X10 Language Specification
Version 2.2

DRAFT — March 16, 2011

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

Please send comments to `bardb@us.ibm.com`

March 16, 2011

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

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

The X10 design team consists of Bard Bloom, David Cunningham, Robert Fuhrer, David Grove, Sreedhar Kodali, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Mikio Takeuchi, Olivier Tardieu, Yoav Zibin.

This version of the language was implemented by a team that includes the designers and Bowen Alpern, Philippe Charles, Ben Herta, Yan Li, Yuki Makino, Toshio Suganuma, Hai Chuan Wang.

Past members include Shivali Agarwal, David Bacon, Raj Barik, Ganesh Bikshandi, Bob Blainey, Perry Cheng, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Stephen Fink, Patrick Gallop, Christian Grothoff, Allan Kielstra, Sriram

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 quantity of computer cycles that one can desire and afford.

Then, one has the problem of how to use those computer cycles effectively. Programming a multiprocessor is far more agonizing than programming a uniprocessor. One can use models of computation which give somewhat of the illusion of programming a uniprocessor. Unfortunately, the models which give the closest imitations of uniprocessing are very expensive to implement, either increasing the monetary cost of the computer tremendously, or slowing it down dreadfully.

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

One response to this problem has been the advent of the *partitioned global address*

space (PGAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [3, 9]. 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 in lock-step.

X10 is a modern object-oriented programming language in the PGAS family. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end computers—for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads.

X10 is based on state-of-the-art object-oriented programming ideas primarily to take advantage of their proven flexibility and ease-of-use for a wide spectrum of programming problems. X10 takes advantage of several years of research (e.g., in the context of the Java Grande forum, [7, 1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *struct types* (such as `Int`, `Float`, `Complex` etc), supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [4]) and supports IEEE-standard floating point arithmetic. Some capabilities for supporting operator overloading are also provided.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the PGAS model with *asynchrony* (yielding the *APGAS* programming model). X10 introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more *activities*. Activities are extremely lightweight threads of execution. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. 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. The type system tracks which memory accesses are local. The programmer may introduce place casts which verify the access is local at run time. Linking with native code is supported.

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 *interfaces* (§7), *classes* (§8) and *structs* (§9). 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, y:Double;
    public def this(x:Double, y:Double) {
      this.x = x; this.y = y;
    }
    public def norm() = Math.sqrt(x*x+y*y);
  }
```

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

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

Classes and Structs   There are two kinds of concrete containers: *classes* (§8) and *structs* (§9). Concrete containers hold data in *fields*, and give concrete implementations of methods, as `Slider` and `PlainPoint` above.

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

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

X10 has no primitive classes per se. However, the standard library `x10.lang` supplies structs 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 `Int`. `of4` takes an `Int`-to-`Int` function and applies it to the number 4. So, `fourSquared` computes `of4(square)`, which is `square(4)`, which is 16, in a fairly complicated way.

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

Functions are used extensively in X10 programs. For example, a common way to construct and initialize an `Array[Int](1)` – that is, a fixed-length one-dimensional array of numbers, like an `int[]` in Java – is to pass two arguments to a factory method: the first argument being the length of the array, and the second being a function which computes the initial value of the ith element. The following code constructs a 1-dimensional array initialized to the squares of 0,1,...,9: `r(0) == 0`, `r(5)==25`, etc.

```
val r : Array[Int](1) = new Array[Int](10, square);
```

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

For example, an implementation of matrices `Mat` might have the numbers of rows and columns as properties. A little bit of care in definitions allows the definition of a + operation that works on matrices of the same shape, and * that works on matrices with appropriately matching shapes.

```
abstract class Mat(rows:Int, cols:Int) {
  static type Mat(r:Int, c:Int) = 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:Int, b:Int, c:Int) {
  val axb1 : Mat(a,b) = makeMat(a,b);
  val axb2 : Mat(a,b) = makeMat(a,b);
  val bxc  : Mat(b,c) = makeMat(b,c);
  val axc  : Mat(a,c) = (axb1 +axb2) * bxc;
}
```

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

A constrained type looks like `Mat{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.

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

---

[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 *.

thus possible to make a list of integers `List[Int]`, a list of non-zero integers `List[Int{self != 0}]`, or a list of people `List[Person]`. In the definition of `List`, `T` is a type parameter; it can be instantiated with any type.

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

2.2 The sequential core of X10

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

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

2.3 Places and activities

The full power of X10 starts to emerge with concurrency. An X10 program is intended to run on a wide range of computers, from uniprocessors to large clusters

of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of *place* (§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 Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of heterogeneous activities in an X10 computation. X10 allows multiple barriers in a form that supports determinate, deadlock-free parallel computation, via the `Clock` type.

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

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

2.5 Arrays, regions and distributions

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

2.6 Annotations

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

2.7   Translating MPI programs to X10

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

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

2.8   Summary and future work

2.8.1   Design for scalability

X10 is designed for scalability, by encouraging working with local data, and limiting the ability of events at one place to delay those at another. For example, an activity may atomically access only multiple locations in the current place. Unconditional atomic blocks are 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.

2.8.2   Design for productivity

X10 is designed for productivity.

Safety and correctness. Programs written in X10 are guaranteed to be statically *type safe*, *memory safe* and *pointer safe*.

Static type safety guarantees that 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.

Because places are reflected in the type system, static type safety also implies *place safety*. All operations that need to be performed locally are, in fact, performed locally. All data which is declared to be stored locally are, in fact, stored locally.

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

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

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

2.8.3   Conclusion

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

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

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

Identifiers   Identifiers 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(`, `0xbabe`, `\`\\`

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

Keywords   X10 uses the following keywords:

abstract	**as**	**assert**
async	**at**	**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 in *italics* are *soft* keywords, which may also be used as identifiers. The other keywords, in **bold**, are *hard* keywords and may not be used as identifiers.

Note that the primitive type names are not considered keywords.

Literals   Briefly, X10 v2.2 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:

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

- `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; *e.g.*, `123`, `-321` are decimal `Int`s, `0123` and `-0321` are octal `Int`s, and `0x123`, `-0X321`, `0xBED`, and `0XEBEC` are hexadecimal `Int`s.

- `Long` literals have the form si`l` or si`L`. *E.g.*, `1234567890L` and `0xBABEL` are `Long` literals.

- `UInt` literals have the form i`u` or i`U`. *E.g.*, `123u`, `0123u`, and `0xBEAU` are `UInt` literals.

- `ULong` literals have the form i`ul` or i`lu`, or capital versions of those. For example, `123ul`, `0124567012ul`, `0xFLU`, `0Xba1eful`, and `0xDecafC0ffeefUL` are `ULong` literals.

- `Short` literals have the form si`s` or si`S`. *E.g.*, `414S`, `0xACES` and `7001s` are short literals.

- `UShort` literals form i`us` or i`su`, or capital versions of those. For example, `609US`, `107us`, and `0xBeaus` 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 `0xBABY` are byte literals.

- `UByte` literals have the form i`uy` or i`yu`, or capitalized versions of those. For example, `9uy` and `0xBUY` are `UByte` literals.

- `Float` literals have the form $sf\texttt{f}$ or $sf\texttt{F}$. 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, $sf\texttt{D}$, and $sf\texttt{d}$. *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;
  - `'\"'`, 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!"`, `""`.

Separators   X10 has the following separators and delimiters:

  `( )   { }   [ ]   ;   ,   .`

---

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

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

3.1   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`
	$\mid$	$Adv^?$ `sad`
Adv	::=	`very`
	$\mid$	*Adv Adv*

Terms in *italics* are called **non-terminals**. They represent kinds of phrases; for example, *ForStatement* (20.59) 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 $A ::= X_1 X_2 ... X_n$, 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 $\mid$.

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:

| $A^?$ | ::= | |
| | $\mid$ | *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 three kinds of runtime entities, *objects*, *structs*, and *functions*. Objects are instances of *classes* (§8). They may contain zero or more mutable fields, and a reference to the list of methods defined on them.

An object is represented by some (contiguous) memory chunk on the heap. Entities (such as variables and fields) contain a *reference* to this chunk. That is, objects are represented through an extra level of indirection. A consequence of this flexibility is that an entity containing a reference to an object o needs only one word of memory to represent that reference, regardless of the number of fields in o. An assignment to this entity simply overwrites the reference with another reference (thus taking constant time). Another consequence is that every class type contains the value null corresponding to the invalid reference. null is often useful as a default value. Further, two objects may be compared for identity (==) in constant time by simply comparing references to the memory used to represent the objects. The default hash code for an object is based on the value of this reference. A downside of this flexibility is that the operations of accessing a field and invoking a method are more expensive than simply reading a register and invoking a static function.

Structs are instances of *struct types* (§9). A struct is represented without the extra level of indirection, with a memory chunk of size N words precisely big enough to store the value of every field of the struct (modulo alignment), plus whatever padding is needed. Thus structs cannot be shared. Entities (such as variables and fields) refering to the struct must allocate N words to directly contain the chunk. An assignment to this entity must copy the N words representing the right hand side into the left hand side. Since there are no references to structs,

`null` is not a legal value for a struct type. Comparison for identity (==) involves examining N words. Additionally, structs do not have any mutable fields, hence they can be freely copied. The payoff for these restrictions lies in that fields can be stored in registers or local variables, and and method invocation is implemented by invoking a static function.

Functions, called closures or lambda-expressions 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:Int)=>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.

These runtime entities are classified by *types*. Types are used in variable declarations, coercions and explicit conversions, object creation, array creation, static state and method accessors, and `instanceof` and `as` expressions.

The basic relationship between values and types is the *is an element of* relation. We also often say "e has type T" to mean "e is an element of type T". For example, 1 has type `Int` (the type of all integers representible in 32 bits). It also has type `Any` (since all entitites have type `Any`), type `Int{self != 0}` (the type of nonzero integers), type `Int{self == 1}` (the type of integers which are equal to 1, which contains only one element), and many others.

The basic relationship between types is *subtyping*: `T <: U` holds if every instance of `T` is also an instance of `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, and the relationship between a class and another interface 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 `Int{self == 1}` are all elements of `Int{self != 0}` as well, because `self == 1` logically implies `self != 0`; so `Int{self  == 1} <: Int{self !=0}` by strengthening. X10 uses both notions of subtyping. See §4.8 for the full definition of subtyping in X10.

4.0.1   Type System

The types in X10 are as follows.

These are the *elementary* types. Other syntactic forms for types exist, but they are simply abbreviations for types in the following system. For example, `Array[Int](1)` is the type of one-dimensional integer-valued arrays; it is an abbreviation for `Array[Int]{rank==1}`.

Type	::=	*FunctionType*	*(20.14)*
	|	*ConstrainedType*	
FunctionType	::=	*TypeParams$^?$* (*FormalParamList$^?$*) *WhereClause$^?$*	*(20.15)*
		Offers$^?$ => *Type*	
ConstrainedType	::=	*NamedType*	*(20.18)*
	|	*AnnotatedType*	
	|	(*Type*)	

Types may be given by name. For example, `Int` is the type of 32-bit integers. Given a class declaration

```
class Triple { /* ... */ }
```

the identifier `Triple` may be used as a type.

The type *TypeName* [*Types$^?$*] is an instance of a *generic* (or *parameterized*) type. For example, `Array[Int]` is the type of arrays of integers. `HashMap[String,Int]` is the type of hash maps from strings to integers.

The type *Type* { *Constraint* } refers to a constrained type. *Constraint* is a Boolean expression – written in a *very* limited subset of X10 – describing the acceptable values of the constrained type. For example, `var n : Int{self != 0};` guarantees that `n` is always a non-zero integer. Similarly, `var x : Triple{x != null};` defines a `Triple`-valued variable `x` whose value is never null.

The qualified type *Type* . *Type* refers to an instance of a *nested* type; that is, a class or struct defined inside of another class or struct, and holding an implicit reference to the outer. For example, given the type declaration

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

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

Type variables, *TypeVar*, refer to types that are parameters. For example, the following class defines a cell in a linked list.

