted as *interfaces* (§7). X10 provides inheritance and subtyping in fairly traditional ways.

Example:

`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; val 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 containers: *classes* (§8) and *structs* (§9). 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 and objects `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 (§10) 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 `Long`. `of4` takes an `Long`-to-`Long` 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:Long) => i*i;
val of4 = (f: (Long)=>Long) => f(4);
val fourSquared = of4(square);
```

Functions are used extensively in X10 programs. For example, a common way to construct and initialize an `Rail[Long]` – that is, a fixed-length one-dimensional array of numbers, like an `long[]` 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 i^{th} element. The following code constructs a 1-dimensional rail initialized to the squares of 0,1,...,9: `r(0) == 0`, `r(5)==25`, etc.

```
val r : Rail[Long] = new Rail[Long](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.

```
abstract class Mat(rows:Long, cols:Long) {
 static type Mat(r:Long, c:Long) = Mat{rows==r&&cols==c};
 abstract operator this + (y:Mat(this.rows,this.cols))
                :Mat(this.rows, this.cols);
 abstract operator this * (y:Mat) {this.cols == y.rows}
                :Mat(this.rows, y.cols);
```

The following code typechecks (assuming that `makeMat(m,n)` is a function which creates an `m`×`n` matrix). However, an attempt to compute `axb1 + bxc` or `bxc * axb1` would result in a compile-time type error:

```
static def example(a:Long, b:Long, c:Long) {
  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;
  //ERROR: val wrong1 = axb1 + bxc;
  //ERROR: val wrong2 = bxc * axb1;
}
```

The "little bit of care" shows off many of the features of constrained types. The `(rows:Long, cols:Long)` in the class definition declares two properties, `rows` and `cols`.[1]

A constrained type looks like `Mat{rows==r && cols==c}`: a type name, followed by a Boolean expression in braces. The `type` declaration on the second line makes `Mat(r,c)` be a synonym for `Mat{rows==r && 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. In particular, constructors can have constrained return types to provide specific information about the constructed values.

[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 *.

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.

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[Long]`, a list of non-zero integers `List[Long{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[T](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[Long]` is the type of lists of element type `Long`, `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 Long`. The types can be constrained: `n as Long{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* (§13). 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 *values*(§8.1) through the execution of lightweight threads called *activities*(§14). An *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;F)` can be used, to move part of the activity to `q`. `F` is a specification of what information will be sent to `q` for use by that part of the computation. It is easy to compute across multiple places, but the expensive operations (*e.g.*, those which require communication) are readily visible in the code.

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 S`, which starts up a new activity running `S`. It does not wait for the new activity to finish; there is a separate statement (`finish`) to do that.

2.4 Distributed heap management

X10 is the language for parallel and distributed computing, which is based on the APGAS (Asynchronous Partitioned Global Address Space) programming model. In (A)PGAS, the address space is partitioned into multiple semi-spaces. The semi-space is called *place* in X10. In Managed X10 (X10 on Java VMs), a place is represented as a single Java VM and the semi-space is mapped to the heap of the Java VM.

X10 supports garbage collection. Objects in a local heap (local objects) are collected with (local) garbage collection and there is no way to explicitly free them. The reference to local objects is called *local reference*.

In addition, X10 has another type of reference called *remote reference*. Unlike local reference, remote reference can reference objects at both local and remote places.

With remote reference, an activity (something like thread, it runs on a place at a time but it can move itself to different places) can access objects at a remote place (remote objects) when the activity has moved to the remote place. The place where an object is created is the home place of the object and it does not change for the lifetime.

To guarantee an activity can access remote objects at their home place, the objects with remote reference are protected from (local) garbage collection at their home place even if they have no local reference. Objects can be garbage collected only when they have neither local nor remote reference. The garbage collection that takes care of remote reference is called distributed garbage collection and it is supported in Managed X10.

Distributed garbage collection in Managed X10 [8] tracks the lifetime of remote reference with reference counting. When the local garbage collection at a remote place detects the remote reference is no longer needed at the place, the count is decremented. When the count becomes zero, the local garbage collection at the home place is ready to collect the referenced object in the ordinary way.

This mechanism works in most cases, but when there is unbalance in heap allocation rate between places, there is a risk of out of memory error at a frequently allocating place. This is because remote reference from infrequently allocating (i.e. infrequently garbage collected) places could retain remotely referenced objects longer than needed.

To avoid the out of memory error even with unbalanced heap allocation rate, there is a way to explicitly release remote reference.

A single call of `PlaceLocalHandle.destroy()` (`PlaceLocalHandle` is an X10 type that bundles multiple remote references to the objects at different places) releases all remote references immediately, thus the local garbage collection at each place becomes ready to collect the referenced object in the ordinary way. It can be called at the point where the all objects referenced by the handle are no longer needed to be accessible with the handle. Local reference to the object at each place won't be affected.

2.5 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 static method `Clock.advanceAll` 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 properly are guaranteed not to suffer from deadlock.

2.6 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.

2.7 Annotations

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

2.8 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.

2.9 Summary and future work

2.9.1 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 dynamically 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.

2.9.2 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*, with certain exceptions given in §4.15.

Static type safety guarantees that every location contains only values whose dynamic type agrees with the location's static type. The compiler allows a choice of how to handle method calls. In strict mode, method calls are statically checked to be permitted by the static types of operands. In lax mode, dynamic checks are inserted when calls may or may not be correct, providing weaker static correctness guarantees but more programming convenience.

Memory safety guarantees that an object may only access memory within its representation, and other objects it has a reference to. X10 does not permit pointer arithmetic, and bound-checks array accesses dynamically if necessary. X10 uses garbage collection to collect objects no longer referenced by any activity. 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 word of memory has previously been written into that word.

X10 programs that use only the common, specified clock idioms and unconditional atomic blocks are guaranteed not to deadlock. Unconditional atomic blocks are non-blocking, hence cannot introduce deadlocks. Many concurrent programs can be shown to be determinate (hence race-free) statically.

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

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

2.9.3 Conclusion

X10 is considerably higher-level than thread-based languages in that it supports dynamically spawning 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.

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

3 Lexical and Grammatical structure

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators, all of them composed of Unicode characters in the UTF-8 (or US-ASCII) encoding.

3.1 Whitespace

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

3.2 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.

3.3 Identifiers

Identifiers consist of a single letter followed by zero or more letters or digits. The letters are the ASCII characters `a` through `z`, `A` through `Z`, and `_`. Digits are defined as the ASCII characters `0` through `9`. Case is significant; `a` and `A` are distinct identifiers, `as` is a keyword, but `As` and `AS` are identifiers. (However, case is insignificant in the hexadecimal numbers, exponent markers, and type-tags of numeric literals – 0xbabe = 0XBABE.)

In addition, any string of characters may be enclosed in backquotes ` to form an identifier – though the backquote character itself, and the backslash character, must be quoted by a backslash if they are to be included. This allows, for example, keywords to be used as identifiers. The following are backquoted identifiers:

```
‘while‘, ‘!‘, ‘(unbalanced(‘,  ‘\‘\\‘, ‘0‘
```

Certain back ends and compilation options do not support all choices of identifier.

3.4 Keywords

X10 uses the following keywords:

`abstract`	`as`	`assert`	`async`	`at`
`athome`	`ateach`	`atomic`	`break`	`case`
`catch`	`class`	`clocked`	`continue`	`def`
`default`	`do`	`else`	`extends`	`false`
`final`	`finally`	`finish`	`for`	`goto`
`haszero`	`here`	`if`	`implements`	`import`
`in`	`instanceof`	`interface`	`native`	`new`
`null`	`offer`	`offers`	`operator`	`package`
`private`	`property`	`protected`	`public`	`return`
`self`	`static`	`struct`	`super`	`switch`
`this`	`throw`	`transient`	`true`	`try`
`type`	`val`	`var`	`void`	`when`
`while`				

Keywords may be used as identifiers by enclosing them in backquotes: `‘new‘` is an identifier, `new` is a keyword but not an identifier.

Note that the primitive type names are not considered keywords.

3.5 Literals

Briefly, X10 v2.4 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:

$$
\begin{array}{rcl}
d & = & \text{one or more decimal digitsonly starting with 0 if it is 0} \\
d_8 & = & \text{one or more octal digits} \\
d_{16} & = & \text{one or more hexadecimal digits, using a-f or A-F for 10-15} \\
i & = & d \mid \texttt{0}d_8 \mid \texttt{0x}d_{16} \mid \texttt{0X}d_{16} \\
s & = & \text{optional + or -} \\
b & = & d \mid d. \mid d.d \mid .d \\
x & = & (\texttt{e} \mid \texttt{E})sd \\
f & = & bx
\end{array}
$$

- `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 si`n` or si`N`. *E.g.*, `123n`, `-321N` are decimal `Int`s, `0123N` and `-0321n` are octal `Int`s, and `0x123n`, `-0X321N`, `0xBEDN`, and `0XEBECN` are hexadecimal `Int`s.
- `Long` literals have the form si, si`l` or si`L`. *E.g.*, `1234567890` and `0xBABEL` are `Long` literals.
- `UInt` literals have the form i`un` or i`nu`, or capital versions of those. *E.g.*, `123un`, `0123un`, and `0xBEAUN` are `UInt` literals.
- `ULong` literals have the form i`u`, i`ul` or i`lu`, or capical versions of those. For example, `123u`, `0124567012u`, `0xFU`, `OXba1efu`, and `0xDecafC0ffeefU` are `ULong` literals.
- `Short` literals have the form si`s` or si`S`. *E.g.*, 414S, `OxACES` and `7001s` are short literals.
- `UShort` literals form i`us` or i`su`, or capital versions of those. For example, `609US`, `107us`, and `OxBeaus` are unsigned short literals.
- `Byte` literals have the form si`y` or si`Y`. (The letter `B` cannot be used for bytes, as it is a hexadecimal digit.) `50Y` and `OxBABY` are byte literals.
- `UByte` literals have the form i`uy` or i`yu`, or capitalized versions of those. For example, `9uy` and `OxBUY` are `UByte` literals.
- `Float` literals have the form sf`f` or sf`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 sf[1], sf`D`, and sf`d`. *E.g.*, `0.0`, `0e100`, `1.3D`, `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;

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

  - `'\"'`, 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*$>$ 0377.

- `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!"`, `""`.

3.6 Separators

X10 has the following separators and delimiters:

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

3.7 Operators

X10 has the following operator, type constructor, and miscellaneous symbols. (`?` and `:` comprise a single ternary operator, but are written separately.)

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

The precedence of the operators is as follows. Earlier rows of the table have higher precedence than later rows, binding more tightly. For example, `a+b*c<d` parses as `(a+(b*c))<d`, and `-1 as Byte` parses as `-(1 as Byte)`.

postfix `()`
`as T`, postfix ++, postfix `--`
unary `-`, unary +, prefix ++, prefix `--`
unary operators `!`, `~`, `^`, *, `|`, `&`, `/`, and `%`
`..`
`*` `/` `%` `**`
`+` `-`
`<<` `>>` `>>>` `->` `>-` `-<` `<-` `!`
`>` `>=` `<` `<=` `instanceof`
`==` `!=` `!` `!~`
`&`
`^`
`|`
`&&`
`||`
`? :`
=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=

3.8 Grammatical Notation

In this manual, ordinary BNF notation is used to specify grammatical constructions, with a few minor extensions. Grammatical rules look like this:

Adj ::= *Adv*$^?$ `happy`
| *Adv*$^?$ `sad`
Adv ::= `very`
| *Adv Adv*

Terms in *italics* are called **non-terminals**. They represent kinds of phrases; for example, *ForStmt* (20.74)[2] describes all `for` statements. Equation numbers refer to the full X10 grammar, in §20. The small example has two non-terminals, *Adv* and *Adj*.

Terms in `fixed-width font` are **terminals**. They represent the words and symbols of the language itself. In X10, the terminals are the words described in this chapter.

A single grammatical rule has the form $A ::= X_1X_2...X_n$, where the X_i's are either terminals or nonterminals. This indicates that the non-terminal A could be an instance of X_1, followed by an instance of X_2, ..., followed by an instance of X_n. Multiple rules for the same A are allowed, giving several possible phrasings of A's. For brevity, two rules with the same left-hand side are written with the left-hand side appearing once, and the right-hand sides separated by |.

In the *Adj* example, there are two rules for *Adv*, $Adv ::=$ `very` and $Adv ::= Adv\ Adv$. So, an adverb could be `very`, or (by three uses of the rule) `very very`, or, one or more `very`s.

The notation $A^?$ indicates an optional A. This is an ordinary non-terminal, defined by the rules:

[2] Grammar rules are given in §20, and referred to by equation number in that section.

$A^?$ ::=
| A

The first rule says that $A^?$ can amount to nothing; the second, that it can amount to an A. This concept shows up so often that it is worth having a separate notation for it. In the *Adj* example, an adjective phrase may be preceded by an optional adverb. Thus, it may be `happy`, or `very happy`, or `very very sad`, etc.

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.

X10 supports four kinds of values, *objects*, *struct values*, *functions*, and `null`. Objects are in the grand tradition of object-oriented languages, and the heart of most X10 computations. They are instances of *classes* (§8); they hold zero or more data fields that may be mutable. They respond to methods, and can inherit behavior from their superclass.

Struct values are similar to objects, though more restricted in ways that make them more efficient in space and time. Their fields cannot be mutable, and, although they respond to methods, they do not inherit behavior. They are instances of struct types (§9).

Together, objects and struct values are called *containers*, because they hold data.

Functions, called closures, lambda-expressions, and blocks in other languages, are instances of *function types* (§10). A function has zero or more *formal parameters* (or *arguments*) and a *body*, which is an expression that can reference the formal parameters and also other variables in the surrounding block. For instance, `(x:Long)=>x*y` is a unary integer function which multiplies its argument by the variable `y` from the surrounding block. Functions may be freely copied from place to place and may be repeatedly applied.

Finally, `null` is a constant, often found as the default value of variables of object type. While it is not an object, it may be stored in variables of class type – except for types which have a constraint (§4.5) which specifically excludes `null`.

These runtime values are classified by *types*. Types are used in variable declarations (§12.2), coercions and explicit conversions (§11.9.1), object creation (§11.21), static state and method accessors (§11.4), generic classes, structs, interfaces, and methods (§4.3), type definitions (§4.4), closures (§10), class, struct, and interface declarations (§8.1.2), subtyping expressions (§11.25), and `instanceof` and `as` expressions (§11.24).

The basic relationship between values and types is the *is a value in* relation: `e` is a value in `T`. We also often say "`e` has type `T`" to or "`e` is an element of type `T`". For example, `1` has type `Long` (the type of all integers representible in 64 bits). It has the more general

type `Any` (since all entitites have type `Any`). Furthermore, it has such types as "Nonzero integer" and "Integer equal to one", and many others. These types are expressable in X10 using constrained types (§4.5). `Long{self!=0}` is the type of `Long`s `self`[1] which are not equal to zero, and `Long{self==1}` is the type of the `Long`s which are equal to one.

The basic relationship between types is *subtyping*: `T <: U` holds if every value in `T` is also a value ind `U`. Two important kinds of subtyping are *subclassing* and *strengthening*. Subclassing is a familiar notion from object-oriented programming. Here we use it to refer to the relationship between a class and another class it extends or an interface (§7) it implements. For instance, in a class hierarchy with classes `Animal` and `Cat` such that `Cat` extends `Mammal` and `Mammal` extends `Animal`, every instance of `Cat` is by definition an instance of `Animal` (and `Mammal`). We say that `Cat` is a subclass of `Animal`, or `Cat <: Animal` by subclassing. If `Animal` implements `Thing`, then `Cat` also implements `Thing`, and we say `Cat <: Thing` by subclassing.

Strengthening is an equally familiar notion from logic. The instances of `Long{self == 1}` are all elements of `Long{self != 0}` as well, because `self == 1` logically implies `self != 0`; so `Long{self == 1} <: Long{self !=0}` by strengthening. X10 uses both notions of subtyping. See §4.9 for the full definition of subtyping in X10.

4.1 Type System

X10 has several sorts of types. In this section, `S`, `T`, and $\texttt{T}_i$ range over types. `X` ranges over type variables, `M` and $\texttt{x}_i$ over identifiers, `c` over constraint expressions (§4.5), and $\texttt{e}_i$ over expressions. For compactness, slanted brackets are used to indicate optional elements. [2]

Type	::=	`T`	
`T`	::=	`M`	*(1)*
	|	`X`	*(2)*
	|	`M [`$\texttt{T}_1$ `, ... ,` $\texttt{T}_n$ `]`	*(3)*
	|	$\texttt{T}_1$ `.` $\texttt{T}_2$	*(4)*
	|	`F`	*(5)*
	|	`M` *`[`* `[`$\texttt{T}_1$ `, ... ,` $\texttt{T}_n$ `]` *`]`* `(` e_1 `,...,` e_k `)`	*(6)*
	|	`T{c}`	*(7)*
`F`	::=	`(` *`[`*$\texttt{x}_1$ *`:]`* $\texttt{T}_1$ `,...,` *`[`*$\texttt{x}_n$ *`:]`* $\texttt{T}_n$ `)` *`[`* `{` c `}` *`]`* `=>` `T`	
	|	`(` *`[`*$\texttt{x}_1$ *`:]`* $\texttt{T}_1$ `,...,` *`[`*$\texttt{x}_n$ *`:]`* $\texttt{T}_n$ `)` *`[`* `{` c `}` *`]`* `=>` `void`	

A type given by (1) is an identifier `M`, like `Point`, `Long`, or `long`. It refer to a unit – a class, struct type, or interface, (§4.2). Or, it can refer to a name defined by a `type` statement (§4.4);

[1] X10 automatically uses the identifier `self` for the element of the type being constrained.