```
class LinkedList[X] {
  val head : X;
  val tail : LinkedList[X];
  def this(head:X, tail:LinkedList[X]) {
      this.head = head; this.tail = tail;
  }
}
```

It doesn't matter what type the cell is, but it has to have *some* type. `LinkedList[Int]` is a linked list of integers; `LinkedList[LinkedList[String]]` a list of lists of strings. Note that `LinkedList` is *not* a type – it is missing a type parameter.

The function type (*Formals*[?]) => *Type* refers to functions taking the listed formal parameters and returning a result of *Type*. In X10 v2.2, function types may not be generic. The closely-related void function type (*Formals*[?]) => `void` takes the listed parameters and returns no value. For example, `(x:Int) => Int{self != x}` is the type of integer-valued functions which have no fixed points. An example of such a function is `(x:Int) => x+1`.

4.1   Classes, Structs, and interfaces

4.1.1   Class types

A *class declaration* (§8) declares a *class type*, giving its name, behavior, data, and relationships to other classes and interfaces.

Example:

The `Position` *class below could describe the position of a slider control*

```
class Position {
  private var x : Int = 0;
  public def move(dx:Int) { x += dx; }
  public def pos() : Int = x;
}
```

Class instances, also called objects, are created by constructor calls: `new Position()`.
Class instances have fields and methods, type members, and value properties
bound at construction time. In addition, classes have static members: static `val`
fields, methods, type definitions, and member classes and member interfaces.

Classes may be *generic*, *i.e.*, defined with one or more type parameters (§4.2).

```
class Cell[T] {
  var contents : T;
  public def this(t:T) { contents = t;  }
  public def putIn(t:T) { contents = t; }
  public def get() = contents;
  }
```

X10 does not permit mutable static state. A fundamental principle of the X10
model of computation is that all mutable state be local to some place (§13), and,
as static variables are globally available, they cannot be mutable. When mutable
global state is necessary, programmers should use singleton classes, putting the
state in an object and using place-shifting commands (§13.3) and atomicity (§14.6)
as necessary to mutate it safely.

Classes are structured in a single-inheritance hierarchy. All classes extend the
class `x10.lang.Object`, directly or indirectly. Each class other than `Object`
extends a single parent class. `Object` provides no behaviors of its own, beyond
those required by `Any`.

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` but not `null` can be defined as
`C{self != null}`.

4.1.2 Struct Types

A *struct declaration* §9 introduces a *struct type* containing all instances of the
struct. The `Coords` struct below gives an immutable position in 3-space:

```
struct Position {
  public val x:Double, y:Double, z:Double;
  def this(x:Double, y:Double, z:Double) {
     this.x = x; this.y = y; this.z = z;
  }
}
```

Structs have many capabilities of classes: they can have methods, implement interfaces, and be generic. However, they have certain restrictions; for example, they cannot contain mutable (`val`) fields, or inherit from superclasses. There is no `null` value for structs. Due to these restrictions, structs may be implemented more efficiently than objects.

4.1.3   Interface types

An *interface declaration* (§7) defines an *interface type*, specifying a set of methods and properties which must be provided by any class declared to implement the interface.

Interfaces can also have static members: static fields, type definitions, and member classes, structs and interfaces. However, interfaces cannot specify that implementing classes must provide static members or constructors.

Example: *In the following interface,* `PI` *is a static field,* `Vec` *a static type definition,* `Pair` *a static member class. It can't insist that implementations provide a static method like* `meth`*, or a nullary constructor.*

```
interface Stat {
  static val PI = 3.14159;
  static type R = Double;
  static class Pair(x:R, y:R) {}
  // ERROR: static def meth():Int;
  // ERROR: static def this();
}
class Example {
  static def example() {
     val p : Stat.Pair = new Stat.Pair(Stat.PI, Stat.PI);
  }
}
```

An interface may extend multiple interfaces.

```
interface Named {
  def name():String;
}
interface Mobile {
  def move(howFar:Int):void;
```

```
   }
   interface Person extends Named, Mobile {}
   interface NamedPoint extends Named, Mobile {}
```

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, `KimThePoint` implements `Person`, and hence `Named` and `Mobile`. It would be a static error if `KimThePoint` had no `name` method, unless `KimThePoint` were also declared `abstract`.

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

4.1.4 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 an array is a property of the array, and an X10 program can specify that two arrays 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: Int, y: Int) {
     def this(x: Int, y: Int) :
       Coords{self.x==x, self.y==y} = {
       property(x, y);
     }

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

Properties, unlike other public `val` fields, can be used at compile time in con-
straints. This allows us to specify subtypes based on properties, by appending a
boolean expression to the type. For example, the type `Coords{x==0}` is the set
of all points whose `x` property is `0`. Details of this substantial topic are found in
§4.4.

4.2   Type Parameters and Generic Types

A class, interface, method, or type definition may have type parameters. Type pa-
rameters can be used as types, and will be bound to types on instantiation. For ex-
ample, a generic stack class may be defined as `Stack[T]{...}`. Stacks can hold
values of any type; *e.g.*, `Stack[Int]` is a stack of integers, and `Stack[Point
{self!=null}]` is a stack of non-null `Points`. 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 (§4.3, §8.4.2,§10.2).

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 [...]. For example, `Stack` is a generic class,
`Stack[Int]` is a type, and can be used as one: `var stack : Stack[Int];`

A `Cell[T]` is a generic object, capable of holding a value of type T. For example,
a `Cell[Int]` can hold an `Int`, and a `Cell[Cell[Int{self!=0}]]` can hold a
`Cell` which in turn can only hold non-zero numbers.

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

`Cell[Int]` is the type of `Int`-holding cells. The `get` method on a `Cell[Int]`
returns an `Int`; the `set` method takes an `Int` as argument. Note that `Cell` alone
is not a legal type because the parameter is not bound.

A class (whether generic or not) may have generic methods. Below, `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[Int](z)`). In certain
cases (*e.g.*, `first(z)`) type parameters may be omitted and are inferred by the
compiler (§4.11).

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

Limitation: X10 v2.2's C++ back end requires generic methods to be static or
final; the Java back end can accomodate generic instance methods as well.

Unlike other kinds of variables, type parameters may *not* be shadowed. If name `X`
is in scope as a type, `X` may not be rebound as a type variable. For example, neither
`class B` nor `class C[B]` are allowed in the following code, because they both
shadow the type variable `B`.

```
class A[B] {
  // ILLEGAL: class B{}
  // ILLEGAL: class C[B]{}
}
```

4.2.1   Use of Generics

An unconstrained type variable `X` can be instantiated any type. Within a generic
struct or class, all the operations of `Any` are available on a variable of type un-
constrained `X`. Additionally, variables of type `X` may be used with `==, !=`, in
`instanceof`, and casts.

If a type variable is constrained, the operations implied by its constraint are avail-
able as well.

Example: *The interface* `Named` *describes entities which know their own name.
The class* `NameMap[T]` *is a specialized map which stores and retrieves* `Named`
entities by name. The call `t.name()` *in* `put()` *is only valid because the constraint*
`{T <: Named}` *implies that* `T` *is a subtype of* `Named`, *and hence provides all the
operations of* `Named`.

```
interface Named { def name():String; }
class NameMap[T]{T <: Named} {
   val m = new HashMap[String, T]();
   def put(t:T) { m.put(t.name(), t); }
   def get(s:String):T = m.getOrThrow(s);
}
```

4.3   Type definitions

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

TypeDefDecl	::=	*Mods$^?$* type *Id* *TypeParams$^?$* *FormalParams$^?$*	*(20.4)*
		WhereClause$^?$ = Type ;	
TypeParams	::=	[*TypeParamList*]	*(20.25)*
FormalParams	::=	(*FormalParamList$^?$*)	*(20.26)*
WhereClause	::=	*DepParams*	*(20.29)*

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.

Thus 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.

Type definitions must appear as static members 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 class type, a `new`

expression may be used to create an instance of the class using the type definition's name. Given the following type definition:

```
type A = C[T₁, ..., Tₖ]{c};
```

where $C[T_1, \ldots, T_k]$ 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`.

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

4.3.1 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 <: Object}  = 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[Int](1)`s is `Array[Int]{self.rank==1 && self.size == n}` but it is often convenient to give a shorter name:

```
type Vec(n:Int) = Array[Int]{self.rank==1 && self.size == n};
var example : Vec(78);
```

The following examples are legal type definitions, given `import x10.util.*`:

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

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

4.4   Constrained types

Basic types, like Int 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 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 is a boolean expression `e` attached to a basic type `T`, written `T{e}`. (Only a limited selection of boolean expressions is available.) The values of type `T{e}` are the values of `T` for which `e` is true. For example:

- `String{self != null}` is the type of non-null strings. `self` is a special variable available only in constraints; it refers to the datum being constrained, and its type is the type to which the constraint is attached.

- If `Matrix` has 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} ;
    ```

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

`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 variable declaration, the variable itself is in scope in its type. For example, `val nz: Int{nz != 0} = 1;` declares a non-zero variable `nz`.

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}`.

4.4.1   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 type – in particular, the harder it is to compute `A{c} <: B{d}`. 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. Expressions computable at compile time, like `3*4+5`;

3. Accessible and immutable variables and parameters;

4. Accessible and immutable fields of objects;

5. Properties of the type being constrained;

6. `this`, if the constraint is in a place where `this` is defined;

7. `here`, if the constraint is in a place where `here` is defined;

8. `self`;

9. Calls to property methods, where the receiver and arguments must be value expressions acceptable in constraints;

10. `T haszero`, if `T` is any type expression.

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, and **Boolean expressions in 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;`[1]

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`;

All variables appearing in a constraint expression must be visible wherever that expression can used. *E.g.*, properties and public fields of an object are always permitted, but private fields of an object can only constrain private members. (Consider a class `PriVio` with a private field `p` and a public method `m(x: Int{self != p})`, and a call `ob.m(10)` made outside of the class. Since `p` is only visible inside the class, there is no way to tell if `10` is of type `Int{self != p}` at the call site.)

Limitation: Certain spurious syntactic forms – such as `a+b`, `a*b`, `(c==a&&b)`, `a <b` – are accepted by the compiler but treated incorrectly.

Semantics of constraints

An assignment of values to variables is said to be a *solution* for a constraint `c` if under this assignment `c` evaluates to `true`. For instance, the assignment that maps the variables `a` and `b` to a value `t` is a solution for the constraint `a==b`. An assignment that maps `a` to `s` and `b` to a distinct value `t` is a solution for `a != b`.

An instance `o` of `C` is said to be of type `C{c}` (or *belong to* `C{c}`) if the constraint `c` evaluates to `true` in the current lexical environment augmented with the binding `self ↦ o`.

A constraint `c` is said to *entail* a constraint `d` if every solution for `c` is also a solution for `d`. For instance the constraint `x==y && y==z && z !=a` entails `x != a`.

The constraint solver considers the assignment `a` to `null` to satisfy any constraint of the form `a.f==t`. Thus, for instance, the assignment `var x:Tree{self.home==p}=null` does not produce an error, since `self==null` is considered a solution for `self.home==p`.

To ensure that type-checking is decidable, we require that property graphs be acyclic. The property graph, at an instant in an X10 execution, is the graph whose

---

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

nodes are all objects in existence at that instance, with an edge from x to y if x is an object with a property whose value is y. The rules for constructors guarantee this.

Constraints participate in the subtyping relationship in a natural way: `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 (§4.4.3)

4.4.2   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 value that satisfies `c` satisfies `d`.

Limitation: X10's Entailment Algorithm is Incomplete However, 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`.

Certain other constraint entailments are prohibitively expensive to calculate. The issues concern constraints that connect different levels of recursively-defined types, such as the following.