[2] The actual grammar, as given in §20, is slightly more intricate for technical reasons. The set of types is the same, however, and this grammar is better for exposition.

Example: `String` *refers to the standard class of strings,* `Long` *to the standard struct type of integers, and* `Any` *to the interface that describes all X10 values.* `long` *is an alias for the type* `Long`*, for the comfort of programmers used to other languages in the C family.*

A type of the form (2), a type variable `X`, refers to a parameter type of a generic (parameterized) type, as described in §4.3.

Example: *The class* `Pair[X]` *below provides a simplistic way to keep two things of the same type together.*[3] `Pair[Long]` *holds two integers;* `Pair[Pair[String]]` *holds two pairs of strings. Within the definition of* `Pair`*, the type variable* `X` *is the parameter type of* `Pair` *– that is, the type which this pair is a pair of.*

```
class Pair[X]{
  public val first : X;
  public val second: X;
  public def this(f:X, s:X) {first = f; second = s;}
}
```

A type of form (3), `M[T,U]`, is a use of a generic type, also described in §4.3, or a generic `type`-defined type without value parameters (§4.4). The types inside the brackets are the actual parameters corresponding to the formal parameters of the parameterized type `M`. `Pair[Long]`, above, is an example of a use of the generic type `Pair`.

A type of form (4), `T.U`, is a qualified type: a unit `U` appearing inside of the unit `T`, as described in §8.14.

Example:

```
class Outer {
  class Inner { /* ... */ }
}
```

then `(new Outer()).new Inner()` *creates a value of type* `Outer.Inner`.

A type of form (5), `F`, such as `(x:Long)=>Long`, is a function type. Its values are functions, *e.g.*, the squaring function taking integers to integers. Function types are described in §4.6, and computing with functions is described in §10.

Example: `square` *is the squaring function on integers. It is used in the* `assert` *line.*

```
val square : (x:Long)=>Long
           = (x:Long)=>x*x;
assert square(5) == 25;
```

A term of form (6), such as `M[T](e)`, is an instance of a parameterized type definition. Such types may be parameterized by both types and values. This is described in §4.4.

Example: `Array[Long](1)` *is the type of one-dimensional arrays of integers. It has one type parameter giving the type of element, here* `Long`*. It has one value parameter*

[3] In practice, most people would use an `Rail` rather than making a new `Pair` class.

giving the number of dimensions, here `1`. `Region(1)` *is the type of one-dimensional regions of points (§16.4.1).*

In the function types (6), the variable names are bound. As with all bound variables in X10, they can be renamed. So, for example, the types `(x:Long)=>Long{self!=x}` and `(y:Long)=>Long{self!=y}` are equivalent, as they differ by nothing but the names of bound variables. This is more visible with types than with, say, methods or functions, because we can test equality of types.

Furthermore, if a variable `x` does not appear anywhere in a function type `F` save as an argument name, it (and its "`:`") can be omitted. *E.g.*, the types `(x:Long)=>Long` and `(Long)=>Long` are equivalent.

Example:

```
val f : (x:Long)=>Long{self!=x} = (x:Long) => (x+1) as Long{self!=x};
val g : (y:Long)=>Long{self!=y} = f;
val t : (x:Long)=>Long          = (x:Long) => x;
val u : ( Long )=>Long          = t;
```

A term of form (7), `T{c}`, is a type whose values are the values of type `T` for which the constraint `c` is true. This is described in §4.5.

Example: *A variable of class* `Point`*, unconstrained, can contain* `null`*:*

```
var gotNPE: Boolean = false;
val p : Point = null;
try {
  val q = p * 2; // method invocation, NPE
}
catch(NullPointerException) {
  gotNPE = true;
}
assert gotNPE;
```

A suitable constraint on that type will prevent a `null` *from ever being assigned to the variable. The variable* `self`*, in a constraint, refers to the value being constrained, so the constraint* `self != null` *means "which is not null". So, adding a* `{self!=null}` *constraint to* `Point` *results in a compile-time error, rather than a runtime null pointer exception.*

```
// ERROR: p : Point{self!=null} = null;
```

4.2 Unit Types: Classes, Struct Types, and Interfaces

Most X10 computation manipulates values via the *unit* types: classes, struct types, and interfaces. These types share a great deal of structure, though there are important differences.

4.2.1 Class types

A *class declaration* declares a *class type* (§8), giving its name, behavior, and data. It may inherit from zero or one *parent* class. It may also implement zero or more interfaces, each one of which becomes a supertype of it.

Example: *The* `Position` *class below could describe the position of a slider control. The* `example` *method uses* `Position` *as a type.* `Position` *is a subtype of the type* `Poser`.

```
interface Poser {
  def pos():Long;
  }
class Position implements Poser {
  private var x : Long = 0;
  public def move(dx:Long) { x += dx; }
  public def pos() : Long = x;
  static def example() {
    var p : Position;
  }
}
```

The null value, represented by the literal `null`, is a value of every class type `C`. The type whose values are all instances of `C` except `null` can be defined as `C{self != null}`.

4.2.2 Struct Types

A *struct declaration* (§9) introduces a *struct type* containing all instances of the struct. Struct types can include nearly all the features that classes have. They can implement interfaces, which become their supertypes just as for classes; but they do not have superclasses, and cannot extend anything.

Example: *The* `Coords` *struct gives an immutable position in 3-space. It is used as a type in* `example()`*:*

```
struct Position {
  public val x:Double; public val y:Double; public val z:Double;
  def this(x:Double, y:Double, z:Double) {
     this.x = x; this.y = y; this.z = z;
  }
  static def example(p: Position, q: Rail[Position]) {
     var r : Position = p;
  }
}
```

4.2.3 Interface types

An *interface declaration* (§7) defines an *interface type*, specifying a set of instance method signatures and property method signatures which must be provided by any container declared to implement the interface. They can also declare static `val` fields, which are provided to all units implementing or extending the interface. They do not have code, and cannot implement anything. An interface may extend multiple interfaces. Each interface it extends becomes one of its superclasses.

Example: `Named` *and* `Mobile` *are interfaces, each specifying a single method.* `Person` *and* `NamedPoint` *are subtypes of both of them. They are used as types in the* `example` *method.*

```
interface Named {
  def name():String;
}
interface Mobile {
  def where():Long;
  def move(howFar:Long):void;
}
interface NamedPoint extends Named, Mobile {}
class Person implements Named, Mobile {
   var name:String; var pos: Long;
   public def name() = this.name;
   public def move(howFar:Long) { pos += howFar; }
   public def where() = this.pos;
   public def example(putAt:Mobile) {
      this.pos = putAt.where();
   }
}
```

4.2.4 Properties

Classes, interfaces, and structs may have *properties*, specified in parentheses after the type name. Properties are much like public `val` instance fields. They have certain restrictions on their use, however, which allows the compiler to understand them much better than other public `val` fields. In particular, they can be used in types. *E.g.*, the number of elements in a rail is a property of the rail, and an X10 program can specify that two rails have the same number of elements.

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.*

```
class Coords(x: Long, y: Long) {
  def this(x: Long, y: Long) :
    Coords{self.x==x, self.y==y} {
```

```
    property(x, y);
  }

  def move(dx: Long, dy: Long) = new Coords(x+dx, y+dy);
}
```

Properties of `self` can be used in constraints. This places certain restrictions on how properties can be used, but allows a great deal of compile-time constraint checking. For a simple example, `new Coords(0,0)` is known to be an instance of `Coords{self.x==0}`. Details of this substantial topic are found in §4.5.

4.3 Type Parameters and Generic Types

A class, interface, method, or type definition may have type parameters. Type parameters can be used as types, and will be bound to types on instantiation. For example, a generic stack class may be defined as `Stack[T]{...}`. Stacks can hold values of any type; *e.g.*, `Stack[Long]` is a stack of longs, and `Stack[Point {self!=null}]` is a stack of non-null `Point`s. Generics *must* be instantiated when they are used: `Stack`, by itself, is not a valid type. Type parameters may be constrained by a guard on the declaration (§4.4, §8.4.6,§10.3).

A *generic class* (or struct, interface, or type definition) is a class (resp. struct, interface, or type definition) declared with $k \geq 1$ type parameters. A generic class (or struct, interface, or type definition) can be used to form a type by supplying k types as type arguments within `[...]`.

Example: `Bottle[T]` *is a generic class. A* `Bottle[T]` *can hold a value of type* `T`*; the variable* `yup` *in* `example()` *is of type* `Bottle[Boolean]` *and thus can hold a* `Boolean`*. Hoever,* `Bottle` *alone is not a type.*[4]

```
class Bottle[T] {
  var contents : T;
  public def this(t:T) { contents = t;  }
  public def putIn(t:T) { contents = t; }
  public def get() = contents;
  static def example() {
    val yup : Bottle[Boolean] = new Bottle[Boolean](true);
    //ERROR: var nope : Bottle = null;
  }
}
```

A class (whether generic or not) may have generic methods.

Example: `NonGeneric` *has a generic method* `first[T](x:List[T])`*. An invocation of such a method may supply the type parameters explicitly (*e.g.*,* `first[Long](z)`*).*

[4] By contrast, in Java, the equivalent of `Bottle` alone *would* be a type, via type erasure of generics.

*In certain cases (*e.g.*,* `first(z)`*) type parameters may be omitted and are inferred by the compiler (§4.12).*

```
class NonGeneric {
  static def first[T](x:List[T]):T = x(0);
  def m(z:List[Long]) {
    val f = first[Long](z);
    val g = first(z);
    return f == g;
  }
}
```

Limitation: X10 v2.4's C++ back end requires generic methods to be static or final; the Java back end can accomodate generic instance methods as well.

4.4 Type definitions

A type definition can be thought of as a type-valued function, mapping type parameters and value parameters to a concrete type.

TypeDefDecln	::=	*Mods*$^?$ `type` *Id TypeParams*$^?$ *Guard*$^?$ `=` *Type* `;`	*(20.170)*
	|	*Mods*$^?$ `type` *Id TypeParams*$^?$ `(` *FormalList* `)` *Guard*$^?$ `=` *Type* `;`	
TypeParams	::=	`[` *TypeParamList* `]`	*(20.176)*
Formals	::=	`(` *FormalList*$^?$ `)`	*(20.80)*
Guard	::=	*DepParams*	*(20.83)*

During type-checking the compiler replaces the use of such a defined type with its body, substituting the actual type and value parameters in the call for the formals. This replacement is performed recursively until the type no longer contains a defined type or a predetermined compiler limit is reached (in which case the compiler declares an error). Thus, recursive type definitions are not permitted.

Type definitions are considered applicative and not generative – they do not define new types, only aliases for existing types.

Type definitions may have guards: an invocation of a type definition is illegal unless the guard is satisified when formal types and values are replaced by the actual parameters.

Type definitions may be overloaded: two type definitions with the same name are permitted provided that they have a different number of type parameters or different number or type of value parameters. The rules for type definition resolution are identical to those for method resolution.

However, `T()` is not allowed. If there is an argument list, it must be nonempty. This avoids a possible confusion between `type T = ...` and `type T() = ...`.

A type definition for a type `T` can appear:

- As a top-level definition in a file named `T.x10`; or

- As a static member in a container definition; or
- In a block statement.

Use of type definitions in constructor invocations If a type definition has no type parameters and no value parameters and is an alias for a container type, a `new` expression may be used to create an instance of the class using the type definition's name. Similarly, a parameterless alias for an interface can be used to construct an instance of an anonymous class. 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`.

Example: *The names of the class* `Cont[X]` *and the interface* `Inte[X]` *can be used to create an object* `a` *of type* `Cont[Long]`*, and an object* `b` *which implements* `Inte[Long]`*. The two types may be given aliases* `A` *and* `B`*, which may then be used in more compact expressions to construct objects* `aa` *and* `bb` *of the same types.*

```
class ConstructorExample {
  static class Cont[X]{}
  static interface Inte[X]{
     def meth():X;
   }
  public static def example() {
    val a = new Cont[Long]();
    val b = new Inte[Long](){public def meth()=3;};
    type A = Cont[Long];
    val aa = new A();
    type B = Inte[Long];
    val bb = new B(){public def meth()=4;};
  }
}
```

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

4.4.1 Motivation and use

The primary purpose of type definitions is to provide a succinct, meaningful name for complex types and combinations of types. With value arguments, type arguments, and constraints, the syntax for X10 types can often be verbose. For example, a non-null list of non-null strings is
`List[String{self!=null}]{self!=null}`.

We could name that type:

```
static type LnSn = List[String{self!=null}]{self!=null};
```

Or, we could abstract it somewhat, defining a type constructor `Nonnull[T]` for the type of `T`'s which are not null:

```
class Example {
  static type Nonnull[T]{T isref}  = T{self!=null};
  var example : Nonnull[Example] = new Example();
}
```

Type definitions can also refer to values, in particular, inside constraints. The type of `n`-element `Array[Long](1)`s is `x10.regionarray.Array[Long]{self.rank==1 && self.size == n}` but it is often convenient to give a shorter name:

```
type Vec(n:Long) = x10.regionarray.Array[Long]{self.rank==1, self.size == n};
var example : Vec(78L);
```

The following examples are legal type definitions,

```
import x10.util.*;
import x10.regionarray.*;
class TypeExamples {
  static type StringSet = Set[String];
  static type MapToList[K,V] = Map[K,List[V]];
  static type Long(x: Long) = Long{self==x};
  static type Dist(r: Long) = Dist{self.rank==r};
  static type Dist(r: Region) = Dist{self.region==r};
  static type Redund(n:Long, r:Region){r.rank==n}
      = Dist{rank==n && region==r};
}
```

The following code illustrates that type definitions are applicative rather than generative. `B` and `C` are both aliases for `String`, rather than new types, and so are interchangeable with each other and with `String`. Similarly, `A` and `Long` are equivalent.

```
def someTypeDefs () {
  type A = Long;
  type B = String;
  type C = String;
  a: A = 3;
  b: B = new C("Hi");
  c: C = b + ", Mom!";
  }
```

4.5 Constrained types

Basic types, like `Long` and `List[String]`, provide useful descriptions of data.

However, one frequently wants to say more. One might want to know that a `String` variable is not `null`, or that a matrix is square, or that one matrix has the same number of columns as another has rows (so they can be multiplied). In the multicore setting, one might wish to know that two values are located at the same processor, or that one is located at the same place as the current computation.

In most languages, there is simply no way to say and check these things statically. Programmers must made do with comments, `assert` statements, and dynamic tests. X10 programs can do better, with *constraints* on types, and guards on class, method and type definitions.

A constraint expression is a Boolean expression `e` of a quite limited form (§4.5.2). . A constraint expression `c` may be attached to a basic type `T`, giving a *constrained type* `T{c}`. The values of type `T{c}` are the values of `T` for which `c` is true. Constraint expressions also serve as guards on methods (§8.4) and functions (§10.3), and invariants on unit types (§8.9.

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

Example:

- `Long{self != 0}` *is the type of non-zero* `Long`*s.*
- `Long{self == 0}` *is the type of* `Long`*s which are zero.*
- `Long{self != 0, self != 1}` *is the type of* `Long`*s which are neither zero nor one.*
- `Long{self == 0, self == 1}` *is the type of* `Long`*s which are both zero and one. There are no such values, so it is an empty type.*
- `String{self != null}` *is the type of non-null strings.*
- *Suppose that* `Matrix` *is a matrix class with properties* `rows` *and* `cols`. `Matrix{self.rows == self.cols}` *is the type of square matrices.*
- *One way to say that* `a` *has the same number of columns that* `b` *has rows (so that* `a*b` *is a valid matrix product), one could say:*

```
val a : Matrix = someMatrix() ;
var b : Matrix{b.rows == a.cols} ;
```

`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*. If the constraint is omitted, it is `true`—that is, the base type is unconstrained.

Constraints may refer to immutable values in the local environment:

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

In a `val` variable declaration, the variable itself is in scope in its type, and can be used in constraints.

Example: *For example,* `val nz: Long{nz != 0} = 1;` *declares a non-zero variable* `nz`*. In this case,* `nz` *could have been declared as* `val nz: Long{self != 0} = 1`.

4.5.1 Examples of Constraints

Example of entailment and subtyping involving constraints.

- `Long{self == 3} <: Long{self != 14}`. The only value of `Long{self ==3}` is 3. All integers but 14 are members of `Long{self != 14}`, and in particular 3 is.
- Suppose we have classes `Child <: Person`, and `Person` has a `ssn:Long` property. If `rhys : Child{ssn == 123456789}`, then `rhys` is also a `Person`. `rhys`'s `ssn` field is the same, `123456789`, whether `rhys` is regarded as a `Child` or a `Person`. Thus, `rhys : Person{ssn==123456789}` as well. So,

  ```
  Child{ssn == 123456789} <: Person{ssn == 123456789}.
  ```

- Furthermore, since `123456789 != 555555555`, is is clear that `rhys : Person{ssn != 555555555}`. So,

  ```
  Child{ssn == 123456789} <: Person{ssn != 555555555}.
  ```

- `T{e} <: T` for any type `T`. That is, if you have a value `v` of some base type `T` which satisfied `e`, then `v` is of that base type `T` (with the constraint ignored).
- If `A <: B`, then `A{c} <: B{c}` for every constraint `{c}` for which `A{c}` and `B{c}` are defined. That is, if every `A` is also a `B`, and `a : A{c}`, then `a` is an `A` and `c` is true of it. So `a` is also a `B` (and `c` is still true of it), so `a : B{c}`.

Constraints can be used to express simple relationships between objects, enforcing some class invariants statically. For example, in geometry, a line is determined by two *distinct* points; a `Line` struct can specify the distinctness in a type constraint:[5]

[5] We call them `Position` to avoid confusion with the built-in class `Point`. Also, `Position` is a struct rather than a class so that the non-equality test `start != end` compares the coordinates. If `Position` were a class, `start != end` would check for different `Position` objects, which might have the same coordinates.

```
struct Position(x: Long, y: Long) {}
struct Line(start: Position, end: Position){start != end}
  {}
```

Extending this concept, a `Triangle` can be defined as a figure with three line segments which match up end-to-end. Note that the degenerate case in which two or three of the triangle's vertices coincide is excluded by the constraint on `Line`. However, not all degenerate cases can be excluded by the type system; in particular, it is impossible to check that the three vertices are not collinear.

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

The `Triangle` class automatically gets a ternary constructor which takes suitably constrained `a`, `b`, and `c` and produces a new triangle.

A constrained type may be constrained further: the type `S{c}{d}` is the same as the type `S{c,d}`. Multiple constraints are equivalent to conjoined constraints: `S{c,d}` in turn is the same as `S{c && d}`.

4.5.2 Syntax of constraints

Only a few kinds of expressions can appear in constraints. For fundamental reasons of mathematical logic, the more kinds of expressions that can appear in constraints, the harder it is to compute the essential properties of constrained types – in particular, the harder it is to compute `A{c} <: B{d}` or even `E : T{c}`. It doesn't take much to make this basic fact undecidable. In order to make sure that it stays decidable, X10 places stringent restrictions on constraints.

Only the following forms of expression are allowed in constraints.

Value expressions in constraints may be:

1. Literal constants, like `3` and `true`;
2. Accessible, immutable (`val`) variables and parameters;
3. `this`, if the constraint is at a point in the program where `this` is defined, but not in `extends` or `implements` clauses or class invariants;
4. `here`, if the constraint is at a point in the program where `here` is defined;
5. `self`;
6. A field selection expression `t.f`, where `t` is a value expression allowed in constraints, and `f` is a field of `t`'s type. If `t` is `self`, then `f` must be a property, not an arbitrary field.

7. Invocations of property methods, `p(a,b,...,c)` or `a.p(b,c,...d)`, where the receiver and arguments must be value expressions acceptable in constraints, as long as the expansion (*viz.*, the expression obtained by taking the body of the definition of `p`, and replacing the formal parameters by the actual parameters) of the invocation is allowed as a value expression in constraints.

For an expression `self.p` to be legal in a constraint, `p` must be a property. However terms `t.f` may be used in constraints (where `t` is a term other than `self` and `f` is an immutable field.)

Constraints may be any of the following, where all value expressions are of the forms which may appear in constraints:

1. Equalities `e == f`;
2. Inequalities of the form `e != f`;[6]
3. Conjunctions of Boolean expressions that may appear in constraints (but only in top-level constraints, not in Boolean expressions in constraints);
4. Subtyping and supertyping expressions: `T <: U` and `T :> U`;
5. Type equalities and inequalities: `T == U` and `T != U`;
6. Invocations of a property method, `p(a,b,...,c)` or `a.p(b,c,...d)`, where the receiver and arguments must be value expressions acceptable in constraints, as long as the expansion of the invocation is allowed as a constraint.
7. Testing a type for a default: `T haszero`.

Note that constraints on methods may include private, protected, or package-protected fields. It is possible to have a method whose guard cannot be directly checked, or even whose result type cannot be expressed as a clause in the program, at some call sites. Nonetheless, X10 uses a broader *internal* type representation, not limited by access rules, and can work with fields in types even though those fields cannot be used in executable code.

Example: *This phenomenon can be used to implement a form of compile-type capability checking. We give a minimal example, providing only security by obscurity: users unaware that the* `key` *method returns the required key will be unable to use the* `secret` *method. This approach can be strengthened to provide better security.*

The class `Keyed` *has a private field* `k`*. The method* `secret(q)` *can only be called when* `q==k`*. In a larger example,* `secret` *could be some priveleged behavior or secret, available only to callers with proper authority.*

At the call site in `Snooper`*,* `keyed.secret()` *is called. It can't be called as* `keyed.secret(keyed.k)`*, because* `k` *is a private field. It can't be called as* `keyed.secret(8)`*, even though*

[6] Currently inequalities of the form `e < f` are not supported.

`keyed.k==8`*, because there is no proof available that* `keyed.k==8` *— indeed, at this point in the code, the requirement that* `keyed.k==8` *cannot even be expressed in X10.*

However, the value of `keyed.k` *can be retrieved, using* `keyed.key()`*. The type of* `kk` *cannot be expressed in* `Snooper`*, because it refers to a private field of* `keyed`*. However, the compiler's internal representation is not bound by the rules of privacy, and can track the fact that* `kk` *is the same as* `keyed.k`*. So, the call* `keyed.secret(kk)` *succeeds.*

```
class Keyed {
  private val k : Long;
  public def this(k : Long) {
    this.k = k;
  }
  public def secret(q:Long){q==this.k} = 11;
  public def key():Long{self==this.k} = this.k;
}
class Snooper {
  public static def main(argv:Rail[String]) {
    val keyed : Keyed = new Keyed(8);
    //ERROR: keyed.secret(keyed.k);
    //ERROR: keyed.secret(8);
    val kk = keyed.key();
    keyed.secret(kk);
  }
}
```

Note: Constraints may not contain casts. In particular, comparisons of values of incompatible types are not allowed. If `i:Long`, then `i==0` is allowed as a constraint, but `i==0L` is an error, and `i as Long==0L` is outside of the constraint language.

Semantics of constraints

The logic of constraints is designed to allow a common and important X10 idiom:

```
class Thing(p:Long){}
static def example(){
   var x : Thing{x.p==3} = null;
}
```

That is, `null` must be an instance of `Thing{x.p==3}`. Of course, it cannot be the case that `null.p==3` — nor can it equal anything else. When evaluated at runtime, `null.p` must throw a `NullPointerException` rather than returning any value at all.

So, X10's logic of constraints — *unlike* the logic of runtime — allows `x`=`null` to satisfy `x.p==3`. Building this logic requires a few definitions.

The property graph, at an instant in an X10 execution, is the graph whose nodes are all objects in existence at that instance, plus `null`, with an edge from x to y if x is

an object with a property whose value is y. The rules for constructors guarantee that property graphs are acyclic, which is crucial for decidability.

As is standard in mathematical logic, we introduce the concept of a *valuation* v, which is a mapping from variable names to their values – in our case, nodes of an X10 property graph. A valuation v can be extended to values to all constraint formulas. The crucial definitions are:

$v($ `a.b....l.m == n.o....y.z` $) =$
`a`$=$`null` $\vee$ `a.b`$=$`null` $\vee$ ... `a.b....l`$=$`null`
$\vee$ `n`$=$`null` $\vee$ `n.o`$=$`null` $\vee$ ... `n.o....y`$=$`null`
$\vee$ $v($`a`$)$`.b....l.m` $=$ $v($`n`$)$`.o....y.`

$v($ `a.b....l.m != n.o....y.z` $) =$
`a`$=$`null` $\vee$ `a.b`$=$`null` $\vee$ ... `a.b....l`$=$`null`
$\vee$ `n`$=$`null` $\vee$ `n.o`$=$`null` $\vee$ ... `n.o....y`$=$`null`
$\vee$ $v($`a`$)$`.b....l.m` $\neq$ $v($`n`$)$`.o....y.`

For example, $v($`a.b==1`$)$ is true if either $v($`a`$) =$`null` or if $v($`a`$)$ is a container whose `b`-field is equal to 1. While such a valuation is perfectly well-defined, it has properties that need to be understood in light of the fact that == is *not* mathematical equality.[7] Given any valuation in which $v($`a`$) =$`null`, both $v($`a.b==1 && a.b==2`$)$ and $v($`a.b==1 && a.b!=1`$)$ are true. This does not contradict logic and mathematics, it does not imply that $v($`false`$)$ is true (it's not), and it does not assert that in X10 there is a number which is both 1 and 2. It simply reflects the fact that, while == is similar to mathematical equality in many respects, it is ultimately a different operation, and in constraints it is given a `null`-safe interpretation.

From this definition of valuation, we define *entailment* in the standard way. Given constraints `c` and `d`, we define `c` *entails* `d`, sometimes written `c` $\models$ `d`, if for all valuations v such that $v($`c`$)$ is true, $v($`d`$)$ is also true.

Limitation: Although nearly-contradictory conjunctions like `x.a==1 && x.a==2` entail `x==null`, X10's constraint solver does not currently use this rule. If you want `x==null`, write `x==null`.

Subtyping of constrained types is defined in terms of entailment. `S[S1,..., Sm]{c}` is a subtype of `T[T1,..., Tn]{d}` if `S[S1,...,Sm]` is a subtype of `T[T1,...,Tn]` and `c` entails `d`.

For examples of constraints and entailment, see (§4.5.1)

4.5.3 Constraint solver: incompleteness and approximation

The constraint solver is sound in that if it claims that `c` entails `d` then in fact it is the case that every valuation that satisfies `c` satisfies `d`.

[7] No experienced programmer should actually think that == is mathematical equality in any case. It is quite common for two objects to appear identical but not be ==. X10's discrepancy between the two concepts is orthogonal to the familiar one.

Limitation: X10's constraint solver is incomplete. There are situations in which `c` entails `d` but the solver cannot establish it. For instance it cannot establish that `a != b && a != c && b != c` entails `false` if `a`, `b`, and `c` are of type `Boolean`. Similarly, although `a.b==1 && a.b==2` entails `a==null`, the constraint solver does not deduce this fact.

4.5.4 Acyclicity of Properties

To ensure that typechecking is decidable, X10 requires that the graph whose nodes are types, with edges from types to the properties of those types, be *acyclic*. This is often stated as "properties are acyclic." That is, given a container type `T`, `T` cannnot have a property of type `T`, nor a property which has a property of type `T`, nor a property which has a property with a property of type `T`, etc.

Example: *The following is forbidden by the acyclicity requirement, as* `ERROR List[T]` *would have a property,* `tail`*, which is also an* `ERROR List[T]`.

```
class ERROR List[T](head:T, tail: ERROR List[T]) {}
```

Without this restriction, typechecking becomes undecidable.

4.5.5 Limitation: Generics and Constraints at Runtime

The X10 runtime does not maintain a representation of constraints as part of the runtime representation of a type. While there various approaches which could be used, they would require far higher prices in space or time than they are worth. A representation suitable for one use of types (such as keeping a closure for testing membership in the type) is unsuitable for others (such as determining if one type is a subtype of another). Furthermore, it would be necessary to compute entailment at runtime, which is currently impractical.

Rather than pay the runtime costs for keeping and manipulating constraints (which can be considerable), X10 omits them. However, this renders certain type checks uncertain: X10 needs some information at runtime, but does not have it. In particular, **casts to instances of generic types, and to type variables, are potentially troublesome.**

Example: *The following code illustrates the dangers of casting to generic types. It constructs a rail* `a` *of* `Long{self==3}`*'s – integers which are statically known to be 3. The only number that can be stored into* `a` *is* `3`*. Then it tricks the ocmpiler into thinking that it is a rail of* `Long`*, without restriction on the elements, giving it the name* `b` *at that type. The cast* `aa as Rail[Long]` *is a cast to an instance of a generic type, which is the problem.*

But, itc an store any `Long` *into the elements of* `b`*, thereby violating the invariant that all the elements of the rail are 3. This could lead to program failures, as illustrated by the failing assertion.*

With the `-VERBOSE` *compiler option, X10 prints a warning about the declaration of* `b`.

```
val a = new Rail[Long{self==3}](10, 3);
// a(0) = 1; would be illegal
a(0) = 3; // LEGAL
val aa = a as Any;
val b = aa as Rail[Long]; // WARNED with -VERBOSE
b(0) = 1;
val x : Long{self==3} = a(0);
assert x == 3 : "This fails at runtime.";
```

Since constraints are not preserved at runtime, `instanceof` and `as` cannot pay attention to them. When types are used generically, they may not behave as one would expect were one to imagine that their constraints were kept. Specifically, constraints at runtime are, in effect, simply replaced by `true`.

Example: *The following code defines generic methods* `inst` *and* `cast`*, which look like generic versions of* `instanceof` *and* `as`*. The* `example()` *code shows that* `inst` *and* `cast` *behave quite differently from* `instanceof` *and* `as`*, due to the loss of constraint information.*

The first section of `assert`*s shows the behavior of* `instanceof` *and* `at`*. We have a value* `pea`*, such that* `pea.p==1`*. It behaves as if its* `p` *field were* `1`*: it answers* `true` *to* `self.p==1`*, and* `false` *to* `self.p==2`*. This is entirely as desired.*

The following section of `assert` *and* `val` *statements does the analogous thing, but using the generic methods* `inst` *and* `cast` *rather than the built-in operations* `instanceof` *and* `cast`*.* `pea` *answers* `true` *to* `inst` *checks concerning both* `Pea{p==1}` *and* `Pea{p==2}`*, and can be* `cast()` *into both these types. This behavior is not what one would expect from runtime types that keep constraint information. It is, however, precisely what one would expect from runtime types that have their constraints replaced by* `true`*.*

The `cast2` *line shows how to use this fact to violate the constraint system at runtime. This dynamic cast produces an object of type* `Pea{p==2}` *for which* `p!=2`*.*

Note that the `-VERBOSE` *compiler flag will produce a warning that* `cast` *is unsound.*

```
class Generic {
  public static def inst[T](x:Any):Boolean = x instanceof T;
  // With -VERBOSE, the following line gets a warning
  public static def cast[T](x:Any):T       = x as T;
}
class Pea(p:Long) {}
class Example{
  static def example() {
     val pea : Pea = new Pea(1);
     // These are what you'd expect:
     assert (pea instanceof Pea{p==1});
     assert (pea as Pea{p==1}).p == 1;
     assert ! (pea instanceof Pea{p==2});
     // 'val x = pea as Pea{p==2};'
     // throws a FailedDynamicCheckException.
```

```
        // But the genericized versions don't do the same thing:
        assert Generic.inst[Pea{p==1}](pea);
        assert Generic.inst[Pea{p==2}](pea);
        // No exception here!
        val cast1: Pea{p==1} = Generic.cast[Pea{p==1}](pea);
        val cast2: Pea{p==2} = Generic.cast[Pea{p==2}](pea);
        assert cast2.p == 1;
        assert !(cast2 instanceof Pea{p==2});
    }
}
```

While in some cases it would be possible to keep constraints around at runtime and operate efficiently on them, in other cases it would not.

4.6 Function types

FunctionType ::= *TypeParams*$^?$ `(` *FormalList*$^?$ `)` *Guard*$^?$ `=>` *Type* *(20.82)*

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 to a list of values `(v1,...,vn)` provided that the constraint `c[v1,...,vn,p/x1,...,xn]` 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`.

Limitation: Constraints on closures are not supported. They parse, but are not checked.

X10 functions, like mathematical functions, take some arguments and produce a result. X10 functions, like other X10 code, can change mutable state and throw exceptions. Closures (§10) are of function type – and so are rails and arrays.

Example: *Typical functions are the reciprocal function:*

```
val recip = (x : Double) => 1/x;
```

and a function which increments element `i` *of a rail* `r`*, or throws an exception if there is no such element, where, for the sake of example, we constrain the type of* `i` *to avoid one of the many longs which are not possible subscripts:*

```
val inc = (r:Rail[Long], i: Long{i != r.size}) => {
  if (i < 0 || i >= r.size) throw new DoomExn();
  r(i)++;
};
```

In general, a function type needs to list the types $\texttt{T}_i$ of all the formal parameters, and their distinct names $\texttt{x}_i$ in case other types refer to them; a constraint `c` on the function as a whole; a return type `T`.

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

The names of the formal parameters, $\texttt{x}_i$, are bound in the type. As usual with bound variables, they can be given new names without changing the meaning of the type. In particular, the names of formals in a function type do not need to be the same as the names in the function in a value of that type.

Example: *The type of* `id` *uses the bound variable* `x`*. The type of* `ie` *uses the bound variable* `z`*, but is otherwise identical to that of* `id`*. The two types are the same, as shown by the assignment of* `id` *to* `ie`*. Also,* `id`*'s type uses* `x`*, and* `id`*'s value uses* `y`*.*

```
val id : (x:Long) => Long{self==x}
       = (y:Long) => y;
val ie : (z:Long) => Long{self==z}
       = id;
```

Limitation: Function types differing only in the names of bound variables may wind up being considered different in X10 v2.2, especially if the variables appear in constraints.

The formal parameter names are in scope from the point of definition to the end of the function type—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 type of the reciprocal function can be written as `(Double)=>Double`.