```
class Listlike(x:Int) {
  val kid : Listlike{self.x == this.x};
  def this(x:Int, kid:Listlike) {
     property(x);
     this.kid = kid as Listlike{self.x == this.x};}
}
```

There is nothing wrong with `Listlike` itself, or with most uses of it; however, a sufficiently complicated use of it could, in principle, cause X10's typechecker to fail. It is hard to give a plausible example of when X10's algorithm fails, as we have not yet observed such a failure in practice for a correct program.

The entailment algorithm of X10 2.0 imposes a certain limit on the number of times such types will be unwound. If this limit is exceeded, the compiler will print a warning, and type-checking will fail in a situation where it is semantically allowed. In this case, insert a dynamic cast at the point where type-checking failed.

Limitation: Support for comparisons of generic type variables is limited. This will be fixed in future releases.

4.4.3 Example of Constraints

Example of entailment and subtyping involving constraints.

- `Int{self == 3} <: Int{self != 14}`. The only value of `Int{self ==3}` is 3. All integers but 14 are members of `Int{self != 14}`, and in particular 3 is.

- Suppose we have classes `Child <: Person`, and `Person` has a long `ssn` property. If `rhys : Child{ssn == 123456789}`, then `rhys` is also a `Person` and still has `ssn==123456789`, so `rhys : Person{ssn==123456789}` as well. So, `Child{ssn == 123456789} <: Person{ssn == 123456789}`.

- Furthermore, since `123456789 != 555555555`, `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:[2]

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

---

[2]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.

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.

4.5   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`, `Float`, `Double`) all have default value 0.

- `Boolean` has default value `false`.

- `Char` has default value `'\0'`.

- Struct types other than those listed above have no default value.

- 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.

For example, `var x: Int{x != 4}` has default value 0, which is allowed be-
cause `0 != 4` satisfies the constraint on `x`. `var y : Int{y==4}` has no default
value, because `0` does not satisfy y==4. The fact that `Int{y==4}` has precisely
one value, *viz.* 4, doesn't matter; the only candidate for its default value, as for
any subtype of `Int`, 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: *A variant* `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,
is given below. 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;}
}
```

4.6   Function types

FunctionType	::=	*TypeParams*$^?$ (*FormalParamList*$^?$) *WhereClause*$^?$	*(20.15)*
		Offers$^?$ => *Type*	

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) and method selectors (§10.3) are of function type. Typical functions are the reciprocal function:

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

and a function which increments element `i` of an array `r`, or throws an exception if there is no such element, where, for the sake of example, we constrain the type of `i`:

```
val inc = (r:Array[Int](1), i: Int{i != r.size}) => {
  if (i < 0 || i >= r.size) throw new DoomExn();
  r(i)++;
};
```

In general, a function type needs to list the types T_i of all the formal parameters, and their distinct names x_i in case other types refer to them; a constraint `c` on the function as a whole; a return type `T`.

$$(x_1: T_1, \ldots, x_n: T_n)\{c\} => T$$

The names x_i of the formal parameters are not relevant. Types which differ only in the names of formals (following the usual rules for renaming of variables, as in

α-renaming in the λ calculus) are considered equal. *E.g.*, the two function types
`(a:Int, b:Array[String](1){b.size==a}) => Boolean` and
`(b:Int, a:Array[String](1){a.size==b}) => Boolean` are equivalent.

Limitation: This is not currently implemented properly; these two types are presently considered different.

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 method `def apply(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 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 $A_1, \ldots, A_n$ using the syntax `@A`$_1$ `...` `@A`$_n$ `T`.

4.8   Subtyping and type equivalence

Intuitively, type T_1 is a subtype of type T_2, written T_1 <: T_2, if every instance of T_1 is also an instance of T_2. For example, `Child` is a subtype of `Person` (assuming a suitably defined class hierarchy): every child is a person. Similarly, `Int{self != 0}` is a subtype of `Int` – 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. `Int{self != 0}` <: `Int` is always true.

Limitation: Subtyping of type variables does not currently work.

- **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 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:** $(x_1: T_1, \ldots, x_n: T_n)\{c\}$ => T is a subtype of $(x'_1: T'_1, \ldots, x'_n: T'_n)\{c'\}$ => T' if:

  1. Each T_i <: T'_i;

  2. c entails c';

  3. 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.

4.9 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 to describe:

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

- Array construction, like [0, "non-zero"] and [0,1];

- Functions with multiple returns, like

  ```
  def f(a:Int) {
    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`. The best type of `0` is `Int{self==0}`, and the best type of `1` is `Int{self==1}`. Certainly E could be given the type `Int`, or even `Any`, and that would describe all possible results. However, we actually know more. `Int{self != 2}` is a better description of the type of E—certainly the result of E can never be `2`. `Int{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 `Int{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 `Int{c}` for some constraint `c`—perhaps simply picking `Int{true}`, *viz.*, `Int`.

- 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.

4.10 Fundamental types

Certain types are used in fundamental ways by X10.

4.10.1 The interface `Any`

It is quite convenient to have a type which all values are instances of; that is, a supertype of all types.[3] 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():Int;
}
```

`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.

---

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

4.10.2   The class `Object`

The class `x10.lang.Object` is the supertype of all classes. A variable of this type can hold a reference to any object. `Object` implements `Any`.

4.11   Type inference

X10 v2.2 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.

4.11.1   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: Int{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.

4.11.2   Return types

The return type of a method can be omitted if the method has a body (*i.e.*, is not `abstract` or `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 X10 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`, such as `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:Int) {
  def this() {property(0);}
  def this(xx: Int) { property(xx); }
}
```

A method or closure that has expression-free `return` statements (`return;` rather than `return e;`) is said to return `void`. `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. A method returning `void` can be specified by `def m():void`:

```
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, or `void`". Despite this informal usage, `void` is not a type. For example, given

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

The call `eval[void](f)` does *not* typecheck; `void` is not a type and thus cannot be used as a type argument. 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 most cases, functions which have no sensible return value can be provided with a dummy return value.

4.11.3 Inferring Type Arguments

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

(Exception: it is an error if the method call provides no information about a type parameter that must be inferred. For example, given the method definition `def m[T](){...}`, an invocation `m()` is considered a static error. The compiler has no idea what `T` the programmer intends.)

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[Int](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());
```

Sketch of X10 Type Inference for Method Calls

When the X10 compiler sees a method call `a.m(b`$_1$`, ...,b`$_n$`)`, and attempts to infer type parameters to see if it could be use of a method `def m[X`$_1$`, ..., X`$_t$`](y`$_1$`: S`$_1$`, ..., y`$_n$`:S`$_n$`)`, it reasons as follows.

Suppose that `b`$_i$ has type `T`$_i$. Then, X10 is seeking a set of type B bindings `X`$_j$ = `U`$_j$, for $1 \leq j \leq t$, such that `T`$_i$ `<:` `S`$_i^*$ for $1 \leq i \leq n$, where `S`* is `S` with each type variable `X`$_j$ replaced by the corresponding `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 $\{T_i \; \texttt{<:} \; S_i | 1 \leq i \leq n\}$. Let B_{n+1} be B_n with one element $F \; \texttt{<:} \; G$ or $F \; \texttt{==} \; G$ removed, and $C(F \; \texttt{<:} \; G)$ or $C(F \; \texttt{==} \; G)$ (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 $\texttt{W}$, but, if the inference is to work, all the choices of $\texttt{W}$ must be equal types anyways.

- 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 $\texttt{a.m[U}_1\texttt{,} \; \ldots\texttt{,} \; \texttt{U}_t\texttt{](b}_1\texttt{,} \; \ldots\texttt{,b}_n\texttt{)}$ is not a call of the original method $\texttt{def m[X}_1\texttt{,} \; \ldots\texttt{,} \; \texttt{X}_t\texttt{](y}_1\texttt{:} \; \texttt{S}_1\texttt{,} \; \ldots\texttt{,} \; \texttt{y}_n\texttt{:S}_n\texttt{)}$ — then type inference also fails.

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

- If F has the form $F'\texttt{\{c\}}$, then $C(r)$ 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. However, in important cases, the replacement is valid, in the sense that the solutions of B_{k+1} are precisely the solutions of B_k. Specifically, if the equation had the form $\texttt{Z\{c\}} \; \texttt{==} \; \texttt{A}$, it could be solved by $\texttt{Z==A}$ or by $\texttt{Z} \; \texttt{=} \; \texttt{A\{c\}}$. By dropping constraints in this rule, we choose the former solution.

- Similarly, we drop constraints on G as well.

- If F has the form $\mathtt{K[F_1, \ \ldots, \ F_k]}$ and G has the form $\mathtt{K[G_1, \ \ldots, \ G_k]}$, then $C(r)$ has one type relation comparing each parameter of F with the corresponding one of G.

  If r is a type equality $F \mathrel{==} G$, then $C(r) = \{F_l \mathrel{==} G_l | 1 \leq l \leq k$.

  If r is a type comparison, and the lth type parameter of $\mathtt{K}$ is invariant, then $C(r)$ has $F_l \mathrel{==} G_l$ (resp. $F_l \mathrel{<:} G_l$ or $G_l \mathrel{<:} F_l$).

  For example, the constraint $\mathtt{List[X] \ == \ List[Y]}$ produces the constraint $\mathtt{X==Y}$.

- Other cases are fairly routine. *E.g.*, if F is a $\mathtt{type}$-defined abbreviation, it is expanded.

Example:

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

4.12   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 $\mathtt{interface \ Loop \ extends \ Loop}$, and indirect self-references such as $\mathtt{interface \ A \ extends \ B.C}$ where $\mathtt{interface \ B \ extends \ A}$.

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. *e.g.* 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.

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-to-whatever 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]

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

Both variables have type Int *(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 : Int = 0;              // Full 'val' syntax
b : Int = 0;                  // 'val' implied
```

---

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

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

5.1   Immutable variables

LocalVariableDeclStatement	::=	*LocalVariableDecl* ;	*(20.155)*
LocalVariableDecl	::=	*Mods*? *VarKeyword VariableDeclarators*	*(20.156)*
	|	*Mods*? *VariableDeclaratorsWithType*	
	|	*Mods*? *VarKeyword FormalDeclarators*	

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 (20.156) using the *VariableDeclarators* or *VariableDeclaratorsWithType* alternatives.

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

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

In two special cases, the declaration and assignment are separate. One case is how constructors give values to `val` fields of objects. In this case, production (20.156) is taken, with the *FormalDeclarators* option, such as `var n:Int;`.

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 : Int; // not initialized here
```

```
    def this() { n = 1; }
    def this(dummy:Boolean) { n = 2;}
  }
```

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

5.2   Initial values of variables

Every assignment, binding, or initialization to a variable of type T{c} must be an instance of type T satisfying the constraint {c}. Variables must be given a value before they are used. This may be done by initialization – giving a variable a value as part of its declaration.

e.g. These variables are all initialized:

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

Or, a variable may 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.

e.g. 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 : Int;
      // ERROR: println(muta);
      muta = 4;
      val use2A = muta * 10;
```

```
val immu : Int;
// 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 vars to the default values of their types (§4.5). 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, §16.1. A point is a sequence of zero or more Int-valued coordinates. It is often useful to get at the coordinates directly, in variables.

VariableDeclarator	::=	*Id HasResultType$^?$ = VariableInitializer*	*(20.153)*
	|	[*IdList*] *HasResultType$^?$ = VariableInitializer*	
	|	*Id* [*IdList*] *HasResultType$^?$ = VariableInitializer*	

The syntax val [a$_1$, ..., a$_n$] = e; declares n Int variables, bound to the first n components of the Point value of e; it is an error if e is not a Point with at least n components. The syntax val p[a$_1$, ..., a$_n$] = e; is similar, but also declares the variable p as Point.

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);
val p[j,k] = Point.make(22,33);
val q[l,m] = [44,55]; // coerces an array to a point.
```

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(Array[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:Int) { i += 1; }
```

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:Int = 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 (§4.11).

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:Int){
  def this(pp:Int):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 <: Int = new Tub(3);
```

5.6 Fields

FieldDeclarators	::=	*FieldDeclarator*	*(20.122)*
	|	*FieldDeclarators* , *FieldDeclarator*	
FieldDecl	::=	*Mods$^?$ FieldKeyword FieldDeclarators* ;	*(20.38)*
	|	*Mods$^?$ FieldDeclarators* ;	
FieldDeclarator	::=	*Id HasResultType*	*(20.152)*
	|	*Id HasResultType$^?$ = VariableInitializer*	
HasResultType	::=	: *Type*	*(20.127)*
	|	<: *Type*	
FieldKeyword	::=	`val`	*(20.36)*
	|	`var`	
Mod	::=	`abstract`	*(20.2)*
	|	*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`. 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 (§4.11). `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 manip-
ulating 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 inter-
faces 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 §6.3.