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 class or struct definition may use a function type

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

in its implements clause; this is equivalent to implementing an interface requiring the single operator

```
public operator this(x1:T1,...,xn:Tn){c}:T
```

Similarly, an interface definition may specify a function type `F` in its `extends` clause. Values of a class or struct implementing `F` can be used as functions of type `F` in all ways. In particular, applying one to suitable arguments calls the `apply` method.

Limitation: A class or struct may not implement two different instantiations of a generic interface. In particular, a class or struct can implement only one function type.

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` may be implemented, but not extended, by a class or function type. Nor is it a struct type, for it has no predefined notion of equality.

4.7 Default Values

Some types have default values, and some do not. Default values are used in situations where variables can legitimately be used without having been initialized; types without default values cannot be used in such situations. For example, a field of an object `var x:T` can be left uninitialized if `T` has a default value; it cannot be if `T` does not. Similarly, a transient (§8.2.3) field `transient val x:T` is only allowed if `T` has a default value.

Default values, or lack of them, is defined thus:

- The fundamental numeric types (`Int`, `UInt`, `Long`, `ULong`, `Short`, `UShort`, `Byte`, `UByte`, `Float`, `Double`) all have default value `0`.
- `Boolean` has default value `false`.
- `Char` has default value `'\0'`.
- If every field of a struct type `T` has a default value, then `T` has a default value. If any field of `T` has no default value, then `T` does not. (§9.7)
- A function type has a default value of `null`.
- A class type has a default value of `null`.
- The constrained type `T{c}` has the same default value as `T` if that default value satisfies `c`. If the default value of `T` doesn't satisfy `c`, then `T{c}` has no default value.

Example: `var x: Long{x != 4}` *has default value 0, which is allowed because* `0 != 4` *satisfies the constraint on* `x`. `var y : Long{y==4}` *has no default value, because* `0` *does not satisfy* `y==4`. *The fact that* `Long{y==4}` *has precisely one value,* viz. *4, doesn't matter; the only candidate for its default value, as for any subtype of* `Long`*, is 0.* `y` *must be initialized before it is used.*

The predicate `T haszero` tells if the type `T` has a default value. `haszero` may be used in constraints.

Example: *The following code defines a sort of cell holding a single value of type* `T`. *The cell is initially empty – that is, has* `T`*'s zero value – but may be filled later.*

```
class Cell0[T]{T haszero} {
  public var contents : T;
  public def put(t:T) { contents = t; }
}
```

The built-in type `Zero` has the method `get[T]()` which returns the default value of type `T`.

Example: *As a variation on a theme of* `Cell0`*, we define a class* `Cell1[T]` *which can be initialized with a value of an arbitrary type* `T`*, or, if* `T` *has a default value, can be created with the default value. Note that* `T haszero` *is a constraint on one of the constructors, not the whole type:*

```
class Cell1[T] {
  public var contents: T;
  def this(t:T) { contents = t; }
  def this(){T haszero} { contents = Zero.get[T](); }
  public def put(t:T) {contents = t;}
}
```

4.8 Annotated types

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

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

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

4.9 Subtyping and type equivalence

Intuitively, type $\texttt{T}_1$ is a subtype of type $\texttt{T}_2$, written $\texttt{T}_1 \ \texttt{<:} \ \texttt{T}_2$, if every instance of $\texttt{T}_1$ is also an instance of $\texttt{T}_2$. For example, `Child` is a subtype of `Person` (assuming a suitably defined class hierarchy): every child is a person. Similarly, `Long{self != 0}` is a subtype of `Long` – every non-zero integer is an integer.

This section formalizes the concept of subtyping. Subtyping of types depends on a *type context*, *viz.*. a set of constraints on type parameters and variables that occur in the type. For example:

```
class ConsTy[T,U] {
   def upcast(t:T){T <: U} :U = t;
}
```

Inside `upcast`, `T` is constrained to be a subtype of `U`, and so `T <: U` is true, and `t` can be treated as a value of type `U`. Outside of `upcast`, there is no reason to expect any relationship between them, and `T <: U` may be false. However, subtyping of types that have no free variables does not depend on the context. `Long{self != 0} <: Long` is always true.

Limitation: Subtyping of type variables does not work under all circumstances in the X10 2.2 implementation.

- **Reflexivity:** Every type `T` is a subtype of itself: `T <: T`.
- **Transitivity:** If `T <: U` and `U <: V`, then `T <: V`.

- **Direct Subclassing:** Let $\vec{X}$ be a (possibly empty) vector of type variables, and $\vec{Y}$, $\vec{Y_i}$ be vectors of type terms over $\vec{X}$. Let $\vec{T}$ be an instantiation of $\vec{X}$, and $\vec{U}$, $\vec{U_i}$ the corresponding instantiation of $\vec{Y}$, $\vec{Y_i}$. Let `c` be a constraint, and $\texttt{c}'$ be the corresponding instantiation. We elide properties, and interpret empty vectors as absence of the relevant clauses. Suppose that `C` is declared by one of the forms:

  1. `class C[`$\vec{X}$`]{c} extends D[`$\vec{Y}$`]{d}`
     `implements I`$_1$`[`$\vec{Y_1}$`]{i`$_1$`},...,I`$_n$`[`$\vec{Y_n}$`]{i`$_n$`}{`
  2. `interface C[`$\vec{X}$`]{c} extends I`$_1$`[`$\vec{Y_1}$`]{i`$_1$`},...,I`$_n$`[`$\vec{Y_n}$`]{i`$_n$`}{`
  3. `struct C[`$\vec{X}$`]{c} implements I`$_1$`[`$\vec{Y_1}$`]{i`$_1$`},...,I`$_n$`[`$\vec{Y_n}$`]{i`$_n$`}{`

  Then:

  1. `C[`$\vec{T}$`] <: D[`$\vec{U}$`]{d}` for a class
  2. `C[`$\vec{T}$`] <: I`$_i$`[`$\vec{U_i}$`]{i`$_i$`}` for all cases.
  3. `C[`$\vec{T}$`] <: C[`$\vec{T}$`]{c'}` for all cases.

- **Function types:**

  `(x`$_1$`: T`$_1$`, ..., x`$_n$`: T`$_n$`){c} => T`

  is a subtype of

  `(x'`$_1$`: T'`$_1$`, ..., x'`$_n$`: T'`$_n$`){c'} => T'`

  if:

  1. Each `T`$_i$ `<: T'`$_i$;
  2. `c[x'`$_1$`, ..., x'`$_n$ `/ x`$_1$`, ..., x`$_n$`]` entails $\texttt{c}'$;
  3. $\texttt{T}'$ `<: T`;

- **Constrained types:** `T{c}` is a subtype of `T{d}` if `c` entails `d`.

- **Any:** Every type `T` is a subtype of `x10.lang.Any`.

- **Type Variables:** Inside the scope of a constraint `c` which entails `A <: B`, we have `A <: B`. *e.g.*, `upcast` above.

Two types are *equivalent*, `T == U`, if `T <: U` and `U <: T`.

4.10 Common ancestors of types

There are several situations where X10 must find a type `T` that describes values of two or more different types. This arises when X10 is trying to find a good type for:

- Conditional expressions, like `test ? 0 : "non-zero"` or even `test ? 0 : 1`;

- Rail construction, like `[0, "non-zero"]` and `[0,1]`;
- Functions with multiple returns, like

```
def f(a:Long) {
  if (a == 0) return 0;
  else return "non-zero";
}
```

In some cases, there is a unique best type describing the expression. For example, if `B` and `C` are direct subclasses of `A`, `pick` will have return type `A`:

```
static def pick(t:Boolean, b:B, c:C) = t ? b : c;
```

However, in many common cases, there is no unique best type describing the expression. For example, consider the expression E

```
b ? 0 : 1   // Call this expression E
```

The best type of `0` is `Long{self==0}`, and the best type of 1 is `Long{self==1}`. Certainly E could be given the type `Long`, or even `Any`, and that would describe all possible results. However, we actually know more. `Long{self != 2}` is a better description of the type of E—certainly the result of E can never be `2`. `Long{self != 2, self != 3}` is an even better description; E can't be `3` either. We can continue this process forever, adding integers which E will definitely not return and getting better and better approximations. (If the constraint sublanguage had `||`, we could give it the type `Long{self == 0 || self == 1}` , which would be nearly perfect. But `||` makes typechecking far more expensive, so it is excluded.) No X10 type is the best description of E; there is always a better one.

Similarly, consider two unrelated interfaces:

```
interface I1 {}
interface I2 {}
class A implements I1, I2 {}
class B implements I1, I2 {}
class C {
  static def example(t:Boolean, a:A, b:B) = t ? a : b;
}
```

`I1` and `I2` are both perfectly good descriptions of `t ? a : b`, but neither one is better than the other, and there is no single X10 type which is better than both. (Some languages have *conjunctive types*, and could say that the return type of `example` was `I1 && I2`. This, too, complicates typechecking.)

So, when confronted with expressions like this, X10 computes *some* satisfactory type for the expression, but not necessarily the *best* type. X10 provides certain guarantees about the common type `V{v}` computed for `T{t}` and `U{u}`:

- If `T{t} == U{u}`, then `V{v} == T{t} == U{u}`. So, if X10's algorithm produces an utterly untenable type for `a ? b : c`, and you want the result to have type `T{t}`, you can (in the worst case) rewrite it to

```
a ? b as T{t} : c as T{t}
```

- If `T == U`, then `V == T == U`. For example, X10 will compute the type of `b ? 0 : 1` as `Long{c}` for some constraint `c`—perhaps simply picking `Long{true}`, *viz.*, `Long`.
- X10 preserves place information about `GlobalRef`s, because it is so important. If both `t` and `u` entail `self.home==p`, then `v` will also entail `self.home==p`.
- X10 similarly preserves nullity information. If `t` and `u` both entail `x == null` or `x != null` for some variable `x`, then `v` will also entail it as well.
- The computed upper bound of function types with the *same* argument types is found by computing the upper bound of the result types. If `T = (`T_1`, ..., `T_n`) => T'` and `U = (`T_1`, ..., `T_n`) => U'`, and `V'` is the computed upper bound of `T'` and `U'`, then the computed upper bound of `T` and `U` is `U = (`T_1`, ..., `T_n`) => V'`. (But, if the argument types are different, the computed upper bound may be `Any`.)

4.11 Fundamental types

Certain types are used in fundamental ways by X10.

4.11.1 The interface `Any`

It is quite convenient to have a type which all values are instances of; that is, a supertype of all types.[8] X10's universal supertype is the interface `Any`.

```
package x10.lang;
public interface Any {
  def toString():String;
  def typeName():String;
  def equals(Any):Boolean;
  def hashCode():Long;
}
```

`Any` provides a handful of essential methods that make sense and are useful for everything. `a.toString()` produces a string representation of `a`, and `a.typeName()` the string representation of its type; both are useful for debugging. `a.equals(b)` is the programmer-overridable equality test, and `a.hashCode()` an integer useful for hashing.

[8] Java, for one, suffers a number of inconveniences because some built-in types like `long` and `char` aren't subtypes of anything else.

4.12 Type inference

X10 v2.4 supports limited local type inference, permitting certain variable types and return types to be elided. It is a static error if an omitted type cannot be inferred or uniquely determined. Type inference does not consider coercions.

4.12.1 Variable declarations

The type of a `val` variable declaration can be omitted if the declaration has an initializer. The inferred type of the variable is the computed type of the initializer. For example, `val seven = 7;` is identical to

```
val seven: Long{self==7} = 7;
```

Note that type inference gives the most precise X10 type, which might be more specific than the type that a programmer would write.

Limitation: At the moment, `var` declarations may not have their types elided in this way.

4.12.2 Return types

The return type of a method can be omitted if the method has a body (*i.e.*, is not `abstract` or `native`). The inferred return type is the computed type of the body. In the following example, the return type inferred for `isTriangle` is `Boolean{self==false}`

```
class Shape {
  def isTriangle() = false;
}
```

Note that, as with other type inference, methods are given the most specific type. In many cases, this interferes with subtyping. For example, if one tried to write:

```
class Triangle extends Shape {
  def isTriangle() = true;
}
```

the compiler would reject this program for attempting to override `isTriangle()` by a method with the wrong type, *viz.*, `Boolean{self==true}`. In this case, supply the type that is actually intended for `isTriangle`:

```
def isTriangle() : Boolean =false;
```

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. 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. For example, the `Spot` class has two constructors, the first of which has inferred return type `Spot{x==0}` and the second of which has inferred return type `Spot{x==xx}`.

```
class Spot(x:Long) {
  def this() {property(0);}
  def this(xx: Long) { property(xx); }
}
```

A method or closure that has expression-free `return` statements (`return;` rather than `return e;`) is said to be a `void` method or closure. `void` is not a type; there are no `void` values, nor can `void` be used as the argument of a generic type. However, `void` takes the syntactic place of a type in a few contexts. A `void` method can be specified by `def m():void`, and similarly for a closure:

```
def m():void {return;}
val f : () => void = () => {return;};
```

By a convenient abuse of language, `void` is sometimes lumped in with types; *e.g.*, we may say "return type of a method" rather than the formally correct but rather more awkward "return type of a method that is not a `void` method". Despite this informal usage, `void` is not a type and cannot be used as the value of a type parameter. For example, given

```
static def eval[T] (f:()=>T):T = f();
```

The call `eval[void](f)` does *not* typecheck. There is no way in X10 to write a generic function which works with both functions which return a value and functions which do not. In such cases it may be convenient to define a type `Unit` thus:

```
struct Unit{}
```

Functions whose return type is `Unit` may simply return the expression `Unit()` which evaluates to the unique value of type `Unit`. (By definition of equality of structurs `Unit()==Unit()`.)

X10 preserves known information when computing return types. A constraint on a method induces a corresponding constraint on its return type.

Example: *In the following code, the type inferred for* `x` *is* `Numb{self.p==n, n!=0, self!=null}`. *In particular, the conjunct* `n != 0` *is preserved from the cast of* `n` *to* `Long{self != 0}`.

```
class Numb(p:Long){
  static def dup(n:Long){n != 0} = new Numb(n);
  public static def example(n:Long) {
    val x = dup(n as Long{self != 0});
    val y : Numb{self.p==n, n!=0, self!=null} = x;
  }
}
```

4.12.3 Inferring Type Arguments

A call to a polymorphic method may omit the explicit type arguments. X10 will compute a type from the types of the actual arguments. Failure of the compiler to infer unique types for omitted type arguments is a compile-time error. For instance, given the method definition `def m[T](){...}`, an invocation `m()` is considered a compile-time error. The compiler has no idea what `T` the programmer intends.

Example: *Consider the following method, which chooses one of its arguments. (A more sophisticated one might sometimes choose the second argument, but that does not matter for the sake of this example.)*

```
static def choose[T](a: T, b: T): T = a;
```

The type argument `T` *can always be supplied:* `choose[Long](1, 2)` *picks an integer, and* `choose[Any](1, "yes")` *picks a value that might be an integer or a string. However, the type argument can be elided. Suppose that* `Sub <: Super`*; then the following compiles:*

```
static def choose[T](a: T, b: T): T = a;
static val j : Any = choose("string", 1);
static val k : Super = choose(new Sub(), new Super());
```

The type parameter doesn't need to be the type of a variable. It can be found inside of the type of a variable; X10 can extract it.

Example: *The* `first` *method below returns the first element of a rail. The type parameter* `T` *represents the type of the rail's elements. There is no parameter of type* `T`*. There is one of type* `Rail[String]{length==3}`*. When doing type inference, the compiler is able to infer that* `T` *should be instantiated to* `String`*:*

```
static def first[T](x:Rail[T]) = x(0);
static def example() {
  val ss <: Rail[String] = ["X10", "Java", "C++"]; // ok
  val s1 <: String = first(ss); // ok
  assert s1.equals("X10");
}
```

Sketch of X10 Type Inference for Method Calls

When the X10 compiler sees a method call

$a.m(b_1, \ldots, b_n)$

and attempts to infer type parameters to see if it could be a use of a method

$def\ m[X_1, \ldots, X_t](y_1 : S_1, \ldots, y_n : S_n),$

it reasons as follows.

Let

$\texttt{T}_i$ be the type of $\texttt{b}_i$

Then, the compiler is seeking a set B of type bindings

$$B = \{\ \texttt{X}_1\ \texttt{=}\ \texttt{U}_1, \ldots,\ \texttt{X}_t\texttt{=U}_t\}$$

such that $\texttt{T}_i\ \texttt{<:}\ \texttt{S}_i^*$ for $1 \leq i \leq n$, where $\texttt{S}^*$ is S with each type variable $\texttt{X}_j$ replaced by the corresponding $\texttt{U}_j$. If it can find such a B, it has a usable choice of type arguments and can do the type inference. If it cannot find B, then it cannot do type inference. (Note that X10's type inference algorithm is incomplete – there may *be* such a B that X10 cannot find. If this occurs in your program, you will have to write down the type arguments explicitly.)

Let B_0 be the set $\{T_i$ <: $S_i | 1 \leq i \leq n\}$. Let B_{n+1} be B_n with one element F <: G or $F = G$ removed, and *Strip*(F <: G) or *Strip*($F = G$), where *Strip* is defined below, added. Repeat this until B_n consists entirely of comparisons with type variables (*viz.*, $\texttt{Y}_j\ \texttt{=}\ \texttt{U}$, $\texttt{Y}_j\ \texttt{<:}\ \texttt{U}$, and $\texttt{Y}_j\ \texttt{:>}\ \texttt{U}$), or until some n exceeds a predefined compiler limit.

The candidate inferred types may be read off of B_n. The guessed binding for $\texttt{X}_j$ is:

- If there is an equality $\texttt{X}_j\texttt{=W}$ in B_n, then guess the binding $\texttt{X}_j\texttt{=W}$. Note that there may be several such equalities with different choices of W; pick any one. If the chosen binding does not equal the others, the candidate binding will be rejected later and type inference will fail.
- Otherwise, if there is one or more upper bounds $\texttt{X}_j\ \texttt{<:}\ \texttt{V}_k$ in B_n, guess the binding $\texttt{X}_j\ \texttt{=}\ \texttt{V}_+$, where $\texttt{V}_+$ is the computed lower bound of all the $\texttt{V}_k$'s.
- Otherwise, if there is one or more lower bounds $\texttt{R}_k\ \texttt{<:}\ \texttt{X}_j$, guess that $\texttt{X}_j\ \texttt{=}\ \texttt{R}_+$, where $\texttt{R}_+$ is the computed upper bound of all the $\texttt{R}_k$'s.

If this does not yield a binding for some variable $\texttt{X}_j$, then type inference fails. Furthermore, if every variable $\texttt{X}_j$ is given a binding $\texttt{U}_j$, but the bindings do not work — that is, if $\texttt{a.m[U}_1\texttt{, ..., U}_t\texttt{](b}_1\texttt{, ...,b}_n\texttt{)}$ is not a well-typed call of the original method $\texttt{def m[X}_1\texttt{, ..., X}_t\texttt{](y}_1\texttt{: S}_1\texttt{, ..., y}_n\texttt{:S}_n\texttt{)}$ — then type inference also fails.

Computation of the Replacement Elements Given a type relation r of the form F <: G or $F = G$, we compute the set *Strip*(r) of replacement constraints. There are a number of cases; we present only the interesting ones.

- If F has the form $F'\texttt{\{c\}}$, then *Strip*(r) is defined to be $F'\ \texttt{=}\ G$ if r is an equality, or $F'\ \texttt{<:}\ G$ if r is a subtyping. That is, we erase type constraints. Validity is not an issue at this point in the algorithm, as we check at the end that the result is valid. Note that, if the equation had the form $\texttt{Z\{c\}}\ \texttt{=}\ \texttt{A}$, it could be solved by either Z=A or by $\texttt{Z}\ \texttt{=}\ \texttt{A\{c\}}$. By dropping constraints in this rule, we choose the former solution, which tends to give more general types in results.
- Similarly, we drop constraints on G as well.

- If F has the form `K[F`$_1$`, ..., F`$_k$`]` and G has the form `K[G`$_1$`, ..., G`$_k$`]`, then *Strip*(r) has one type relation comparing each parameter of F with the corresponding one of G:

$$\textit{Strip}(r) = \{F_l = G_l | 1 \leq l \leq k\}$$

  For example, the constraint `List[X] = List[Y]` induces the constraint `X=Y`. `List[X] <: List[Y]` also induces the same constraint. The only way that `List[X]` could be a subtype of `List[Y]` in X10 is if `X=Y`. List of different types are incomparable.[9]

- Other cases are fairly routine. *E.g.*, if F is a `type`-defined abbreviation, it is expanded.

Example: *Consider the program:*

```
import x10.util.*;
class Cl[C1, C2, C3]{}
class Example {
  static def me[X1, X2](Cl[Long, X1, X2]) =
     new Cl[X1, X2, Point]();
  static def example() {
    val a = new Cl[Long, Boolean, String]();
    val b : Cl[Boolean, String, Point]
          = me[Boolean, String](a);
    val c : Cl[Boolean, String, Point]
          = me(a);
  }
}
```

The method call for `b` *has explicit type parameters. The call for* `c` *infers the parameters. The computation starts with one equation, saying that the formal parameter of* `me` *has to be able to accept the actual parameter* `a`*:*

```
Cl[Long, Boolean, String] <: Cl[Long, X1, X2]
```

Note that both terms are `Cl` *of three things. This is broken into three equations:*

```
Long = Long
```

which is easy to satisfy,

```
X1 = Boolean
```

which suggests a possible value for `X1`*, and*

```
X2 = String
```

[9] The situation would be more complex if X10 had covariant and contravariant types.

which suggests a value for `X2`*. All of these equations are simple enough, so the algorithm terminates.*

Then, X10 confirms that the binding `X1=Boolean`*,* `X2=String` *actually generates a correct call, which it does.*

Example: *When there is no way to infer types correctly, the type inference algorithm will fail. Consider the program:*

```
public class Failsome {
  static def fail[X](a:Rail[X], b:Rail[X]):void {}
  public static def main(argv:Rail[String]) {
    val aint  : Rail[Long]    = [1,2,3];
    val abool : Rail[Boolean] = [true, false];
    fail(aint, abool);    // THIS IS WRONG
  }
}
```

The type inference computation starts, as always, by insisting that the types of the formals to `fail` *are capable of accepting the actuals:*

$$B_0 = \{\texttt{Rail[Long] <: Rail[X]}, \texttt{ Rail[Boolean] <: Rail[X]}\}$$

Arbitrarily picking the first relation to Strip first, we get:

$$B_1 = \{\texttt{Long = X}, \texttt{ Rail[Boolean] <: Rail[X] }\}$$

and then

$$B_2 = \{\texttt{Long = X}, \texttt{ Boolean = X }\}$$

(At this point it is clear to a human that B is inconsistent, but the algorithm's check comes a bit later.) B_2 consists entirely of comparisons with type variables, so the loop is over. Arbitrarily picking the first equality, it guesses the binding

$$B = \{\texttt{X = Long}\}.$$

In the validation step, it checks that

```
fail[Long](aint, abool)
```

is a well-typed call to `fail`*. Of course it is not;* `abool` *would have to be a value of type* `Rail[Long]`*, which it is not. So type inference fails at this point. In this case it is correct: there is no way to give a proper type to this program.*[10]

[10] In particular, `X=Any` doesn't work either. A `Rail[Long]` is not a `Rail[Any]` — and it must not be, for you can put a boolean value into a `Rail[Any]`, but you cannot put a boolean value into an `Rail[Long]`. However, if the types of the arguments had simply been `X` rather than `Rail[X]`, then type inference would correctly infer `X=Any`.

4.13 Type Dependencies

Type definitions may not be circular, in the sense that no type may be its own supertype, nor may it be a container for a supertype. This forbids interfaces like `interface Loop extends Loop`, and indirect self-references such as `interface A extends B.C` where `interface B extends A`. The formal definition of this is based on Java's.

An *entity type* is a class, interface, or struct type.

Entity type E *directly depends on* entity type F if F is mentioned in the `extends` or `implements` clause of E, either by itself or as a qualifier within a super-entity-type name.

Example: *In the following,* `A` *directly depends on* `B`*,* `C`*,* `D`*,* `E`*, and* `F`*. It does not directly depend on* `G`*.*

```
class A extends B.C implements D.E, F[G] {}
```

It is an ordinary programming idiom to use `A` *as an argument to a generic interface that* `A` *implements. For example,* `ComparableTo[T]` *describes things which can be compared to a value of type* `T`*. Saying that* `A` *implements* `ComparableTo[A]` *means that one* `A` *can be compared to another, which is reasonable and useful:*

```
interface ComparableTo[T] {
  def eq(T):Boolean;
}
class A implements ComparableTo[A] {
  public def eq(other:A) = this.equals(other);
}
```

Entity type E *depends on* entity type F if either E directly depends on F, or E directly depends on an entity type that depends on F. That is, the relation "depends on" is the transitive closure of the relation "directly depends on".

It is a static error if any entity type E depends on itself.

4.14 Typing of Variables and Expressions

Variable declarations, field declarations, and some other expressions introduce constraints on their types. These extra constraints represent information that is known at the point of declaration. They are used in deductions and type inference later on – as indeed all constraints are, but the automatically-added constraints are added because they are particularly useful.

Any variable declaration of the form

```
val x : A ...
```

results in declaring `x` to have the type `A{self==x}`, rather than simply `A`. (`var` declarations get no such addition, because `var`s cannot appear in constraints.)

A field or property declaration of the form:

```
class A {
   ...
   val f : B ...
   ...
}
```

results in declaring `f` to be of type `B{self==this.f}`. And, if `y` has type `A{c}`, then the type for `y.f` has a constraint `self==y.f`, and, additionally, preserves the information from `c`.

Example:

The following code uses a method `typeIs[T](x)` *to confirm, statically, that the type of* `x` *is* `T` *(or a subtype of* `T`*).*

On line `(A)` *we confirm that the type of* `x` *has a* `self==x` *constraint. The error line* `(!A)` *confirms that a* different *variable* doesn't *have the* `self==x` *constraint.* `(B)` *shows the extra information carried by a field's type.*

`(C)` *shows the extra information carried by a field's type when the object's type is constrained. Note that the constraint* `ExtraConstraint{self.n==8}` *on the type of* `y` *has to be rewritten for* `y.f`*, since the constraint* `Long{self.n==8}` *is not correct or even well-typed. In this case, the* `ExtraConstraint` *whose* `n`*-field is* `8` *has the name* `y`*, so we can write the desired type with a conjunct* `y.n==8`.[11]

Note that we use one of the extra constraints here – this reasoning requires the information that the type of `y` *has the constraint* `self==y`*, so X10 can infer* `y.n==8` *from* `self.n==8`*. This sort of inference is the reason why X10 adds these constraints in the first place: without them, even the simplest data flows would be beyond the ability of the type system to detect.*

```
class Extra(n:Long) {
  val f : Long;
  def this(n:Long, f:Long) { property(n); this.f = f; }
  static def typeIs[T](val x:T) {}
  public static def main(argv:Rail[String]) {
     val x : Extra = new Extra(1,2L);
     typeIs[ Extra{self==x} ]   (x);    //(A)
     val nx: Extra = new Extra(1,2L);
     // ERROR: typeIs[ Extra{self==x} ]   (nx); //(!A)
     typeIs[ Long{self == x.f} ]          (x.f);  //(B)
     val y : Extra{self.n==8} = new Extra(8, 4L);
```

[11] If `y` were an expression rather than a variable, there would be no good way to express its type in X10's type system. (The compiler has a more elaborate internal representation of types, not all of which are expressible in X10 version 2.2.)

```
      typeIs[ Long{self == y.f, y.n == 8}] (y.f);  //(C)
    }
  }
```

Once in a while, the additional information will interfere with other typechecking or type inference. In this case, use `as` (§11.23) to erase it, using expressions like `x as A`.

Example: *The following code creates a one-element rail (§11.26) containing* `x`*.*

If the `ERROR` *line were to be used, X10 would infer that the type of this rail were* `Rail[T]`*, where* `T` *is the type of* `x` *— that is,* `Rail[Extra{self==x}]`*.* `[x]` *is a rail of* `x`*'s, not a rail of* `Extra`*s. Since* `Rail[Extra{self==x}]` *is not a subtype of* `Rail[Extra]`*, the rail* `[x]` *cannot be used in a place where an* `Rail[Extra]` *is called for.*

The expression `[x as Extra]` *uses a type cast to erase the automatically-added extra information about* `x`*.* `x as Extra` *simply has type* `Extra`*, and thus* `[x as Extra]` *is a* `Rail[Extra]` *as desired.*

```
class Extra {
  static def useRail(Rail[Extra]) {}
  public static def main(argv:Rail[String]) {
     val x : Extra = new Extra();
     //ERROR: useRail([x]);
     useRail([x as Extra]);
  }
}
```

4.15 Limitations of Strict Typing

X10's type checking provides substantial guarantees. In most cases, a program that passes the X10 type checker will not have any runtime type errors. However, there are a modest number of compromises with practicality in the type system: places where a program can pass the typechecker and still have a type error.

1. As seen in §4.5.5, generic types do not have constraint information at runtime. This allows one to write code which violates constraints at runtime, as seen in the example in that section.
2. The library type `x10.util.IndexedMemoryChunk` provides a low-level interface to blocks of memory. A few methods on that class are not type-safe. See the API if you must.
3. Custom serialization (§13.3.2) allows user code to construct new objects in ways that can subvert the type system.
4. Code written to use the underlying Java or C++ (§18) can break X10's guarantees.

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.

- 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 bindings enjoy similar concepts of scope and visibility.

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

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

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

Both variables have type `Long` *(or perhaps something more specific). The scope of* `n` *is the body of* `Counter`*; the scope of* `nxt` *is the body of* `bump`*. The lifetime of* `n` *is the lifetime of the* `Counter` *object holding it; the lifetime of* `nxt` *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`.

A variable—even a `val` – can be declared in one statement, and then initialized later on. It must be initialized before it can be used (§19).

Example: *The following example illustrates many of the variations on variable declaration:*

```
val a : Long = 0;               // Full 'val' syntax
b : Long = 0;                   // 'val' implied
val c = 0;                     // Type inferred
var d : Long = 0;               // Full 'var' syntax
var e : Long;                   // Not initialized
var f : Long{self != 100} = 0;  // Constrained type
val g : Long;                   // Init. deferred
if (a > b) g = 1; else g = 2;  // Init. done here.
```

5.1 Immutable variables

LocVarDeclnStmt	::=	*LocVarDecln* ;	*(20.111)*
LocVarDecln	::=	*Mods*$^?$ *VarKeyword VariableDeclrs*	*(20.110)*
	|	*Mods*$^?$ *VarDeclsWType*	
	|	*Mods*$^?$ *VarKeyword FormalDeclrs*	

An immutable (`val`) variable can be given a value (by initialization or assignment) at most once, and must be given a value before it is used. Usually this is achieved by declaring and initializing the variable in a single statement, such as `val x = 3`, with syntax (20.110) using the *VariableDeclarators* or *VarDelcsWType* alternatives.

Example: *After these declarations,* `a` *and* `b` *cannot be assigned to further, or even redeclared:*

```
val a : Long = 10;
val b = (a+1)*(a-1);
// ERROR: a = 11;  // vals cannot be assigned to.
// ERROR: val a = 11; // no redeclaration.
```

In three special cases, the declaration and assignment are separate. One case is how constructors give values to `val` fields of objects. In this case, production (20.110) is taken, with the *FormalDeclarators* option, such as `var n:Long;`.

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

The second case of separating declaration and assignment is in function and method call, described in §5.4. The formal parameters are bound to the corresponding actual parameters, but the binding does not happen until the function is called.

Example: *In the code below,* `x` *is initialized to* `3` *in the first call and* `4` *in the second.*

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

The third case is delayed initialization (§19), useful in cases where the code has to make decisions (possibly asynchronously) before assigning values to variables.

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 – giving a variable a value as part of its declaration.

Example: *These variables are all initialized:*

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

A variable may also be given a value by an assignment. `var` variables may be assigned to repeatedly. `val` variables may only be assigned once; the compiler will ensure that they are assigned before they are used (§19).

Example: *The variables in the following example are given their initial values by assignment. Note that they could not be used before those assignments, nor could* `immu` *be assigned repeatedly.*

```
var muta : Long;
// ERROR:  println(muta);
muta = 4;
val use2A = muta * 10;
val immu : Long;
// ERROR: println(immu);
if (cointoss())   {immu = 1;}
else              {immu = use2A;}
val use2B = immu * 10;
// ERROR: immu = 5;
```

Every class variable must be initialized before it is read, through the execution of an explicit initializer. Every instance variable must be initialized before it is read, through the execution of an explicit or implicit initializer or a constructor. Implicit initializers initialize `var`s to the default values of their types (§4.7). Variables of types which do not have default values are not implicitly initialized.

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 (§19).

5.3 Destructuring syntax

X10 permits a *destructuring* syntax for local variable declarations with explicit initializers, and for formal parameters, of type `Point`, §16.3.1 and `Array`, §16. A point is a sequence of zero or more `Long`-valued coordinates; an array is an indexed collection of data. It is often useful to get at the coordinates or elements directly, in variables.

VariableDeclr ::= *Id HasResultType*$^?$ = *VariableInitializer* *(20.203)*
| [*IdList*] *HasResultType*$^?$ = *VariableInitializer*
| *Id* [*IdList*] *HasResultType*$^?$ = *VariableInitializer*

The syntax `val [`a_1`, ..., `a_n`] = e;`, where `e` is a `Point`, declares n `Long` variables, bound to the precisely n components of the `Point` value of `e`; it is an error if `e` is not a `Point` with precisely n components. The syntax `val p[`a_1`, ..., `a_n`] = e;` is similar, but also declares the variable `p` to be of type `Point(n)`.

The syntax `val [a1, ..., an] = e;`, where `e` is an `Array[T]` for some type `T`, declares n variables of type `T`, bound to the precisely n components of the `Array[T]` value of `e`; it is an error if `e` is not a `Array[T]` with `rank==1` and `size==`n. The syntax `val p[a1, ..., an] = e;` is similar, but also declares the variable `p` to be of type `Array[T]{rank==1,size==n}`.