6.1.2 Hiding

Shadowing is ubiquituous in X10. Another, and considerably rarer, way that one definition of a given simpl ename 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`, then, for `Subs`, references to the `x` field get `Sub`'s `x` rather than `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` is 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 (§8.9).

Example: *In the following example, there are three static methods with qual-ified 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* `Int` *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:Int) {
    if (i > 10) {
      return i() + 1;
    }
    return i;
  }
  static def example() {
    assert m() == 1;
    assert m(true) == 2;
    assert m(20) == 4;
  }
}
```

X10 allows certain modifiers to limit the use of named entities outside of the body that they were defined in: the `public`, `protected`, `private` and uninflected package-specific scopes, as used in Java [5, §6,§7] and various other languages.

6.2 Packages

A package is a named collection of top-level type declarations, *viz.*, class, inter-face, 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 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`.

6.2.1   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 pack-
age 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 con-
vention that package names are lower-case and package members are capitalized
prevents such collisions.

6.3   `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.

6.3.1 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 (§6.3.2) in f.

6.3.2 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. Names automatically imported never shadow any other names.

6.3.3 Implicit Imports

The packages `x10.lang` and `x10.array` are automatically imported in all files without need for further specification.

6.4   Conventions on Type Names

TypeName	::=	*Id*	*(20.98)*
	|	*TypeName* . *Id*	
PackageName	::=	*Id*	*(20.102)*
	|	*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 vals`, 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 = Int;
  property text():Int;
  property prio():Int;
}
class MessageButton(text:String, prio:Int)
  implements Pushable{self.prio()==Pushable.MAX_PRIO} {
  public def push() {
    x10.io.Console.OUT.println(text + " pushed");
  }
  public property text() = text;
  public property prio() = 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.

A concrete type—a class or struct—can *implement* an interface, typically by having all the methods and property methods that the interface requires.

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

NormalInterfaceDecl	::=	*Mods$^?$* interface *Id TypeParamsWithVariance$^?$*	*(20.11)*
		WhereClause$^?$ ExtendsInterfaces$^?$ InterfaceBody	
TypeParamsWithVariance	:=	[*TypeParamWithVarianceList*]	*(20.24)*
WhereClause	::=	*DepParams*	*(20.29)*
ExtendsInterfaces	::=	extends *Type*	*(20.140)*
	|	*ExtendsInterfaces* , *Type*	
InterfaceBody	::=	{ *InterfaceMemberDecls$^?$* }	*(20.141)*
InterfaceMemberDecl	::=	*MethodDecl*	*(20.143)*
	|	*PropertyMethodDecl*	
	|	*FieldDecl*	
	|	*ClassDecl*	
	|	*InterfaceDecl*	
	|	*TypeDefDecl*	
	|	;	

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,

- C's property methods include all the property methods of I,

- Each method m defined by I is also a method of C – with the public modifier added. These methods may be abstract if C is abstract.

If C implements I, then the class invariant (§8.8) for C, $inv(C)$, implies the class invariant for I, $inv(I)$. 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   Property Methods

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

7.4   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;
}
```

7.4.1   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 local class (§8.12)* B *is defined, with an inner interface* I. *The field* V *of* I *uses a variable* n *which is global to* B. *In this case it is a truly baroque way to bind a* val, *but other uses are nontrivial.*

```
class A {
  static def example(n: Int) {
    class B {
      interface I { val V = n*n; }
    }
   return B.I.V + 1;
  }
}
```

7.5 Generic Interfaces

Interfaces, like classes and structs, can have type parameters. The discussion of generics in §4.2 applies to interfaces, without modification.

Example:

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

7.6   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.

7.7   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, §8.4.4

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, better in some circumstances but not in all. `x10.lang.Object` is the most general class; all other classes inherit from it, directly or indirectly.

Classes are structured in a single-inheritance code hierarchy. They 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.

X10 objects do not have locks associated with them. Programmers should use atomic blocks (§14.6) 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:

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

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

- Casting (§11.22) and instance testing with `instanceof` (§11.23) 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*), a single 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 (§4.1).

NormalClassDecl	::=	*Mods*$^?$ class *Id TypeParamsWithVariance*$^?$	*(20.32)*
		Properties$^?$ *WhereClause*$^?$ *Super*$^?$ *Interfaces*$^?$	
		ClassBody	
TypeParamsWithVariance	:=	[*TypeParamWithVarianceList*]	*(20.24)*
TypeParamWithVarianceList	:=	*TypeParamWithVariance*	*(20.87)*
	|	*TypeParamWithVarianceList* , *TypeParamWithVari-ance*	
Properties	::=	(*PropertyList*)	*(20.5)*
PropertyList	::=	*Property*	*(20.6)*
	|	*PropertyList* , *Property*	
Property	::=	*Annotations*$^?$ *Id ResultType*	*(20.7)*
WhereClause	::=	*DepParams*	*(20.29)*
Super	::=	extends *ClassType*	*(20.35)*
Interfaces	::=	implements *InterfaceTypeList*	*(20.115)*
InterfaceTypeList	::=	*Type*	*(20.116)*
	|	*InterfaceTypeList* , *Type*	
ClassBody	::=	{ *ClassBodyDecls*$^?$ }	*(20.117)*
ClassBodyDecls	::=	*ClassBodyDecl*	*(20.118)*
	|	*ClassBodyDecls ClassBodyDecl*	
ClassMemberDecl	::=	*FieldDecl*	*(20.120)*
	|	*MethodDecl*	
	|	*PropertyMethodDecl*	
	|	*TypeDefDecl*	
	|	*ClassDecl*	
	|	*InterfaceDecl*	
	|	;	

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).

Class may have *static fields*, which store data pertinent to the entire class of ob-
jects. See §8.6 for more information.

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 (§8.9.3).

8.2.1   Field Initialization

Fields may be given values via *field initialization expressions*: `val f1 = E;` and `var f2 : Int = F;`. Other fields of `this` may be referenced, but only those that *precede* the field being initialized. For 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() : Int = super.f; // 1
}

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; this) { // causes copying of 'this'
         assert(this.copied.equals("copied"));
```

```
            assert(this.transy == null);
        }
      }
    }
```

8.3   Properties

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

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

```
class Proper(t:Int) {
  def this(t:Int) {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;
```

It also defines a nullary getter method,

```
public final def x()=x;
```

Properties are initialized by the invocation of a special `property` statement:

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

The number and types of arguments to the `property` statement must match the number and types of the properties in the class declaration. Every constructor of a class with properties must invoke `property(...)` precisely once; it is a static error if X10 cannot prove that this holds. The requirement to use the `property` statement means that all properties must be given values at the same time.

---

[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.

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`.

8.3.1 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.

8.3.2 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*$^?$	*(20.3)*
	|	*MethMods* `property`	
	|	*MethMods Mod*	
MethodDecl	::=	*MethMods* `def` *Id TypeParams*$^?$ *FormalParams*	*(20.8)*
		WhereClause$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ (*Formal-Param*) *BinOp* (*FormalParam*) *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ *PrefixOp* (*FormalParam*) *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ `this` *BinOp* (*FormalParam*) *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ (*Formal-Param*) *BinOp* `this` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ *PrefixOp* `this` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` `this` *TypeParams*$^?$ *FormalParams* *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` `this` *TypeParams*$^?$ *FormalParams* = (*FormalParam*) *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ (*FormalParam*) `as` *Type WhereClause*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ (*FormalParam*) `as` ? *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
	|	*MethMods* `operator` *TypeParams*$^?$ (*FormalParam*) *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *Method-Body*	
TypeParams	::=	[*TypeParamList*]	*(20.25)*
FormalParams	::=	(*FormalParamList*$^?$)	*(20.26)*
FormalParamList	::=	*FormalParam*	*(20.128)*
	|	*FormalParamList* , *FormalParam*	
HasResultType	::=	: *Type*	*(20.127)*
	|	<: *Type*	
MethodBody	::=	= *LastExp* ;	*(20.135)*
	|	= *Annotations*$^?$ { *BlockStatements*$^?$ *LastExp* }	
	|	= *Annotations*$^?$ *Block*	
	|	*Annotations*$^?$ *Block*	
	|	;	

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   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.

8.4.2   Method Guards

Often, a method will only make sense to invoke under certain statically-determinable conditions. For example, `example(x)` is only well-defined when `x != null`, as `null.toString()` throws a null pointer exception:

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

(We could have used a constrained type `Object{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 that the guard is satisfied. (As usual with static constraint checking, there is no runtime cost. Indeed, this code can be more efficient than usual, as it is statically provable that `x != null`.) This may require a cast:

```
def exam(e:Example, x:Object) {
  if (x != null)
      e.example(x as Object{x != null});
  // WRONG: 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.

8.4.3 Property methods

Property methods are methods that can be evaluated in constraints. For 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`. These can be used in constraints, as illustrated in the `example()` method.

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

```
}
```

A method declared with the modifier `property` may be used in constraints. A property method declared in a class must have a body and must not be `void`. The body of the method must consist of only a single `return` statement or a single expression. It is a static error if the expression cannot be represented in the constraint system. Property methods may be `abstract` in `abstract` classes, but are implicitly `final` in non-`abstract` classes.

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

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 formal parameters and the `def` keyword. That is, the following are equivalent:

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

and

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

Similarly, nullary property methods can be inspected in constraints without `()`. 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]

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

---

[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 large.

8.4.4 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 types. *E.g.*, the following is legal:

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

X10 v2.2 does not permit overloading based on constraints. That is, the following is *not* legal, although either method definition individually is legal:

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

The definition of a method declaration m_1 "having the same signature as" a method declaration m_2 involves identity of types.

(V21: The following few paragraphs need to be stared at carefully. I think they are contradictory and/or wrong.)

The *constraint erasure* of a type T is defined as follows. The constraint erasure of (a) a class, interface or struct type T is T; (b) a type $T\{c\}$ is the constraint erasure of T; (b) a type $T[S_1, \ldots, S_n]$ is $T'[S_1', \ldots, S_n']$ where each primed type is the erasure of the corresponding unprimed type. Two methods are said to have *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 equivalent. It is a compile-time error for there to be two methods with the same name and equivalent signatures in a class (either defined in that class or in a superclass), unless the signatures

are identical and one of the methods is defined in a superclass (in which case the superclass's method is overridden by the subclass's).

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

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 M_1 in a class C overrides a method M_2 in a superclass D if M_1 and M_2 have the same signature with constraints erased. Methods are overriden on a signature-by-signature basis.