Example: *The following code makes an anonymous point with one coordinate* `11`*, and binds* `i` *to* `11`*. Then it makes a point with coordinates* `22` *and* `33`*, binds* `p` *to that point, and* `j` *and* `k` *to* `22` *and* `33` *respectively.*

```
val [i] : Point = Point.make(11);
assert i == 11L;
val p[j,k] = Point.make(22,33);
assert j == 22L && k == 33L;
val q[l,m] = [44,55] as Point;
assert l == 44L && m == 55L;
//ERROR: val [n] = p;
```

Destructuring is allowed wherever a `Point` *or* `Array[T]` *variable is declared,* e.g.*, as the formal parameters of a method.* **Example:** *The methods below take a single argument each: a three-element point for* `example1`*, a three-element array for* `example2`*. The argument itself is bound to* `x` *in both cases, and its elements are bound to* `a`*,* `b`*, and* `c`*.*

```
static def example1(x[a,b,c]:Point){}
static def example2(x[a,b,c]:Array[String]{rank==1,size==3L}){}
```

5.4 Formal parameters

Formal parameters are the variables which hold values transmitted into a method or function. They are always declared with a type. (Type inference is not available, because there is no single expression to deduce a type from.) The variable name can be omitted if it is not to be used in the scope of the declaration, as in the type of the method `static def main(Rail[String]):void` executed at the start of a program that does not use its command-line arguments.

`var` and `val` behave just as they do for local variables, §5.5. In particular, the following `inc` method is allowed, but, unlike some languages, does *not* increment its actual parameter. `inc(j)` creates a new local variable `i` for the method call, initializes `i` with the value of `j`, increments `i`, and then returns. `j` is never changed.

```
static def inc(var i:Long) { i += 1; }
static def example() {
   var j : Long = 0;
   assert j == 0;
   inc(j);
   assert j == 0;
}
```

5.5 Local variables and Type Inference

Local variables are declared in a limited scope, and, dynamically, keep their values only for so long as the scope is being executed. They may be `var` or `val`. They may have initializer expressions: `var i:Long = 1;` introduces a variable `i` and initializes it to 1. If the variable is immutable (`val`) the type may be omitted and inferred from the initializer type (§4.12).

The variable declaration `val x:T=e;` confirms that `e`'s value is of type `T`, and then introduces the variable `x` with type `T`. For example, consider a class `Tub` with a property `p`.

```
class Tub(p:Long){
  def this(pp:Long):Tub{self.p==pp} {property(pp);}
  def example() {
    val t : Tub = new Tub(3);
  }
}
```

produces a variable `t` of type `Tub`, even though the expression `new Tub(3)` produces a value of type `Tub{self.p==3}` – that is, a `Tub` whose `p` field is 3. This can be inconvenient when the constraint information is required.

Including type information in variable declarations is generally good programming practice: it explains to both the compiler and human readers something of the intent of the variable. However, including types in `val t:T=e` can obliterate helpful information. So, X10 allows a *documentation type declaration*, written

```
val t <: T = e
```

This has the same effect as `val t = e`, giving `t` the full type inferred from `e`; but it also confirms statically that that type is at least `T`.

Example: *The following gives* `t` *the type* `Tub{self.p==3}` *as desired. However, a similar declaration with an inappropriate type will fail to compile.*

```
val t <: Tub = new Tub(3);
// ERROR: val u <: Long = new Tub(3);
```

5.6 Fields

FieldDeclrs	::=	*FieldDeclr*	*(20.70)*
	|	*FieldDeclrs* `,` *FieldDeclr*	
FieldDecln	::=	*Mods*$^?$ *VarKeyword FieldDeclrs* `;`	*(20.68)*
	|	*Mods*$^?$ *FieldDeclrs* `;`	
FieldDeclr	::=	*Id HasResultType*	*(20.69)*
	|	*Id HasResultType*$^?$ `=` *VariableInitializer*	
HasResultType	::=	*ResultType*	*(20.86)*
	|	`<:` *Type*	
Mod	::=	`abstract`	*(20.121)*
	|	*Annotation*	
	|	`atomic`	
	|	`final`	
	|	`native`	
	|	`private`	
	|	`protected`	
	|	`public`	
	|	`static`	
	|	`transient`	
	|	`clocked`	

Like most other kinds of variables in X10, the fields of an object can be either `val` or `var`. `val` fields can be `static` (§8.2). Field declarations may have optional initializer expressions, as for local variables, §5.5. `var` fields without an initializer are initialized with the default value of their type. `val` fields without an initializer must be initialized by each constructor.

For `val` fields, as for `val` local variables, the type may be omitted and inferred from the initializer type (§4.12). `var` fields, like `var` local variables, must be declared with a type.

6 Names and packages

6.1 Names

An X10 program consists largely of giving names to entities, and then manipulating the entities by their names. The entities involved may be compile-time constructs, like packages, types and classes, or run-time constructs, like numbers and strings and objects.

X10 names can be *simple names*, which look like identifiers: `vj`, `x10`, `AndSoOn`. Or, they can be *qualified names*, which are sequences of two or more identifiers separated by dots: `x10.lang.String`, `somePack.someType`, `a.b.c.d.e.f`. Some entities have only simple names; some have both simple and qualified names.

Every declaration that introduces a name has a *scope*: the region of the program in which the named entity can be referred to by a simple name. In some cases, entities may be referred to by qualified names outside of their scope. *E.g.*, a `public` class `C` defined in package `p` can be referred to by the simple name `C` inside of `p`, or by the qualified name `p.C` from anywhere.

Many sorts of entities have *members*. Packages have classes, structs, and interfaces as members. Those, in turn, have fields, methods, types, and so forth as members. The member `x` of an entity named `E` (as a simple or qualified name) has the name `E.x`; it may also have other names.

6.1.1 Shadowing

One declaration d may *shadow* another declaration d' in part of the scope of d', if d and d' declare variables with the same simple name n. When d shadows d', a use of n might refer to d's n (unless some d'' in turn shadows d), but will never refer to d''s n.

X10 has four namespaces:

- **Types:** for classes, interfaces, structs, and defined types.
- **Values:** for `val`- and `var`-bound variables; fields; and formal parameters of all sorts.

- **Methods:** for methods of classes, interfaces, and structs.
- **Packages:** for packages.

A declaration d in one namespace, binding a name n to an entity e, shadows all other declarations of that name n in scope at the point where d is declared. This shadowing is in effect for the entire scope of d. Declarations in different namespaces do not shadow each other. Thus, a local variable declaration may shadow a field declaration, but not a class declaration.

Declarations which only introduce qualified names — in X10, this is only package declarations — cannot shadow anything.

The rules for shadowing of imported names are given in §6.4.

6.1.2 Hiding

Shadowing is ubiquituous in X10. Another, and considerably rarer, way that one definition of a given simple name can render another definition of the same name unavailable is *hiding*. If a class `Super` defines a field named `x`, and a subclass `Sub` of `Super` also defines a field named `x`, and `b:Sub`, then `b.x` is `Sub`'s `x` field, not `Super`'s. In this case, `Super`'s `x` is said to be *hidden*.

Hiding is technically different from shadowing, because hiding applies in more circumstances: a use of class `Sub`, such as `sub.x`, may involve hiding of name `x`, though it could not involve shadowing of `x` because `x` need not be declared as a name at that point.

6.1.3 Obscuring

The third way in which a definition of a simple name may become unavailable is *obscuring*. This well-named concept says that, if `n` can be interpreted as two or more of: a variable, a type, and a package, then it will be interpreted as a variable if that is possible, or a type if it cannot be interpreted as a variable. In this case, the unavailable interpretations are *obscured*.

Example: *In the* `example` *method of the following code, both a struct and a local variable are named* `eg`*. Following the obscuring rules, the call* `eg.ow()` *in the first* `assert` *uses the variable rather than the struct. As the second* `assert` *demonstrates, the struct can be accessed through its fully-qualified name. Note that none of this would have happened if the coder had followed the convention that structs have capitalized names,* `Eg`*, and variables have lower-case ones,* `eg`*.*

```
package obscuring;
struct eg {
   static def ow()= 1;
   static struct Bite {
```

```
        def ow() = 2;
    }
    def example() {
        val eg = Bite();
        assert eg.ow() == 2;
        assert obscuring.eg.ow() == 1;
      }
}
```

Due to obscuring, it may be impossible to refer to a type or a package via a simple name in some circumstances. Obscuring does not block qualified names.

6.1.4 Ambiguity and Disambiguation

Neither simple nor qualified names are necessarily unique. There can be, in general, many entities that have the same name. This is perfectly ordinary, and, when done well, considered good programming practice. Various forms of *disambiguation* are used to tell which entity is meant by a given name; *e.g.*, methods with the same name may be disambiguated by the types of their arguments (§8.12).

Example: *In the following example, there are three static methods with qualified name* `DisambEx.Example.m`*; they can be disambiguated by their different arguments. Inside the body of the third, the simple name* `i` *refers to both the* `Long` *formal of* `m`*, and to the static method* `DisambEx.Example.i`.

```
package DisambEx;
class Example {
  static def m() = 1;
  static def m(Boolean) = 2;
  static def i() = 3;
  static def m(i:Long) {
    if (i > 10) {
      return i() + 1;
    }
    return i;
  }
  static def example() {
    assert m() == 1;
    assert m(true) == 2;
    assert m(3) == 3;
    assert m(20) == 4;
  }
}
```

6.2 Access Control

X10 allows programmers *access control*, that is, the ability to determine statically where identifiers of most sorts are visible. In particular, X10 allows *information hiding*, wherein certain data can be accessed from only limited parts of the program.

There are four access control modes: `public` , `protected`, `private` and uninflected package-specific scopes, much like those of Java. Most things can be public or private; a few things (*e.g.*, class members) can also be protected or package-scoped.

Accessibility of one X10 entity (package, container, member, etc.) from within a package or container is defined as follows:

- Packages are always accessible.
- If a container `C` is public, and, if it is inside of another container `D`, container `D` is accessible, then `C` is accessible.
- A member `m` of a container `C` is accessible from within another container `E` if `C` is accessible, and:
  - `m` is declared `public`; or
  - `C` is an interface; or
  - `m` is declared `protected`, and either the access is from within the same package that `C` is defined in, or from within the body of a subclass of `C` (but see §6.2.1 for some fine points); or
  - `m` is declared `private`, and the access is from within the top-level class which contains the definition of `C` — which may be `C` itself, or, if `C` is a nested container, an outer class around `C`; or
  - `m` has no explicit class declaration (hence using the implicit "package"-level access control), and the access occurs from the same package that `C` is declared in.

6.2.1 Details of `protected`

`protected` access has a few fine points. Within the body of a subclass `D` of the class `C` containing the definition of a protected member `m`,

- An access `e.fld` to a field, or `e.m(...)` to a method, is permitted precisely when the type of `e` is either `D` or a subtype of `D`. For example, the access to `that.f` in the following code is acceptable, but the access to `xhax.f` is not.

```
class C {
  protected var f : Long = 0;
}
class X extends C {}
```

```
class D extends C {
  def usef(that:D, xhax:X) {
     this.f += that.f;
     // ERROR: this.f += xhax.f;
  }
}
```

Limitation: The X10 compiler improperly allows access to `xhax` – as, indeed, some Java compilers do, despite Java having the analogous rule. The compiler allows you to do everything the spec says and a bit more.

- An access through a qualified name `Q.N` is permitted precisely when the type of `Q` is `D` or a subtype of `D`.

Qualified access to a protected constructor is subtle. Let `C` be a class with a `protected` constructor c, and let `S` be the innermost class containing a use u of c. There are three cases for u:

- Superclass construction invocations, `super(...)` or `E.super(...)`, are permitted.
- Anonymous class instance creations, of the forms `new C(...){...}` and `E.new C(...){...}`, are permitted.
- No other accesses are permitted.

6.3 Packages

A package is a named collection of top-level type declarations, *viz.*, class, interface, and struct declarations. Package names are sequences of identifiers, like `x10.lang` and `com.ibm.museum`. The multiple names are simply a convenience, though there is a tenuous relationship between packages `a`, `a.b`, and `a.c`. Packages can be accessed by name from anywhere: a package may contain private elements, but may not itself be private.

Packages and protection modifiers determine which top-level names can be used where. Only the `public` members of package `pack.age` can be accessed outside of `pack.age` itself.

```
package pack.age;
class Deal {
  public def make() {}
}
public class Stimulus {
  private def taxCut() = true;
  protected def benefits() = true;
```

```
  public def jobCreation() = true;
  /*package*/ def jumpstart() = true;
}
```

The class `Stimulus` can be referred to from anywhere outside of `pack.age` by its full name of `pack.age.Stimulus`, or can be imported and referred to simply as `Stimulus`. The public `jobCreation()` method of a `Stimulus` can be referred to from anywhere as well; the other methods have smaller visibility. The non-`public` class `Deal` cannot be used from outside of `pack.age`.

6.3.1 Name Collisions

It is a static error for a package to have two members with the same name. For example, package `pack.age` cannot define two classes both named `Crash`, nor a class and an interface with that name.

Furthermore, `pack.age` cannot define a member `Crash` if there is another package named `pack.age.Crash`, nor vice-versa. (This prohibition is the only actual relationship between the two packages.) This prevents the ambiguity of whether `pack.age.Crash` refers to the class or the package. Note that the naming convention that package names are lower-case and package members are capitalized prevents such collisions.

6.4 `import` Declarations

Any public member of a package can be referred to from anywhere through a fully-qualified name: `pack.age.Stimulus`.

Often, this is too awkward. X10 has two ways to allow code outside of a class to refer to the class by its short name (`Stimulus`): single-type imports and on-demand imports.

Imports of either kind appear at the start of the file, immediately after the `package` directive if there is one; their scope is the whole file.

6.4.1 Single-Type Import

The declaration `import` *TypeName* `;` imports a single type into the current namespace. The type it imports must be a fully-qualified name of an extant type, and it must either be in the same package (in which case the `import` is redundant) or be declared `public`.

Furthermore, when importing `pack.age.T`, there must not be another type named `T` at that point: neither a `T` declared in `pack.age`, nor a `inst.ant.T` imported from some other package.

The declaration `import E.n;`, appearing in file f of a package named `P`, shadows the following types named `n` when they appear in f:

- Top-level types named `n` appearing in other files of `P`, and
- Types named `n` imported by automatic imports (§6.4.2) in f.

6.4.2 Automatic Import

The automatic import `import pack.age.*;`, loosely, imports all the public members of `pack.age`. In fact, it does so somewhat carefully, avoiding certain errors that could occur if it were done naively. Types defined in the current package, and those imported by single-type imports, shadow those imported by automatic imports. If two automatic imports provide the same short name `n`, it is an error to use `n` – but it is not an error if no conflicting name is ever used. Names automatically imported never shadow any other names.

6.4.3 Implicit Imports

The package `x10.lang` is automatically imported in all files without need for further specification. Furthermore, the public static members of the class named `_` in `x10.lang` are imported everywhere as well. This provides a number of aliases, such as `Console` and `int` for `x10.io.Console` and `Int`.

6.5 Conventions on Type Names

TypeName	::=	*Id*	*(20.172)*
	|	*TypeName* `.` *Id*	
PackageName	::=	*Id*	*(20.128)*
	|	*PackageName* `.` *Id*	

While not enforced by the compiler, classes and interfaces in the X10 library follow 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. (Type variables are often single capital letters, such as `T`.) 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 `static val` fields are in all uppercase with words separated by `_`'s.

7 Interfaces

An interface specifies signatures for zero or more public methods, property methods, `static val`s, classes, structs, interfaces, types and an invariant.

The following puny example illustrates all these features:

```
interface Pushable{prio() != 0} {
  def push(): void;
  static val MAX_PRIO = 100;
  abstract class Pushedness{}
  struct Pushy{}
  interface Pushing{}
  static type Shove = Long;
  property text():String;
  property prio():Long;
}
class MessageButton(text:String)
  implements Pushable{self.prio()==Pushable.MAX_PRIO} {
  public def push() {
    x10.io.Console.OUT.println(text + " pushed");
  }
  public property text() = text;
  public property prio() = Pushable.MAX_PRIO;
}
```

`Pushable` defines two property methods, one normal method, and a static value. It also establishes an invariant, that `prio() != 0`. `MessageButton` implements a constrained version of `Pushable`, *viz.* one with maximum priority. It defines the `push()` method given in the interface, as a `public` method—interface methods are implicitly `public`.

Limitation: X10 may not always detect that type invariants of interfaces are satisfied, even when they obviously are.

A container—a class or struct—can *implement* an interface, typically by having all the methods and property methods that the interface requires, and by providing a suitable `implements` clause in its definition.

A variable may be declared to be of interface type. Such a variable has all the property and normal methods declared (directly or indirectly) by the interface; nothing else is statically available. Values of any concrete type which implement the interface may be stored in the variable.

Example: *The following code puts two quite different objects into the variable* `star`*, both of which satisfy the interface* `Star`*.*

```
interface Star { def rise():void; }
class AlphaCentauri implements Star {
   public def rise() {}
}
class ElvisPresley implements Star {
   public def rise() {}
}
class Example {
   static def example() {
      var star : Star;
      star = new AlphaCentauri();
      star.rise();
      star = new ElvisPresley();
      star.rise();
   }
}
```

An interface may extend several interfaces, giving X10 a large fraction of the power of multiple inheritance at a tiny fraction of the cost.

Example:

```
interface Star{}
interface Dog{}
class Sirius implements Dog, Star{}
class Lassie implements Dog, Star{}
```

7.1 Interface Syntax

InterfaceDecln	::=	*Mods*$^?$ `interface` *Id* *TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *ExtendsInterfaces*$^?$ *InterfaceBody*	*(20.99)*
TypeParamsI	::=	`[` *TypeParamIList* `]`	*(20.177)*
Guard	::=	*DepParams*	*(20.83)*
ExtendsInterfaces	::=	`extends` *Type*	*(20.66)*
	|	*ExtendsInterfaces* `,` *Type*	
InterfaceBody	::=	`{` *InterfaceMemberDeclns*$^?$ `}`	*(20.98)*
InterfaceMemberDecln	::=	*MethodDecln*	*(20.100)*
	|	*PropMethodDecln*	
	|	*FieldDecln*	
	|	*TypeDecln*	

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

A class `C` implements an interface `I` if `I`, or a subtype of `I`, appears in the `implements` list of `C`. In this case, `C` implicitly gets all the methods and property methods of `I`, as `abstract public` methods. If `C` does not declare them explicitly, then they are `abstract`, and `C` must be `abstract` as well. If `C` does declare them all, `C` may be concrete.

If `C` implements `I`, then the class invariant (§8.9) for `C`, $inv(\texttt{C})$, implies the class invariant for `I`, $inv(\texttt{I})$. That is, if the interface `I` specifies some requirement, then every class `C` that implements it satisfies that requirement.

7.2 Access to Members

All interface members are `public`, whether or not they are declared public. There is little purpose to non-public methods of an interface; they would specify that implementing classes and structs have methods that cannot be seen.

7.3 Member Specification

An interface can specify that all containers implementing it must have certain instance methods. It cannot require constructors or static methods, though.

Example: *The* `Stat` *interface requires that its implementers provide an* `ick` *method. It can't insist that implementations provide a static method like* `meth`*, or a nullary constructor.*

```
interface Stat {
  def ick():void;
```

```
  // ERROR: static def meth():Long;
  // ERROR: static def this();
}
class Example implements Stat {
  public def ick() {}
  def example() {
     this.ick();
  }
}
```

7.4 Property Methods

An interface may declare `property` methods. All non-`abstract` containers implementing such an interface must provide all the `property` methods specified.

7.5 Field Definitions

An interface may declare a `val` field, with a value. This field is implicitly `public static val`. In particular, it is *not* an instance field.

```
interface KnowsPi {
  PI = 3.14159265358;
}
```

Classes and structs implementing such an interface get the interface's fields as `public static` fields. Unlike methods, there is no need for the implementing class to declare them.

```
class Circle implements KnowsPi {
  static def area(r:Double) = PI * r * r;
}
class UsesPi {
  def circumf(r:Double) = 2 * r * KnowsPi.PI;
}
```

7.5.1 Fine Points of Fields

If two parent interfaces give different static fields of the same name, those fields must be referred to by qualified names.

```
interface E1 {static val a = 1;}
interface E2 {static val a = 2;}
interface E3 extends E1, E2{}
```

```
class Example implements E3 {
  def example() = E1.a + E2.a;
}
```

If the *same* field `a` is inherited through many paths, there is no need to disambiguate it:

```
interface I1 { static val a = 1;}
interface I2 extends I1 {}
interface I3 extends I1 {}
interface I4 extends I2,I3 {}
class Example implements I4 {
  def example() = a;
}
```

The initializer of a field in an interface may be any expression. It is evaluated under the same rules as a `static` field of a class.

Example: *In this example, a class* `TheOne` *is defined, with an inner interface* `WelshOrFrench`*, whose field* `UN` *(named in either Welsh or French) has value 1. Note that* `WelshOrFrench` *does not define any methods, so it can be trivially added to the* `implements` *clause of any class, as it is for* `Onesome`*. This allows the body of* `Onesome` *to use* `UN` *through an unqualified name, as is done in* `example()`.

```
class TheOne {
  static val ONE = 1;
  interface WelshOrFrench {
    val UN = 1;
  }
  static class Onesome implements WelshOrFrench {
    static def example() {
      assert UN == ONE;
    }
  }
}
```

7.6 Generic Interfaces

Interfaces, like classes and structs, can have type parameters. The discussion of generics in §4.3 applies to interfaces, without modification.

Example:

```
interface ListOfFuns[T,U] extends x10.util.List[(T)=>U] {}
```

7.7 Interface Inheritance

The *direct superinterfaces* of a non-generic interface `I` are the interfaces (if any) mentioned in the `extends` clause of `I`'s definition. If `I` is generic, the direct superinterfaces are of an instantiation of `I` are the corresponding instantiations of those interfaces. A *superinterface* of `I` is either `I` itself, or a direct superinterface of a superinterface of `I`, and similarly for generic interfaces.

`I` inherits the members of all of its superinterfaces. Any class or struct that has `I` in its `implements` clause also implements all of `I`'s superinterfaces.

Classes and structs may be declared to implement multiple interfaces. Semantically, the interface type is the set of all objects that are instances of classes or structs that implement the interface. A class or struct implements an interface if it is declared to and if it concretely or abstractly implements all the methods and properties defined in the interface. For example, `Kim` implements `Person`, and hence `Named` and `Mobile`. It would be a static error if `Kim` had no `name` method, unless `Kim` were also declared `abstract`.

```
class Kim implements Person {
   var pos : Long = 0;
   public def name() = "Kim (" + pos + ")";
   public def move(dPos:Long) { pos += dPos; }
}
```

7.8 Members of an Interface

The members of an interface `I` are the union of the following sets:

1. All of the members appearing in `I`'s declaration;
2. All the members of its direct super-interfaces, except those which are hidden (§6.1.2) by `I`
3. The members of `Any`.

Overriding for instances is defined as for classes, §8.4.8

8 Classes

8.1 Principles of X10 Objects

8.1.1 Basic Design

Objects are instances of classes: the most common and most powerful sort of value in X10. The other kinds of values, structs and functions, are more specialized.

Classes are structured in a forest of single-inheritance code hierarchies. Like C++, but unlike Java, there is no single root class (e.g. `java.lang.Object`) that all classes inherit from. Classes may have any or all of these features:

- Implementing any number of interfaces;
- Static and instance `val` fields;
- Instance `var` fields;
- Static and instance methods;
- Constructors;
- Properties;
- Static and instance nested containers.
- Static type definitions

X10 objects (unlike Java objects) do not have locks associated with them. Programmers may use atomic blocks (§14.7) for mutual exclusion and clocks (§15) for sequencing multiple parallel operations.

An object exists in a single location: the place that it was created. One place cannot use or even directly refer to an object in a different place. A special type, `GlobalRef[T]`, allows explicit cross-place references.

The basic operations on objects are:

- Construction (§8.11). Objects are created, their `var` and `val` fields initialized, and other invariants established.

- Field access (§11.4). The static, instance, and property fields of an object can be retrieved; `var` fields can be set.

- Method invocation (§11.6). Static, instance, and property methods of an object can be invoked.

- Casting (§11.22) and instance testing with `instanceof` (§11.24) Objects can be cast or type-tested.

- The equality operators == and `!=`. Objects can be compared for equality with the == operation. This checks object *identity*: two objects are == iff they are the same object.

8.1.2 Class Declaration Syntax

The *class declaration* has a list of type parameters, a list of properties, a constraint (the *class invariant*), zero or one superclass, zero or more interfaces that it implements, and a class body containing the the definition of fields, properties, methods, and member types. Each such declaration introduces a class type (§4.2).

ClassDecln	::=	*Mods*$^?$ `class` *Id TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *Super*$^?$ *Interfaces*$^?$ *ClassBody*	*(20.34)*
TypeParamsI	::=	`[` *TypeParamIList* `]`	*(20.177)*
TypeParamIList	::=	*TypeParam*	*(20.174)*
	|	*TypeParamIList* `,` *TypeParam*	
	|	*TypeParamIList* `,`	
Properties	::=	`(` *PropList* `)`	*(20.142)*
PropList	::=	*Prop*	*(20.140)*
	|	*PropList* `,` *Prop*	
Prop	::=	*Annotations*$^?$ *Id ResultType*	*(20.139)*
Guard	::=	*DepParams*	*(20.83)*
Super	::=	`extends` *ClassType*	*(20.156)*
Interfaces	::=	`implements` *InterfaceTypeList*	*(20.103)*
InterfaceTypeList	::=	*Type*	*(20.102)*
	|	*InterfaceTypeList* `,` *Type*	
ClassBody	::=	`{` *ClassMemberDeclns*$^?$ `}`	*(20.33)*
ClassMemberDeclns	::=	*ClassMemberDecln*	*(20.36)*
	|	*ClassMemberDeclns ClassMemberDecln*	
ClassMemberDecln	::=	*InterfaceMemberDecln*	*(20.35)*
	|	*CtorDecln*	

8.2 Fields

Objects may have *instance fields*, or simply *fields* (called "instance variables" in C++ and Smalltalk, and "slots" in CLOS): places to store data that is pertinent to the object. Fields, like variables, may be mutable (`var`) or immutable (`val`).

A class may have *static fields*, which store data pertinent to the entire class of objects. See §8.6 for more information. Because of its emphasis on safe concurrency, X10 requires static fields to be immutable (`val`).

No two fields of the same class may have the same name. A field may have the same name as a method, although for fields of functional type there is a subtlety (§8.12.4).

8.2.1 Field Initialization

Fields may be given values via *field initialization expressions*: `val f1 = E;` and `var f2 : Long = F;`. Other fields of `this` may be referenced, but only those that *precede* the field being initialized.

Example: *The following is correct, but would not be if the fields were reversed:*

```
class Fld{
  val a = 1;
  val b = 2+a;
}
```

8.2.2 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`.

With inner classes, it is occasionally necessary to write `Cls.super.f` to get at a hidden field `f` of an outer class `Cls`.

Example: *The* `f` *field in* `Sub` *hides the* `f` *field in* `Super` *The* `superf` *method provides access to the* `f` *field in* `Super`.

```
class Super{
  public val f = 1;
}
class Sub extends Super {
  val f = true;
  def superf() : Long = super.f; // 1
}
```

Example: *Hidden fields of outer classes can be accessed by suitable forms:*

```
class A {
   val f = 3;
}
class B extends A {
   val f = 4;
   class C extends B {
      // C is both a subclass and inner class of B
      val f = 5;
       def example() {
         assert f         == 5 : "field of C";
         assert super.f   == 4 : "field of superclass";
         assert B.this.f  == 4 : "field of outer instance";
         assert B.super.f == 3 : "super.f of outer instance";
       }
    }
}
```

8.2.3 Field qualifiers

The behavior of a field may be changed by a field qualifier, such as `static` or `transient`.

`static` qualifier

A `val` field may be declared to be *static*, as described in §8.2.

`transient` Qualifier

A field may be declared to be *transient*. Transient fields are excluded from the deep copying that happens when information is sent from place to place in an `at` statement. The value of a transient field of a copied object is the default value of its type, regardless of the value of the field in the original. If the type of a field has no default value, it cannot be marked `transient`.

```
class Trans {
   val copied = "copied";
   transient var transy : String = "a very long string";
   def example() {
      at (here) { // causes copying of 'this'
         assert(this.copied.equals("copied"));
         assert(this.transy == null);
      }
   }
}
```

8.3 Properties

The properties of an object (or struct) are a restricted form of public `val` fields.[1] For example, every array has a `rank` telling how many subscripts it takes. User-defined classes can have whatever properties are desired.

Properties differ from public `val` fields in a few ways:

1. Property references are allowed on `self` in constraints: `self.prop`. Field references are not.
2. Properties are in scope for all instance initialization expressions. `val` fields are not.
3. The graph of values reachable from a given object by following only property links is acyclic. Conversely, it is possible (and routine) for two objects to point to each other with `val` fields.
4. Properties are declared in the class header; `val` fields are defined in the class body.
5. Properties are set in constructors by a `property` statement. `val` fields are set by assignment.

Properties are defined in parentheses, after the name of the class. They are given values by the `property` command in constructors.

Example: `Proper` *has a single property,* `t`. `new Proper(4)` *creates a* `Proper` *object with* `t==4`.

```
class Proper(t:Long) {
  def this(t:Long) {property(t);}
}
```

It is a static error for a class defining a property `x: T` to have a subclass class that defines a property or a field with the name `x`.

A property `x:T` induces a field with the same name and type, as if defined with:

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

Properties are initialized in a constructor by the invocation of a special `property` statement. The requirement to use the `property` statement means that all properties must be given values at the same time: a container either has its properties or it does not.

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

[1] In many cases, a `val` field can be upgraded to a `property`, which entails no compile-time or runtime cost. Some cannot be, *e.g.*, in cases where cyclic structures of `val` fields are required.

The number and types of arguments to the `property` statement must match the number and types of the properties in the class declaration, in order. Every constructor of a class with properties must invoke `property(...)` precisely once; it is a static error if X10 cannot prove that this holds.

By construction, the graph whose nodes are values and whose edges are properties is acyclic. *E.g.*, there cannot be values `a` and `b` with properties `c` and `d` such that `a.c == b` and `b.d == a`.

Example:

```
class Proper(a:Long, b:String) {
  def this(a:Long, b:String) {
      property(a, b);
  }
  def this(z:Long) {
      val theA = z+5;
      val theB = "X"+z;
      property(theA, theB);
  }
  static def example() {
      val p = new Proper(1, "one");
      assert p.a == 1 && p.b.equals("one");
      val q = new Proper(10);
      assert q.a == 15 && q.b.equals("X10");
  }
}
```

8.3.1 Properties and Field Initialization

Fields with explicit initializers are evaluated immediately after the `property` command, and all properties are in scope when initializers are evaluated.

Example: *Class* `Init` *initializes the field* `a` *to be a rail of* `n` *elements, where* `n` *is a property. When* `new Init(4)` *is executed, the constructor first sets* `n` *to* `4` *via the* `property` *statement, and then initializes* `a` *to a 4-element rail.*

However, `Outit` *uses a field rather than a property for* `n`*. If the* `ERROR` *line were present, it would not compile.* `n` *has not been definitely assigned (§19) at this point, and* `n` *has not been given its value, so* `a` *cannot be computed. (If one insisted that* `n` *be a property,* `a` *would have to be initialized in the constructor, rather than by an initialization expression.)*

```
class Init(n:Long) {
  val a = new Rail[String](n, "");
  def this(n:Long) { property(n); }
}
class Outit {
```

```
    val n : Long;
    //ERROR: val a = new Rail[String](n, "");
    def this(m:Long) { this.n = m; }
  }
```

8.3.2 Properties and Fields

A container with a property named `p`, or a nullary property method named `p()`, cannot have a field named `p` — either defined in that container, or inherited from a superclass.

8.3.3 Acyclicity of Properties

X10 has certain restrictions that, ultimately, require that properties are simpler than their containers. For example, `class A(a:A){}` is not allowed. Formally, this requirement is that there is a total order $\preceq$ on all classes and structs such that, if A extends B, then $A \prec B$, and if A has a property of type B, then $A \prec B$, where $A \prec B$ means $A \preceq B$ and $A \neq B$. For example, the preceding class `A` is ruled out because we would need `A`$\prec$`A`, which violates the definition of $\prec$. The programmer need not (and cannot) specify $\preceq$, and rarely need worry about its existence.

Similarly, the type of a property may not simply be a type parameter. For example, `class A[X](x:X){}` is illegal.

8.4 Methods

As is common in object-oriented languages, objects can have *methods*, of two sorts. *Static methods* are functions, conceptually associated with a class and defined in its namespace. *Instance methods* are parameterized code bodies associated with an instance of the class, which execute with convenient access to that instance's fields.

Each method has a *signature*, telling what arguments it accepts, what type it returns, and what precondition it requires. Method definitions may be overridden by subclasses; the overriding definition may have a declared return type that is a subtype of the return type of the definition being overridden. Multiple methods with the same name but different signatures may be provided on a class (called "overloading" or "ad hoc polymorphism"). Methods may be declared `public`, `private`, `protected`, or given default package-level access rights.

MethMods	::=	*Mods*$^?$	*(20.115)*
	|	*MethMods* `property`	
	|	*MethMods Mod*	
MethodDecln	::=	*MethMods* `def` *Id TypeParams*$^?$ *Formals Guard*$^?$ *Throws*$^?$ *HasResultType*$^?$ *MethodBody*	*(20.117)*
	|	*BinOpDecln*	
	|	*PrefixOpDecln*	
	|	*ApplyOpDecln*	
	|	*SetOpDecln*	
	|	*ConversionOpDecln*	
	|	*KeywordOpDecln*	
TypeParams	::=	`[` *TypeParamList* `]`	*(20.176)*
Formals	::=	`(` *FormalList*$^?$ `)`	*(20.80)*
FormalList	::=	*Formal*	*(20.79)*
	|	*FormalList* `,` *Formal*	
Throws	::=	`throws` *ThrowList*	*(20.84)*
ThrowsList	::=	*Type*	*(20.85)*
	|	*ThrowsList* `,` *Type*	
HasResultType	::=	*ResultType*	*(20.86)*
	|	`<:` *Type*	
MethodBody	::=	`=` *LastExp* `;`	*(20.116)*
	|	*Annotations*$^?$ *Block*	
	|	`;`	
BinOpDecln	::=	*MethMods* `operator` *TypeParams*$^?$ `(` *Formal* `)` *BinOp* `(` *Formal* `)` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	*(20.24)*
	|	*MethMods* `operator` *TypeParams*$^?$ `this` *BinOp* `(` *Formal* `)` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ `(` *Formal* `)` *BinOp* `this` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	
PrefixOpDecln	::=	*MethMods* `operator` *TypeParams*$^?$ *PrefixOp* `(` *Formal* `)` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	*(20.137)*
	|	*MethMods* `operator` *TypeParams*$^?$ *PrefixOp* `this` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	
ApplyOpDecln	::=	*MethMods* `operator` `this` *TypeParams*$^?$ *Formals* *Guard*$^?$ *HasResultType*$^?$ *MethodBody*	*(20.7)*
ConversionOpDecln	::=	*ExplConvOpDecln*	*(20.51)*
	|	*ImplConvOpDecln*	

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

8.4.1 Forms of Method Definition

There are several syntactic forms for defininining methods. The forms that include a block, such as `def m(){S}`, allow an arbitrary block. These forms can define a `void` method, which does not return a value.