8.5   Constructors

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

$$
\begin{array}{rcl}
\textit{ClassInstCreationExp} & ::= & \text{new } \textit{TypeName TypeArguments}^? \text{ (} \textit{ArgumentList}^? \text{)} \quad \textit{(20.12)} \\
& & \textit{ClassBody}^? \\
& | & \text{new } \textit{TypeName } [\textit{ Type }] [\textit{ ArgumentList}^?] \\
& | & \textit{Primary . } \text{new } \textit{Id TypeArguments}^? \text{ (} \textit{ArgumentList}^? \\
& & \text{) } \textit{ClassBody}^? \\
& | & \textit{AmbiguousName . } \text{new } \textit{Id TypeArguments}^? \text{ (} \\
& & \textit{ArgumentList}^? \text{) } \textit{ClassBody}^?
\end{array}
$$

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 are 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 §8.8.3 for the details.

Example:

The following class provides two constructors. The unary constructor `def this(b : Int)` *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 : Int;
  def this(b : Int) { 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:Int){property(x);}` *This implicit constructor is valid, and so is the class.*

```
class C(x:Int) {
  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:Int) {
  val d: Int;
  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 {
  val a : Int;
  def this(a:Int) { 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 early in a constructor, precede it with a dummy statement, such as `if(false);`

8.6   Static initialization

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

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

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

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

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

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

If all these activities terminate normally, all static fields have values of their declared types, and the `finish` terminates normally. If any activity throws an exception, the `finish` throws an exception. Since no user code is executing which can catch exceptions thrown by the finish, such exceptions are printed on the console, and computation aborts.

If the activities deadlock, the implementation deadlocks.

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

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

8.7   User-Defined Operators

MethodDecl	::=	*MethMods* `def` *Id TypeParams$^?$ FormalParams* (20.8)
		WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody
	|	*MethMods* `operator` *TypeParams$^?$* (*FormalParam*) *BinOp* (*FormalParam*) *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$ PrefixOp* (*FormalParam*) *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$* `this` *BinOp* (*FormalParam*) *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$* (*FormalParam*) *BinOp* `this` *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$ PrefixOp* `this` *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` `this` *TypeParams$^?$ FormalParams WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` `this` *TypeParams$^?$ FormalParams* = (*FormalParam*) *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$* (*FormalParam*) `as` *Type WhereClause$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$* (*FormalParam*) `as` ? *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*
	|	*MethMods* `operator` *TypeParams$^?$* (*FormalParam*) *WhereClause$^?$ HasResultType$^?$ Offers$^?$ MethodBody*

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

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

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

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

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

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

Most X10 operators can be given definitions.[3] (However, `&&` and `||` are only short-circuiting for `Boolean` expressions; user-defined versions of these operators have

---

[3]Indeed, even for the standard types, these operators are defined in the library. Not even as basic an operation as integer addition is built into the language. Conversely, if you define a full-featured numeric type, it will have most of the privileges that the standard ones enjoy. The missing

no special execution behavior.)

The user-definable operations are (in order of precedence):

```
 implicit type coercions
 postfix ()
 as T
 unary -, unary +, !, ~
 ..
 *       /       %
 +       -
 <<      >>      >>>     ->
 >       >=      <       <=      in      &
 ^
 |
 &&
 ||
```

Many operators may be defined either in `static` or instance forms. Those defined in instance form are dynamically dispatched, just like an instance method. Those defined in static form are statically dispatched, just like a static method.

Example:

```
static class Trace(n:Int){
  public static operator !(f:Trace) = new Trace(10 * f.n + 1);
  public operator -this = new Trace (10 * this.n + 2);
}
static class Brace extends Trace{
  def this(n:Int) { super(n); }
  public operator -this = new Brace (10 * this.n + 3);
  static def example() {
      val t = new Trace(1);
      assert (!t).n == 11;
      assert (-t).n == 12 && (-t instanceof Trace);
      val b = new Brace(1);
```

---

priveleges are (1) literals; (2) the `..` operator won't compute the `zeroBased` and `rail` properties as it does for `Int` ranges; (3) `*` won't track ranks, as it does for `Region`s; (4) `&&` and `||` won't short-circuit, as they do for `Boolean`s, (5) `a==b` will only coincide with `a.equals(b)` if coded that way; and (6) it is impossible to define an operation like `String.+` which converts both its left and right arguments from any type. For example, a `Polar` type might have many representations for the origin, as radius 0 and any angle; these will be `equals()`, but will not be `==`.

```
        assert (!b).n == 11;
        assert (-b).n == 13 && (-b instanceof Brace);
    }
}
```

Operators may be invoked by unambiguous syntax, loosely akin to a fully-qualified name. For example, !t above may be invoked as `Trace.operator !(t)`. This unambiguous syntax may be used even if there are several ! operators that could apply to t, rendering the convenient short form !t unavailable in some context.

8.7.1   Binary Operators

Binary operators, illustrated by +, may be defined statically in a container A as:

```
    static operator (b:B) + (c:C) = ...;
```

In this case it may be invoked as `A.operator +(b,c)`. Or, it may be defined as as an instance operator by one of the forms:

```
    operator this + (b:B) = ...;
    operator (b:B) + this = ...;
```

and be invoked as `a.operator +(b)` and as `a.operator ()+(b)` respectively.

Example:

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

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

      public operator this + (p:Poly) =  new Poly(
        new Array[Int](
          Math.max(this.coeff.size, p.coeff.size),
          (i:Int) => this.a(i) + p.a(i)
```

```
        ));
    // ...
```

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

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

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

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

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

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

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

8.7.2   Unary Operators

Unary operators, illustrated by !, may be defined statically in container A as

```
        static operator !(x:A) = ...;
```

or as instance operators by:

```
operator !this = ...;
```

A statically-defined unary operator ! may be invoked on a:A as A.operator
!(a). An instance operator may be invoked as a.operator !().

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

Example: *The operator to negate a polynomial is:*

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

8.7.3 Type Conversions

Explicit type conversions, e as A, can be defined as operators on class A. These
must be static operators, and are defined in the the form:

```
static operator (x:B) as ? {c} = ...
```

The ? indicates the containing type A. The guard clause {c} may be omitted.
They may be invoked by the unambiguous syntax:

```
A.operator as[A](b)
```

Example:

```
class Poly {
  public val coeff : Array[Int](1);
  public def this(coeff: Array[Int](1)) { this.coeff = coeff;}
  public static operator (a:Int) as ? = new Poly([a as Int]);
  public static def main(Array[String](1)):void {
     val three : Poly = 3 as Poly;
  }
}
```

The ? may be given a bound, such as as ? <: Caster, if desired.

8.7.4   Implicit Type Coercions

An implicit type conversion from U to T may be specified in either U or T. The syntax for it, in either case, is:

```
static operator (u:U) : T = e;
```

which may be invoked by the unambiguous syntax `T.operator[T](u)` or `U.operator[T](u)`.

(V21: I don't know what to do about constraints on T or U.)

(V21: We need spec checks on this)

Implicit coercions are used automatically by the compiler on method calls (§8.9) and assignments (§**??**).

For example, we can define an implicit coercion from Int to Poly, and avoid having to define the sum of an integer and a polynomial as many special cases. In the following example, we only define + on two polynomials (using a static operator, so that implicit coercions will be used – they would not be for an instance method operator). The calculation 1+x coerces 1 to a polynomial and uses polynomial addition to add it to x.

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

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

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

8.7.5   Assignment and Application Operators

X10 allows types to implement the subscripting / function application operator, and indexed assignment. The Array-like classes take advantage of both of these in `a(i) = a(i) + 1`.

a(b,c,d) is an operator call, to an operator defined with `public operator this(b:B, c:C, d:D)`. It may be overloaded. For example, an ordered dictionary structure could allow subscripting by numbers with `public operator this(i:Int)`, and by string-valued keys with `public operator this(s:String)`.

a(i,j)=b is an `operator` as well, with zero or more indices i,j. It may also be overloaded.

The update operations `a(i) += b` are defined to be the same as the corresponding `a(i) = a(i) + b`. This applies for all arities of arguments, and all types, and all binary operations. Of course to use this, both the application and assignment `operator`s must be defined.

Example:

The `Oddvec` class of somewhat peculiar vectors illustrates this. `a()` returns a string representation of the oddvec, which probably should be done by `toString()` instead. `a(i)` sensibly picks out one of the three coordinates of `a`. `a()=b` sets all the coordinates of `a` to `b`. `a(i)=b` assigns to one of the coordinates. `a(i,j)=b` assigns different values to `a(i)` and `a(j)`, purely for the sake of the example.

```
class Oddvec {
  var v : Array[Int](1) = new Array[Int](3, (Int)=>0);
  public operator this () =
      "(" + v(0) + "," + v(1) + "," + v(2) + ")";
  public operator this () = (newval: Int) {
    for(p in v) v(p) = newval;
  }
  public operator this(i:Int) = v(i);
  public operator this(i:Int, j:Int) = [v(i),v(j)];
  public operator this(i:Int) = (newval:Int) = {v(i) = newval;}
  public operator this(i:Int, j:Int) = (newval:Int) = {
      v(i) = newval; v(j) = newval+1;}
  public def example() {
    this(1) = 6;    assert this(1) == 6;
    this(1) += 7;   assert this(1) == 13;
  }
```

8.8   Class Guards and Invariants

Classes (and structs and interfaces) may specify a *class guard*, a constraint which
must hold on all values of the class. In the following example, a `Line` is defined
by two distinct `Pts`[4]

```
class Pt(x:Int, y:Int){}
class Line(a:Pt, b:Pt){a != b} {}
```

In most cases the class guard could be phrased as a type constraint on a property
of the class instead, if preferred. Arguably, a symmetric constraint like two points
being different is better expressed as a class guard, rather than asymmetrically as
a constraint on one type:

```
class Line(a:Pt, b:Pt{a != b}) {}
```

With every defined class, struct, or interface `T` we associate a *type invariant* $inv(T)$,
which describes the guarantees on the properties of values of type `T`.

Every value of `T` satisfies $inv(T)$ at all times. This is somewhat stronger than the
concept of type invariant in most languages (which only requires that the invariant
holds when no method calls are active). X10 invariants only concern properties,
which are immutable; thus, once established, they cannot be falsified.

The type invariant associated with `x10.lang.Any` is `true`.

The type invariant associated with any interface or struct `I` that extends interfaces
I_1, ..., I_k and defines properties x_1: P_1, ..., x_n: P_n and specifies a guard
`c` is given by:

inv(`I`$_1$) `&&` ... `&&` inv(`I`$_k$)
    `&& self.`x_1 `instanceof P`$_1$ `&&` ... `&&` `self.`x_n `instanceof P`$_n$
    `&& c`

Similarly the type invariant associated with any class `C` that implements interfaces
I_1, ..., I_k, extends class `D` and defines properties x_1: P_1, ..., x_n: P_n and
specifies a guard `c` is given by the same thing with the invariant of the superclass
`D` conjoined:

inv(`I`$_1$) `&&` ... `&&` inv(`I`$_k$)
    `&& self.`x_1 `instanceof P`$_1$ `&&` ... `&&` `self.`x_n `instanceof P`$_n$

---

[4]We use `Pt` to avoid any possible confusion with the built-in class `Point`.

```
&& c
&& inv(D)
```

Note that the type invariant associated with a class entails the type invariants of each interface that it implements (directly or indirectly), and the type invariant of each ancestor class. It is guaranteed that for any variable v of type T{c} (where T is an interface name or a class name) the only objects o that may be stored in v are such that o satisfies $inv(\texttt{T}[\texttt{o/this}]) \land \texttt{c}[\texttt{o/self}]$.

8.8.1 Invariants for `implements` and `extends` clauses

Consider a class definition

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

Each of the following static semantics rules must be satisfied:

(V21: reformat this)

The type invariant $inv(\texttt{C})$ of C must entail $\texttt{c}_i\,[\texttt{this/self}]$ for each i in $\{1, \ldots, k\}$

The return type c of each constructor in a class C must entail the invariant $inv(\texttt{C})$.

8.8.2 Invariants and constructor definitions

A constructor for a class C is guaranteed to return an object of the class on successful termination. This object must satisfy $inv(\texttt{C})$, the class invariant associated with C (§8.8). However, often the objects returned by a constructor may satisfy *stronger* properties than the class invariant. X10's dependent type system permits these extra properties to be asserted with the constructor in the form of a constrained type (the "return type" of the constructor):

CtorDecl	::=	*Mods?* def this *TypeParams?* *FormalParams*	*(20.34)*
		WhereClause? HasResultType? CtorBody	

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

Example: *Here is another example, constructed as a simplified version of*
`x10.array.Region`. *The* `mockUnion` *method has the type, though not the value,*
that a true `union` *method would have.*

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

The first constructor returns the empty region of rank `r`. *The second constructor*
takes a `Array[Int](1)` *of arbitrary length* n *and returns a* `MyRegion(n)` *(in-*
tended to represent the set of points in the rectangular parallelopiped between the
origin and the `diag`.*)*

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

Let C be a class with properties p_1: P_1, ..., p_n: P_n, and invariant c extending
the constrained type D{d} (where D is the name of a class).

For every constructor in C the compiler checks that the call to super invokes a
constructor for D whose return type is strong enough to entail d. Specifically, if
the call to super is of the form super(e_1, ..., e_k) and the static type of each
expression e_i is S_i, and the invocation is statically resolved to a constructor def
this(x_1: T_1, ..., x_k: T_k){c}: D{d_1} then it must be the case that

x_1: S_1, ..., x_i: S_i entails x_i: T_i (for $i \in \{1,\ldots,k\}$)
x_1: S_1, ..., x_k: S_k entails c
d_1[a/self], x_1: S_1, ..., x_k: S_k entails d[a/self]

where a is a constant that does not appear in x_1: $S_1 \wedge \ldots \wedge x_k$: S_k.

The compiler checks that every constructor for C ensures that the properties p_1, ..., p_n are initialized with values which satisfy $inv(T)$, and its own return type c' as follows. In each constructor, the compiler checks that the static types T_i of the expressions e_i assigned to p_i are such that the following is true:

    p_1: T_1, ..., p_n: T_n entails $inv(T) \wedge$ c'

(Note that for the assignment of e_i to p_i to be type-correct it must be the case that p_i: $T_i \wedge p_i$: P_i.)

The compiler must check that every invocation C(e_1, ..., e_n) to a constructor is type correct: each argument e_i must have a static type that is a subtype of the declared type T_i for the ith argument of the constructor, and the conjunction of static types of the argument must entail the constraint in the parameter list of the constructor.

8.8.3 Object Initialization

X10 does object initialization safely. It avoids certain bad things which trouble some other languages:

1. Use of a field before the field has been initialized.

2. A program reading two different values from a `val` field of a container.

3. `this` escaping from a constructor, which can cause problems as noted below.

It should be unsurprising that fields must not be used before they are initialized. At best, it is uncertain what value will be in them, as in x below. Worse, the value might not even be an allowable value; y, declared to be nonzero in the following example, might be zero before it is initialized.

```
// Not correct X10
class ThisIsWrong {
  val x : Int;
  val y : Int{y != 0};
  def this() {
    x10.io.Console.OUT.println("x=" + x + "; y=" + y);
    x = 1; y = 2;
```

```
        }
    }
```

One particularly insidious way to read uninitialized fields is to allow `this` to escape from a constructor. For example, the constructor could put `this` into a data structure before initializing it, and another activity could read it from the data structure and look at its fields:

```
    class Wrong {
      val shouldBe8 : Int;
      static Cell[Wrong] wrongCell = new Cell[Wrong]();
      static def doItWrong() {
          finish {
            async { new Wrong(); } // (A)
            assert( wrongCell().shouldBe8 == 8); // (B)
          }
      }
      def this() {
          wrongCell.set(this); // (C) - ILLEGAL
          this.shouldBe8 = 8; // (D)
      }
    }
```

In this example, the underconstructed `Wrong` object is leaked into a storage cell at line (C), and then initialized. The `doItWrong` method constructs a new `Wrong` object, and looks at the `Wrong` object in the storage cell to check on its `shouldBe8` field. One possible order of events is the following:

1. `doItWrong()` is called.

2. (A) is started. Space for a new `Wrong` object is allocated. Its `shouldBe8` field, not yet initialized, contains some garbage value.

3. (C) is executed, as part of the process of constructing a new `Wrong` object. The new, uninitialized object is stored in `wrongCell`.

4. Now, the initialization activity is paused, and execution of the main activity proceeds from (B).

5. The value in `wrongCell` is retrieved, and is `shouldBe8` field is read. This field contains garbage, and the assertion fails.

6. Now let the initialization activity proceed with (D), initializing `shouldBe8` — too late.

The `at` statement (§13.3) introduces the potential for escape as well. The following class prints an uninitialized value:

```
class Example {
  val a: Int;
  def this() {
    at(here.next()) {
      // Recall that 'this' is a copy of 'this' outside 'at'.
      Console.OUT.println("this.a = " + this.a);
    }
    this.a = 1;
  }
}
```

X10 must protect against such possibilities. The rules explaining how constructors can be written are somewhat intricate; they are designed to allow as much programming as possible without leading to potential problems. Ultimately, they simply are elaborations of the fundamental principles that uninitialized fields must never be read, and `this` must never be leaked.

8.8.4 Raw and Cooked Objects

An object is *raw* before its constructor ends, and *cooked* after its constructor ends. Note that, when an object is cooked, all its subobjects are cooked.

8.8.5 Constructors and NonEscaping Methods

In general, constructors must not be allowed to call methods with `this` as an argument or receiver. Such calls could leak references to `this`, either directly from a call to `cell.set(this)`, or indirectly because `toString` leaks `this`, and the concatenation '"Escaper = "+this' calls `toString`.[5]

---

[5]This is abominable behavior for `toString`, but it cannot be prevented – save by a scheme such as we present in this section.

```
class Escaper {
  static val Cell[Escaper] cell = new Cell[Escaper]();
  def toString() {
    cell.set(this);
    return "Evil!";
  }
  def this() {
    cell.set(this);
    x10.io.Console.OUT.println("Escaper = " + this);
  }
}
```

However, it is convenient to be able to call methods from constructors; *e.g.*, a class might have eleven constructors whose common behavior is best described by three methods. Under certain stringent conditions, it *is* safe to call a method: the method called must not leak references to `this`, and must not read `val`s or `var`s which might not have been assigned.

So, X10 performs a static dataflow analysis, sufficient to guarantee that method calls in constructors are safe. This analysis requires having access to or guarantees about all the code that could possibly be called. This can be accomplished in two ways:

1. Ensuring that only code from the class itself can be called, by forbidding overriding of methods called from the constructor: they can be marked `final` or `private`, or the whole class can be `final`.

2. Marking the methods called from the constructor by `@NonEscaping`.

Non-Escaping Methods

A method may be annotated with `@NonEscaping`. This imposes several restrictions on the method body, and on all methods overriding it. However, it is the only way that a method can be called from constructors. The `@NonEscaping` annotation makes explicit all the X10 compiler's needs for constructor-safety.

A method can, however, be safe to call from constructors without being marked `@NonEscaping`. We call such methods *implicitly non-escaping*. Implicitly non-escaping methods need to obey the same constraints on `this`, `super`, and variable

usage as @NonEscaping methods. An implicitly non-escaping method *could* be marked as @NonEscaping for some list of variables; the compiler, in effect, infers the annotation. In addition, implicitly non-escaping methods must be private or final or members of a final class; this corresponds to the hereditary nature of @NonEscaping (by forbidding inheritance of implicitly non-escaping methods).

We say that a method is *non-escaping* if it is either implicitly non-escaping, or annotated @NonEscaping.

The first requirement on non-escaping methods is that they do not allow this to escape. Inside of their bodies, this and super may only be used for field access and assignment, and as the receiver of non-escaping methods.

Finally, if a method m in class C is marked @NonEscaping, then every method which overrides m in any subclass of C must be annotated with precisely the same annotation, @NonEscaping, as well.

The following example uses most of the possible variations (leaving out final class). aplomb() explicitly forbids reading any field but a. boric() is called after a and b are set, but c is not. The @NonEscaping annotation on boric() is optional, but the compiler will print a warning if it is left out. cajoled() is only called after all fields are set, so it can read anything; its annotation, too, is not required. SeeAlso is able to override aplomb(), because aplomb() is @NonEscaping("a"); it cannot override the final method boric() or the private one cajoled(). Even for overriding aplomb(), it is crucial that SeeAlso.aplomb() be declared @NonEscaping("a"), just like C2.aplomb().

```
import x10.compiler.*;

final class C2 {
  protected val a:Int, b:Int, c:Int;
  protected var x:Int, y:Int, z:Int;
  def this() {
    a = 1;
    this.aplomb();
    b = 2;
    this.boric();
    c = 3;
    this.cajoled();
  }
  @NonEscaping def aplomb() {
```

```
    x = a;
    // this.boric(); // not allowed; boric reads b.
    // z = b; // not allowed -- only 'a' can be read here
  }
  @NonEscaping final def boric() {
    y = b;
    this.aplomb(); // allowed; a is definitely set before boric is called
    // z = c; // not allowed; c is not definitely written
  }
  @NonEscaping private def cajoled() {
    z = c;
  }
}
```

8.8.6   Fine Structure of Constructors

The code of a constructor consists of four segments, three of them optional and
one of them implicit.

1. The first segment is an optional call to `this(...)` or `super(...)`. If this
   is supplied, it must be the first statement of the constructor. If it is not
   supplied, the compiler treats it as a nullary super-call `super()`;

2. If the class or struct has properties, there must be a single `property(...)`
   command in the constructor. Every execution path through the constructor
   must go through this `property(...)` command precisely once. The sec-
   ond segment of the constructor is the code following the first segment, up
   to and including the `property()` statement.

   If the class or struct has no properties, the `property()` call must be omit-
   ted. If it is present, the second segment is defined as before. If it is absent,
   the second segment is empty.

3. The third segment is automatically generated. Fields with initializers are
   initialized immediately after the `property` statement. In the following ex-
   ample, b is initialized to `y*9000` in segment three. The initialization makes
   sense and does the right thing; b will be `y*9000` for every `Overdone` ob-
   ject. (This would not be possible if field initializers were processed earlier,
   before properties were set.)

4. The fourth segment is the remainder of the constructor body.

The segments in the following code are shown in the comments.

```
class Overlord(x:Int) {
  def this(x:Int) { property(x); }
}//Overlord
class Overdone(y:Int) extends Overlord  {
  val a : Int;
  val b =  y * 9000;
  def this(r:Int) {
    super(r);                        // (1)
    x10.io.Console.OUT.println(r); // (2)
    property(r+1);                   // (2)
    // field initializations here  // (3)
    a = r + 2;                       // (4)
  }
}//Overdone
```

The rules of what is allowed in the three segments are different, though unsurprising. For example, properties of the current class can only be read in segment 3 or 4—naturally, because they are set at the end of segment 2.