The forms that include an expression, such as `def m()=E`, require a syntactically and semantically valid expression. These forms cannot define a `void` method, because expressions cannot be `void`.

There are no other semantic differences between the two forms.

8.4.2 Method Return Types

A method with an explicit return type returns values of that type. A method without an explicit return type is given a return type by type inference. A *call* to a method has type given by substituting information about the actual `val` parameters for the formals.

Example:

In the example below, `met1` *has an explicit return type* `Ret{n==a}`. `met2` *does not, so its return type is computed, also to be* `Ret{n==a}`*, because that's what the implicitly-defined constructor returns.*

`use3` *requires that its argument have* `n==3`. `example` *shows that both* `met1` *and* `met2` *can be used to produce such an object. In both cases, the actual argument* `3` *is substituted for the formal argument* `a` *in the return type expression for the method* `Ret{n==a}`*, giving the type* `Ret{n==3}` *as required by* `use3`.

```
class Ret(n:Long) {
  static def met1(a:Long) : Ret{n==a} = new Ret(a);
  static def met2(a:Long)             = new Ret(a);
  static def use3(Ret{n==3}) {}
  static def example() {
     use3(met1(3));
     use3(met2(3));
  }
}
```

8.4.3 Throws Clause

The `throws` clause indicates what checked exceptions may be raised during the execution of the method and are not handled by `catch` blocks within the method. If a checked exception may escape from the method, then it must be by a subtype of one of the types listed in the `throws` clause of the method. Checked exceptions are defined to be any subclass of `x10.lang.CheckedThrowable` that are not also subclasses of either `x10.lang.Exception` or `x10.lang.Error`.

If a method is implementing an interface or overriding a superclass method the set of types represented by its `throws` clause must by a (potentially improper) subset of the types of the `throws` clause of the method it is overriding.

8.4.4 Final Methods

An instance method may be given the `final` qualifier. `final` methods may not be overridden.

8.4.5 Generic Instance Methods

Limitation: In X10, an instance method may be generic:

```
class Example {
  def example[T](t:T) = "I like " + t;
}
```

However, the C++ back end does not currently support generic virtual instance methods like `example`. It does allow generic instance methods which are `final` or `private`, and it does allow generic static methods.

8.4.6 Method Guards

Often, a method will only make sense to invoke under certain statically-determinable conditions. These conditions may be expressed as a guard on the method.

Example: *For example,* `example(x)` *is only well-defined when* `x != null`*, as* `null.toString()` *throws a null pointer exception, and returns nothing:*

```
class Example {
   var f : String = "";
   def setF(x:Any){x != null} : void {
      this.f = x.toString();
   }
}
```

(We could have used a constrained type `Any{self!=null}` *for* `x` *instead; in most cases it is a matter of personal preference or convenience of expression which one to use.)*

The requirement of having a method guard is that callers must demonstrate to the X10 compiler and/or runtime that the guard is satisfied. With the `STATIC_CHECKS` compiler option in force (§C.1.3), this is checked at compile time, and there is no runtime cost. Indeed, this code can be more efficient than usual, as it is statically provable that `x != null`.

When STATIC_CHECKS is not in force, dynamic checks are generated as needed; method guards are checked at runtime. This is potentially more expensive, but may be more convenient.

Example: *The following code fragment contains a line which will not compile with* STATIC_CHECKS *on (assuming the guarded* example *method above). (X10's type system does not attempt to propagate information from* if*s.) It will compile with* STATIC_CHECKS *off, but it may insert an extra* null*-test for* x*.*

```
def exam(e:Example, x:Any) {
  if (x != null)
     e.example(x as Any{x != null});
     // If STATIC_CHECKS is in force:
     // ERROR: if (x != null) e.example(x);
}
```

The guard {c} in a guarded method def m(){c} = E; specifies a constraint c on the properties of the class C on which the method is being defined. The method, in effect, only exists 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}.

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

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

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

Limitation: Using a reference to an outer class, Outer.this, in a constraint, is not supported.

8.4.7 Property methods

PropMethodDecln ::= *Mods*$^?$ property *Id TypeParams*$^?$ *Formals Guard*$^?$ *Throws*$^?$ *HasResultType*$^?$ *MethodBody* (20.141)
| *Mods*$^?$ property *Id Guard*$^?$ *HasResultType*$^?$ *MethodBody*

Property methods are methods that can be evaluated in constraints, as properties can. They provide a means of abstraction over properties; *e.g.*, interfaces can specify property methods that implementing containers must provide, but, just as they cannot specify ordinary fields, they cannot specify property fields. Property methods are very

limited in computing power: they must obey the same restrictions as constraint expressions. In particular, they cannot have side effects, or even much code in their bodies.

Example: *The* `eq()` *method below tells if the* `x` *and* `y` *properties are equal; the* `is(z)` *method tells if they are both equal to* `z`*. The* `eq` *and* `is` *property methods are used in types in the* `example` *method.*

```
class Example(x:Long, y:Long) {
   def this(x:Long, y:Long) { property(x,y); }
   property eq() = (x==y);
   property is(z:Long) = x==z && y==z;
   def example( a : Example{eq()}, b : Example{is(3)} ) {}
}
```

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 with an expression, or a single expression. It is a static error if the expression cannot be represented in the constraint system. Property methods may be `abstract` in `abstract` classes, and may be specified in interfaces, but are implicitly `final` in non-`abstract` classes.

The expression may contain invocations of other property methods. The compiler ensures that there are no circularities in property methods, so property method evaluations always terminate.

Property methods in classes are implicitly `final`; they cannot be overridden. It is a static error if a superclass has a property method with a given signature, and a subclass has a method or property method with the same signature. It is a static error if a superclass has a property with some name `p`, and a subclass has a nullary method of any kind (instance, static, or property) also named `p`.

A nullary property method definition may omit 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 `()`. If `ob`'s type has a property `p`, then `ob.p` is that property. Otherwise, if it has a nullary property method `p()`, `ob.p` is equivalent to `ob.p()`. As a consequence, if the type provides both a property `p` and a nullary method `p()`, then the property can be accessed as `ob.p` and the method as `ob.p()`.[2]

[2]This only applies to nullary property methods, not nullary instance methods. Nullary property methods perform limited computations, have no side effects, and always return the same value, since they have to be expressed in the constraint sublanguage. In this sense, a nullary property method does not behave hugely different from a property. Indeed, a compilation scheme which cached the value of the property method would all but erase the distinction. Other methods may have more behavior, *e.g.*, side effects, so we keep the `()` to make it clear that a method call is potentially complex.

`w.rail`, with either definition above, is equivalent to `w.rail()`

Limitation of Property Methods

Limitation: Currently, X10 forbids the use of property methods which have all the following features:

- they are abstract, and
- they have one or more arguments, and
- they appear as subterms in constraints.

Any two of these features may be combined, but the three together may not be.

Example: *The constraint in* `example1` *is concrete, not abstract. The constraint in* `example2` *is nullary, and has no arguments. The constraint in* `example3` *appears at the top level, rather than as a subterm (*cf. *the equality expressions* `A==B` *in the other examples). However,* `example4` *combines all three features, and is not allowed.*

```
class Cls {
  property concrete(a:Long) = 7;
}
interface Inf {
  property nullary(): Long;
  property topLevel(z:Long):Boolean;
  property allThree(z:Long):Long;
}
class Example{
  def example1(Cls{self.concrete(3)==7}) = 1;
  def example2(Inf{self.nullary()==7})   = 2;
  def example3(Inf{self.topLevel(3)})    = 3;
  //ERROR: def example4(Inf{self.allThree(3)==7}) = "fails";
}
```

8.4.8 Method overloading, overriding, hiding, shadowing and obscuring

The definitions of method overloading, overriding, hiding, shadowing and obscuring in X10 are familiar from languages such as 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 constraint-erased types (in some instantiation of the generic parameters).

Example: *The following overloading of* `m` *is unproblematic.*

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

A class definition may include methods which are ambiguous in *some* generic instantiation. (It is a compile-time error if the methods are ambiguous in *every* generic instantiation, but excluding class definitions which are are ambiguous in *some* instantiation would exclude useful cases.) It is a compile-time error to *use* an ambiguous method call.

Example: *The following class definition is acceptable. However, the marked method calls are ambiguous, and hence not acceptable.*

```
class Two[T,U]{
  def m(x:T)=1;
  def m(x:Long)=2;
  def m[X](x:X)=3;
  def m(x:U)=4;
  static def example() {
    val t12 = new Two[Long, Any]();
    // ERROR: t12.m(2);
    val t13  = new Two[String, Any]();
    t13.m("ferret");
    val t14 = new Two[Boolean,Boolean]();
    // ERROR: t14.m(true);
  }
}
```

The call `t12.m(2)` *could refer to either the* `1` *or* `2` *definition of* `m`*, so it is not allowed. The call* `t14.m(true)` *could refer to either the* `1` *or* `4` *definition, so it, too, is not allowed.*

The call `t13.m("ferret")` *refers only to the* `1` *definition. If the* `1` *definition were absent, type argument inference would make it refer to the* `3` *definition. However, X10 will choose a fully-specified call if there is one, before trying type inference, so this call unambiguously refers to* `1`*.*

X10 v2.4 does not permit overloading based on constraints. That is, the following is *not* legal, although either method definition individually is legal:

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

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

The *constraint erasure* of a type T, ce(T), is obtained by removing all the constraints outside of functions in T, specificially:

$$ce(\mathtt{T}) = \mathtt{T} \text{ if } \mathtt{T} \text{ is a container or interface} \quad (8.1)$$

$$ce(\mathtt{T}\{\mathtt{c}\}) = ce(\mathtt{T}) \quad (8.2)$$

$$ce(\mathtt{T}[\mathtt{S}_1, \ldots, \mathtt{S}_n]) = ce(\mathtt{T})[ce(\mathtt{S}_1), \ldots, ce(\mathtt{S}_n)] \quad (8.3)$$

$$ce((\mathtt{S}_1, \ldots, \mathtt{S}_n) => \mathtt{T}) = (ce(\mathtt{S}_1), \ldots, ce(\mathtt{S}_n)) => ce(\mathtt{T}) \quad (8.4)$$

Two methods are said to have *erasure equivalent signatures* 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 erasure equivalent. It is a compile-time error for there to be two methods with the same name and erasure equivalent signatures in a class (either defined in that class or in a superclass), unless the signatures are identical (without erasures) and one of the methods is defined in a superclass (in which case the superclass's method is overridden by the subclass's, and the subclass's method's return type must be a subtype of the superclass's method's return type).

In addition, the guard of an overridden method must entail the guard of the overriding method. This ensures that any virtual call to the method satisfies the guard of the callee.

Example: *In the following example, the call to* `s.recip(3)` *in* `example()` *will invoke* `Sub.recip(n)`. *The call is legitimate because* `Super.recip`*'s guard,* `n != 0`*, is satisfied by* `3`. *The guard on* `Sub.recip(n)` *is simply* `true`*, which is also satisfied. However, if we had used the* `ERROR` *line's definition, the guard on* `Sub.recip(n)` *would be* `n != 0, n != 3`*, which is not satisfied by* `3`*, so – despite the call statically type-checking – at runtime the call would violate its guard and (in this case) throw an exception.*

```
class Super {
  def recip(n:Long){n != 0} = 1.0/n;
}
class Sub extends Super{
  //ERROR: def recip(n:Long){n != 0, n != 3} = 1.0/(n * (n-3));
  def recip(m:Long){true} = 1.0/m;
}
class Example{
  static def example() {
     val s : Super = new Sub();
     s.recip(3);
  }
}
```

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 erasedly equivalent signatures. Methods are overriden on a signature-by-signature basis. It is a compile-time error if an instance method overrides a static method. (But is it permitted for an instance *field* to hide a static *field*; that's hiding (§8.2.2), not overriding, and hence totally different.)

8.5 Constructors

Instances of classes are created by the `new` expression:

ObCreationExp ::= `new` *TypeName TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClassBody*$^?$ *(20.126)*
| *Primary* `.` `new` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClassBody*$^?$
| *FullyQualifiedName* `.` `new` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClassBody*$^?$

This constructs a new object, and calls some code, called a *constructor*, to initialize the newly-created object properly.

Constructors are defined like methods, except that they must be named `this` and ordinary methods may not be. The content of a constructor body has certain capabilities (*e.g.*, `val` fields of the object may be initialized) and certain restrictions (*e.g.*, most methods cannot be called); see §8.11 for the details.

Example:

The following class provides two constructors. The unary constructor `def this(b : Long)` *allows initialization of the* `a` *field to an arbitrary value. The nullary constructor* `def this()` *gives it a default value of 10. The* `example` *method illustrates both of these calls.*

```
class C {
  public val a : Long;
  def this(b : Long) { a = b; }
  def this()        { a = 10; }
  static def example() {
     val two = new C(2);
     assert two.a == 2;
     val ten = new C();
     assert ten.a == 10;
  }
}
```

8.5.1 Automatic Generation of Constructors

Classes that have no constructors written in the class declaration are automatically given a constructor which sets the class properties and does nothing else. If this automatically-generated constructor is not valid (*e.g.*, if the class has `val` fields that

need to be initialized in a constructor), the class has no constructor, which is a static error.

Example: *The following class has no explicit constructor. Its implicit constructor is* `def this(x:Long){property(x);}` *This implicit constructor is valid, and so is the class.*

```
class C(x:Long) {
  static def example() {
    val c : C = new C(4);
    assert c.x == 4;
  }
}
```

The following class has the same default constructor. However, that constructor does not initialize `d`*, and thus is invalid. This class does not compile; it needs an explicit constructor.*

```
// THIS CODE DOES NOT COMPILE
class Cfail(x:Long) {
  val d: Long;
  static def example() {
    val wrong = new Cfail(40);
  }
}
```

8.5.2 Calling Other Constructors

The *first* statement of a constructor body may be a call of the form `this(a,b,c)` or `super(a,b,c)`. The former will execute the body of the matching constructor of the current class; the latter, of the superclass. This allows a measure of abstraction in constructor definitions; one may be defined in terms of another.

Example: *The following class has two constructors.* `new Ctors(123)` *constructs a new* `Ctors` *object with parameter 123.* `new Ctors()` *constructs one whose parameter has a default value of 100:*

```
class Ctors {
  public val a : Long;
  def this(a:Long) { this.a = a; }
  def this()      { this(100);  }
}
```

In the case of a class which implements `operator ()` — or any other constructor and application with the same signature — this can be ambiguous. If `this()` appears as the first statement of a constructor body, it could, in principle, mean either a constructor call or an operator evaluation. This ambiguity is resolved so that `this()` always

means the constructor invocation. If, for some reason, it is necessary to invoke an application operator as the first meaningful statement of a constructor, write the target of the application as `(this)`, *e.g.*, `(this)(a,b);`.

8.5.3 Return Type of Constructor

A constructor for class `C` may have a return type `C{c}`. The return type specifies a constraint on the kind of object returned. It cannot change its *class* — a constructor for class `C` always returns an instance of class `C`. If no explicit return type is specified, the constructor's return type is inferred.

Example: *The constructor* `(A)` *below, having no explicit return type, has its return type inferred.* `n` *is set by the* `property` *statement to* `1`*, so the return type is inferred as* `Ret{self.n==1}`*. The constructor* `(B)` *has* `Ret{n==self.n}` *as an explicit return type. The* `example()` *code shows both of these in action.*

```
class Ret(n:Long) {
   def this()    { property(1); }     // (A)
   def this(n:Long) : Ret{n==self.n} { // (B)
      property(n);
   }
   static def typeIs[T](x:T){}
   static def example() {
     typeIs[Ret{self.n==1}](new Ret());  // uses (A)
     typeIs[Ret{self.n==3}](new Ret(3)); // uses (B)
   }
}
```

8.6 Static initialization

Static fields in X10 are immutable and are guaranteed to be initialized before they are accessed. Static fields are initialized on a per-Place basis; thus an activity that reads a static field in two different Places may read different values for the content of the field in each Place. Static fields are not eagerly initialized, thus if a particular static field is not accessed in a given Place then the initializer expression for that field may not be evaluated in that Place.

When an activity running in a Place `P` attempts to read a static field `F` that has not yet been initialized in `P`, then the activity will evaluate the initializer expression for `F` and store the resulting value in `F`. It is guaranteed that at most one activity in each Place will attempt to evaluate the initializer expression for a given static field. If a second activity attempts to read `F` while the first activity is still executing the initializer expression the second activity will be suspended until the first activity finishes evaluating the initializer and stores the resulting value in `F`.