Initialization and Inner Classsses

Constructors of inner classes are tantamount to method calls on `this`. For example, the constructor for Inner **is** acceptable. It does not leak `this`. It leaks `Outer.this`, which is an utterly different object. So, the call to `this.new Inner()` in the `Outer` constructor *is* illegal. It would leak `this`. There is no special rule in effect preventing this; a constructor call of an inner class is no different from a method as far as leaking is concerned.

```
class Outer {
  static val leak : Cell[Outer] = new Cell[Outer](null);
  class Inner {
    def this() {Outer.leak.set(Outer.this);}
  }
  def /*Outer*/this() {
    //ILLEGAL: val inner = this.new Inner();
```

```
    }
  }
```

Initialization and Closures

Closures in constructors may not refer to `this`. They may not even refer to fields of `this` that have been initialized. For example, the closure `bad1` is not allowed because it refers to `this`; `bad2` is not allowed because it mentions `a` — which is, of course, identical to `this.a`.

```
class C {
  val a:Int;
  def this() {
    this.a = 1;
    // val bad1 = () => this;
    // val bad2 = () => a*10;
  }
}
```

8.8.7   Definite Initialization in Constructors

An instance field `var x:T`, when `T` has a default value, need not be explicitly initialized. In this case, `x` will be initialized to the default value of type `T`. For example, a `Score` object will have its `currently` field initialized to zero, below:

```
class Score {
  public var currently : Int;
}
```

All other sorts of instance fields do need to be initialized before they can be used. `val` fields must be initialized, even if their type has a default value. It would be silly to have a field `val z : Int` that was always given default value of `0` and, since it is `val`, can never be changed. `var` fields whose type has no default value must be initialized as well, such as `var y : Int{y != 0}`, since it cannot be assigned a sensible initial value.

The fundamental principles are:

1. `val` fields must be assigned precisely once in each constructor on every possible execution path.

2. `var` fields of defaultless type must be assigned at least once on every possible execution path, but may be assigned more than once.

3. No variable may be read before it is guaranteed to have been assigned.

4. Initialization may be by field initialization expressions (`val x : Int = y+z`), or by uninitialized fields `val x : Int;` plus an initializing assignment `x = y+z`. Recall that field initialization expressions are performed after the `property` statement, in segment 3 in the terminology of §8.8.6.

8.8.8 Summary of Restrictions on Classes and Constructors

The following table tells whether a given feature is (yes), is not (no) or is with some conditions (note) allowed in a given context. For example, a property method is allowed with the type of another property, as long as it only mentions the preceding properties. The first column of the table gives examples, by line of the following code body.

	Example	**Prop.**	`self==this`**(1)**	**Prop.Meth.**	`this`	**fields**
Type of property	(A)	yes (2)	no	no	no	no
Class Invariant	(B)	yes	yes	yes	yes	no
Supertype (3)	(C), (D)	yes	yes	yes	no	no
Property Method Body	(E)	yes	yes	yes	yes	no
Static field (4)	(F) (G)	no	no	no	no	no
Instance field (5)	(H), (I)	yes	yes	yes	yes	yes
Constructor arg. type	(J)	no	no	no	no	no
Constructor guard	(K)	no	no	no	no	no
Constructor ret. type	(L)	yes	yes	yes	yes	yes
Constructor segment 1	(M)	no	yes	no	no	no
Constructor segment 2	(N)	no	yes	no	no	no
Constructor segment 4	(O)	yes	yes	yes	yes	yes
Methods	(P)	yes	yes	yes	yes	yes

Details:

- (1) Top-level `self` only.

- (2) The type of the ith property may only mention properties 1 through i.

- (3) Super-interfaces follow the same rules as supertypes.

- (4) The same rules apply to types and initializers.

The example indices refer to the following code:

```
class Example (
   prop : Int,
   proq : Int{prop != proq},                        // (A)
   pror : Int
   )
   {prop != 0}                                      // (B)
   extends Supertype[Int{self != prop}]         // (C)
   implements SuperInterface[Int{self != prop}] // (D)
{
   property def propmeth() = (prop == pror);    // (E)
   static staticField
      : Cell[Int{self != 0}]                        // (F)
      = new Cell[Int{self != 0}](1);                // (G)
   var instanceField
      : Int {self != prop}                          // (H)
      = (prop + 1) as Int{self != prop};            // (I)
   def this(
      a : Int{a != 0},
      b : Int{b != a}                               // (J)
      )
      {a != b}                                      // (K)
      : Example{self.prop == a && self.proq==b} // (L)
   {
      super();                                      // (M)
      property(a,b,a);                              // (N)
      // fields initialized here
      instanceField = b as Int{self != prop};   // (O)
   }

   def someMethod() =
        prop + staticField + instanceField;     // (P)
```

```
  }
```

8.9 Method Resolution

Method resolution is the problem of determining, statically, which method (or
constructor or operator) should be invoked, when there are several choices that
could be invoked. For example, the following class has two overloaded `zap`
methods, one taking an `Object`, and the other a `Resolve`. Method resolution
will figure out that the call `zap(1..4)` should call `zap(Object)`, and `zap(new
Resolve())` should call `zap(Resolve)`.

Example:

```
class Res {
  public static interface Surface {}
  public static interface Deface {}

  public static class Ace implements Surface {
    public static operator (Boolean) : Ace = new Ace();
    public static operator (Place) : Ace = new Ace();
  }
  public static class Face implements Surface, Deface{}

  public static class A {}
  public static class B extends A {}
  public static class C extends B {}

  def m(x:A) = 0;
  def m(x:Int) = 1;
  def m(x:Boolean) = 2;
  def m(x:Surface) = 3;
  def m(x:Deface) = 4;


  def example() {
    assert m(100) == 1 : "Int";
    assert m(new C()) == 0 : "C";
```

```
// An Ace is a Surface, unambiguous best choice
assert m(new Ace()) == 3 : "Ace";
// ERROR: m(new Face());

// Boolean could be handled directly, or by
// implicit coercion Boolean -> Ace.
// Direct matches always win.
assert m(here) == 3 : "Place";
assert m(true) == 2 : "Boolean";


    }
```

In the "Int" *line, there is a very close match.* `100` *is an* `Int`*. In fact,* `100` *is an* `Int{self==100}`*, so even in this case the type of the actual parameter is not* precisely *equal to the type of the method.*

In the "C" *line of the example,* `new C()` *is an instance of* C*, which is a subtype of* A*, so the* A *method applies. No other method does, and so the* A *method will be invoked.*

Similarly, in the "Ace" *line, the* Ace *class implements* `Surface`*, and so* `new Ace()` *matches the* `Surface` *method.*

However, a `Face` *is both a* `Surface` *and a* `Deface`*, so there is no unique best match for the invocation* `m(new Face())`*. This invocation would be forbidden, and a compile-time error issued.*

None of the defined methods match `Place`*. However, there is an implicit coercion (§8.7.4) from* `Place` *to* `Ace`*. So, the* `Ace` *method will be called for* `m(here)` *— with the implicit coercion applied to* `here` *before the method call. (**Limitation:** : this does not currently compile in all cases.)*

For `m(true)`*, both the* `Boolean` *and, with the implicit coercion,* `Ace` *methods could apply. Since the* `Boolean` *method applies directly, and the* `Ace` *method requires an implicit coercion, this call resolves to the* `Boolean` *method, without an error.*

The basic concept of method resolution is:

1. List all the methods that could possibly be used, inferring generic types but not performing implicit coercions. If the STATIC_CALLS compiler flag is

specified, the constraints must match exactly at this step; if not, they do not, but run-time tests will be generated if necessary.

2. If one possible method is more specific than all the others, that one is the desired method.

3. If there are two or more methods neither of which is more specific than the others, then the method invocation is ambiguous. Method resolution fails and reports an error.

4. Otherwise, no possible methods were found without implicit coercions. Try the preceding steps again, but with coercions allowed. If a single most specific method is found with coercions, it is the desired method. If there are several, the invocation is ambiguous and erronious.

5. If no methods were found even with coercions, then the method invocation is undetermined. Method resolution fails and reports an error.

In the presence of X10's highly-detailed type system, some subtleties arise. One point, at least, is *not* subtle. The same procedure is used, *mutatis mutandis* for method, constructor, and operator resolution.

8.9.1 Space of Methods

X10 allows some constructs, particularly `operator`s, to be defined in a number of ways, and invoked in a number of ways. This section specifies which forms of definition could correspond to a given definiendum, and (redundantly) the syntax for invoking that definition unambiguously.

Method invocations `a.m(b)`, where `a` is an expression, can be either of the following forms. There may be any number of arguments.

- An instance method on `a`, of the form `def m(B)`, so that the invocation is `a.m(b)`;

- A static method on `a`'s class, of the form `static def m(B)`, so that the invocation is `A.m(b)`.

Static method invocations, `A.m(b)`, where `A` is a container name, can only be static. There may be any number of arguments.

- A static method on A, of the form `static def m(B)`, so that the invocation is `A.m(b)`;

Constructor invocations, `new A(b)`, must invoke constructors. There may be any number of arguments.

- A constructor on A, of the form `def this(B)`, so that the invocation is `new A(b)`.

A unary operator $\star$ a may be defined as:

- An instance operator on A, defined as `operator` $\star$ `this()`, so that the invocation is `a.operator` $\star$`()`; or

- A static operator on A, defined as `operator` $\star$`(a:A)` so that the invocation is `A.operator` $\star$`(a)`

A binary operator `a` $\star$ `b` may be defined as:

- An instance operator on A, defined as `operator this` $\star$`(b:B)`; so that the invocation is `a.operator` $\star$`(b)`, or

- A right-hand operator on B, defined as `operator (a:A)` $\star$ `this`, so that the invocation is `b.operator ()`$\star$`(b)`; or

- A static operator on A, defined as `operator (a:A)` $\star$ `(b:B)`, so that the invocation is `A.operator` $\star$`(a,b)` ; or

- A static operator on B, if A and B are different classes, defined as `operator (a:A)` $\star$ `(b:B)`, so that the invocation is `B.operator` $\star$`(a,b)`.

If none of those resolve to a method, then either operand may be cast to the other. The following alternate calls are attempted:

- `(a as B).operator`$\star$`(b)`

- `(b as A).operator () ` $\star$`(a)`

An application `a(b)`, for any number of arguments, can only come from an application definition:

- `operator this(b:B)`, so that the invocation is `a.operator()(b)`

An indexed assignment, `a(b)=c`, for any number of b's, can only come from an indexed assignment definition:

- `operator this(b:B)=(c:C) {...}`, so that the invocation is `a.operator()=(b,c)`.

An implicit coercion, as in

```
val a:A = ...;
... a as B ...
```

can come from one of two places: an implicit coercion operation defined on B, or, failing that, one defined on `A`:

1. an implicit coercion in B: `static operator (a:A):B`, so that the coercion is `B.operator[B](a)`;

2. or, failing that, an implicit coercion in `A`: `static operator this():B`, so that the coercion is `A.operator[B](a)`

An explicit conversion `a as B` can come from an explicit conversion operator, or an implicit coercion operator. X10 tries three things, in order, only checking 2 if 1 fails, and 3 if 2 fails:

1. An as operator in B: `static operator (a:A) as ?`, so that the conversion is `B.operator as[B](a)` ;

2. or, failing that, an implicit coercion in B: `static operator (a:A):B`, so that the conversion is `B.operator[B](a)`;

3. or, failing that, an implicit coercion in `A`: `static operator this():B`, so that the conversion is `A.operator[B](a)`.

8.9.2 Possible Methods

This section describes what it means for a method to be a *possible* resolution of a method invocation.

Generics introduce several subtleties, especially with the inference of generic types. For the purposes of method resolution, all that matters about a method, constructor, or operator M — we use the word "method" to include all three choices for this section — is its signature, plus which method it is. So, a typical M might look like `def m[`G_1`,..., `G_g`](`x_1`:`T_1`,..., `x_f`:`T_f`){c} =....` The code body `...` is irrelevant for the purpose of whether a given method call means M or not, so we ignore it for this section.

All that matters about a method definition, for the purposes of method resolution, is:

1. The method name `m`;

2. The generic type parameters of the method `m`, G_1,..., G_g. If there are no generic type parameters, $g = 0$.

3. The types x_1:T_1,..., x_f:T_f of the formal parameters. If there are no formal parameters, $f = 0$. In the case of an instance method, the receiver will be the first formal parameter.[6]

4. The constraint `c` of the method M. If no constraint is specified, `c` is `true`.

5. A *unique identifier* `id`, sufficient to tell the compiler which method body is intended. A file name and position in that file would suffice. The details of the identifier are not relevant.

For the purposes of understanding method resolution, we assume that all the actual parameters of an invocation are simply variables: `x1.meth(x2,x3)`. This is done routinely by the compiler in any case; the code `tbl(i).meth(true, a+1)` would be treated roughly as

```
val x1 = tbl(i);
val x2 = true;
val x3 = a+1;
x1.meth(x2,x3);
```

All that matters about an invocation I is:

1. The method name m';

---

[6]The variable names are relevant because one formal can be mentioned in a later type, or even a constraint: `def f(a:Int, b:Point{rank==a})=....`

2. The generic type parameters $G'_1, \ldots, G'_g$. If there are no generic type parameters, $g = 0$.

3. The names and types $\mathtt{x}_1 : T'_1, \ldots, \mathtt{x}_f : T'_f$ of the actual parameters. If there are no actual parameters, $f = 0$. In the case of an instance method, the receiver is the first actual parameter.

The signature of the method resolution procedure is: `resolve(invo : Invocation, context: Set[Method]) : MethodID`. Given a particular invocation and the set `context` of all methods which could be called at that point of code, method resolution either returns the unique identifier of the method that should be called, or (conceptually) throws an exception if the call cannot be resolved.

The procedure for computing `resolve(invo, context)` is:

1. Eliminate from `context` those methods which are not *acceptable*; *viz.*, those whose name, type parameters, formal parameters, and constraint do not suitably match `invo`. In more detail:

   - The method name m must simply equal the invocation name $\mathtt{m}'$;

   - X10 infers type parameters, by an algorithm given in §4.11.3.

   - The method's type parameters are bound to the invocation's for the remainder of the acceptability test.

   - The actual parameter types must be subtypes of the formal parameter types, or be coercible to such subtypes. Parameter i is a subtype if $T'_i \mathrel{<:} T_i$. It is implicitly coercible to a subtype if there is an implicit coercion operator defined from T'_i to some type U, and $U \mathrel{<:} T_i$. . If coercions are used to resolve the method, they will be called on the arguments before the method is invoked.

   - The formal constraint c must be satisfied in the invoking context.

2. Eliminate from `context` those methods which are not *available*; *viz.*, those which cannot be called due to visibility constraints, such as methods from other classes marked `private`. The remaining methods are both acceptable and available; they might be the one that is intended.

3. From the remaining methods, find the unique `ms` which is more specific than all the others, *viz.*, for which `specific(ms,mo) = true` for all other methods `mo`. The specificity test `specific` is given next.

- If there is a unique such `ms`, then `resolve(invo,context)` returns the `id` of `ms`.

- If there is not a unique such `ms`, then `resolve` reports an error.

The subsidiary procedure `specific(m1, m2)` determines whether method `m1` is equally or more specific than `m2`. `specific` is not a total order: is is possible for each one to be considered more specific than the other, or either to be more specific. `specific` is computed as:

1. Construct an invocation `invo1` based on `m1`:

   - `invo1`'s method name is `m1`'s method name;

   - `invo1`'s generic parameters are those of `m1`— simply some type variables.

   - `invo1`'s parameters are those of `m1`.

2. If `m2` is acceptable for the invocation `invo1`, `specific(m1,m2)` returns true;

3. Construct an invocation `invo2p`, which is `invo1` with the generic parameters erased. Let `invo2` be `invo2p` with generic parameters as inferred by X10's type inference algorithm. If type inference fails, `specific(m1,m2)` returns false.

4. If `m2` is acceptable for the invocation `invo2`, `specific(m1,m2)` returns true;

5. Otherwise, `specific(m1,m2)` returns false.

8.9.3   Other Disambiguations

It is possible to have a field of the same name as a method. Indeed, it is a common pattern to have private field and a public method of the same name to access it:
Example:

```
class Xhaver {
  private var x: Int = 0;
  public def x() = x;
  public def bumpX() { x ++; }
}
```

Example: *However, this can lead to syntactic ambiguity in the case where the field* f *of object* a *is a function, array, list, or the like, and where* a *has a method also named* f. *The term* a.f(b) *could either mean "call method* f *of* a *upon* b*", or "apply the function* a.f *to argument* b*".*

```
class Ambig {
  public val f : (Int)=>Int =  (x:Int) => x*x;
  public def f(y:int) = y+1;
  public def example() {
      val v = this.f(10);
      // is v 100, or 11?
  }
}
```

In the case where a syntactic form E.m(F$_1$, ..., F$_n$) could be resolved as either a method call, or the application of a field E.m to some arguments, it will be treated as a method call. The application of E.m to some arguments can be specified by adding parentheses: (E.m)(F$_1$, ..., F$_n$).

Example:

```
class Disambig {
  public val f : (Int)=>Int =  (x:Int) => x*x;
  public def f(y:int) = y+1;
  public def example() {
      assert(  this.f(10)  == 11  );
      assert( (this.f)(10) == 100 );
  }
}
```

Similarly, it is possible to have a method with the same name as a struct, say ambig, giving an ambiguity as to whether ambig() is a struct constructor invocation or a method invocation. This ambiguity is resolved by treating it as a method invocation. If the constructor invocation is desired, it can be achieved by including the optional new. That is, new ambig() is struct constructor invocation; ambig() is a method invocation.

8.10   Static Nested Classes

One class (or struct or interface) may be nested within another. The simplest way to do this is as a `static` nested class, written by putting one class definition at top level inside another, with the inner one having a `static` modifier. For most purposes, a static nested class behaves like a top-level class. However, a static inner class has access to private static fields and methods of its containing class.

Nested interfaces and static structs are permitted as well.

```
class Outer {
  private static val priv = 1;
  private static def special(n:Int) = n*n;
  public static class StaticNested {
      static def reveal(n:Int) = special(n) + priv;
  }
}
```

8.11   Inner Classes

Non-static nested classes are called *inner classes*. An inner class instance can be thought of as a very elaborate member of an object — one with a full class structure of its own. The crucial characteristic of an inner class instance is that it has an implicit reference to an instance of its containing class.

This feature is particularly useful when an instance of the inner class makes no sense without reference to an instance of the outer, and is closely tied to it. For example, consider a range class, describing a span of integers m to n, and an iterator over the range. The iterator might as well have access to the range object, and there is little point to discussing iterators-over-ranges without discussing ranges as well. In the following example, the inner class `RangeIter` iterates over the enclosing `Range`.

It has its own private cursor field `n`, telling where it is in the iteration; different iterations over the same `Range` can exist, and will each have their own cursor. It is perhaps unwise to use the name `n` for a field of the inner class, since it is also a field of the outer class, but it is legal. (It can happen by accident as well – *e.g.*, if a programmer were to add a field `n` to a superclass of the outer class, the inner class

would still work.) It does not even interfere with the inner class's ability to refer to the outer class's n field: the cursor initialization refers to the `Range`'s lower bound through a fully qualified name `Range.this.n`. Its `hasNext()` method refers to the outer class's m field, which is not shadowed.

```
class Range(m:Int, n:Int) implements Iterable[Int]{
  public def iterator ()  = new RangeIter();
  private class RangeIter implements Iterator[Int] {
     private var n : Int = m;
     public def hasNext() = n <= Range.this.n;
     public def next() = n++;
  }
  public static def main(argv:Array[String](1)) {
    val r = new Range(3,5);
    for(i in r) Console.OUT.println("i=" + i);
  }
}
```

An inner class has full access to the members of its enclosing class, both static and instance. In particular, it can access `private` information, just as methods of the enclosing class can.

An inner class can have its own members. Inside instance methods of an inner class, `this` refers to the instance of the *inner* class. The instance of the outer class can be accessed as *Outer*.`this` (where *Outer* is the name of the outer class). If, for some dire reason, it is necessary to have an inner class within an inner class, the innermost class can refer to the `this` of either outer class by using its name.

An inner class can inherit from any class in scope, with no special restrictions. `super` inside an inner class refers to the inner class's superclass. If it is necessary to refer to the outer classes's superclass, use a qualified name of the form *Outer*.`super`.

The only restriction placed on the members of inner classes is that the static fields of an inner class must be compile-time constant expressions.

An inner class `IC1` of some outer class `OC1` can be extended by another class `IC2`. However, since an `IC1` only exists as a dependent of an `OC1`, each `IC2` must be associated with an `OC1` — or a subtype thereof — as well. For example, one often extends an inner class when one extends its outer class:

```
class OC1 {
```

```
    class IC1 {}
}
class OC2 extends OC1 {
    class IC2 extends IC1 {}
}
```

The hiding of method names has one fine point. If an inner class defines a method named doit, then *all* methods named doit from the outer class are hidden — even if they have different argument types than the one defined in the inner class. They are still accessible via Outer.this.doit(), but not simply via doit(). The following code is correct, but would not be correct if the ERROR line were uncommented.

```
class Outer {
  def doit() {}
  def doit(String) {}
  class Inner {
     def doit(Boolean, Outer) {}
     def example() {
        doit(true, Outer.this);
        Outer.this.doit();
        //ERROR: doit("fails");
     }
  }
}
```

8.11.1   Constructors and Inner Classes

If IC is an inner class of OC, then instance code in the body of OC can create instances of IC simply by calling a constructor new IC(...):

```
class OC {
  class IC {}
  def method(){
     val ic = new IC();
  }
}
```

Instances of `IC` can be constructed from elsewhere as well. Since every instance of `IC` is associated with an instance of `OC`, an `OC` must be supplied to the `IC` constructor. The syntax for doing so is: `oc.new IC()`. For example:

```
/*NONSTATIC*/class OC {
  class IC {}
  static val oc1 = new OC();
  static val oc2 = new OC();
  static val ic1 = oc1.new IC();
  static val ic2 = oc2.new IC();
}
class Elsewhere{
  def method(oc : OC) {
    val ic = oc.new IC();
  }
}
```

8.12   Local Classes

Classes can be defined and instantiated in the middle of methods and other code blocks. A local class in a static method is a static class; a local class in an instance method is an inner class. Local classes are local to the block in which they are defined. They have access to almost everything defined at that point in the method; the one exception is that they cannot use `var` variables. Local classes cannot be `public` `protected`, or `private`, because they are only visible from within the block of declaration. They cannot be `static`.

Example: *The following example illustrates the use of a local class* `Local`, *defined inside the body of method* `m()`.

```
class Outer {
  val a = 1;
  def m() {
    val a = -2;
    val b = 2;
    class Local {
      val a = 3;
      def m() = 100*Outer.this.a + 10*b + a;
```

```
      }
      val l : Local = new Local();
      assert l.m() == 123;
    }//end of m()
  }
```

Note that the middle `a`, *whose value is* `-2`, *is not accessible inside of* `Local`; *it is shadowed by* `Local`'s `a` *field.* `Outer`'s `a` *is also shadowed, but the notation* `Outer.this` *gives a reference to the enclosing* `Outer` *object. There is no corresponding notation to access shadowed local variables from the enclosing block; if you need to get them, rename the fields of* `Local`.

8.13   Anonymous Classes

It is possible to define a new class and instantiate it as part of an expression. The new class can extend an extant class or interface. Its body can include all of the usual members of a class. It can refer to any identifiers available at that point in the expression — except for `var` variables. An anonymous class in a static context is a static inner class.

Anonymous classes are useful when you want to package several pieces of behavior together (a single piece of behavior can often be expressed as a function, which is syntactically lighter-weight), or if you want to extend and vary an extant class without going through the trouble of actually defining a whole new class.

The syntax for an anonymous class is a constructor call followed immediately by a braced class body: `new C(1){def foo()=2;}`.

In the following minimalist example, the abstract class `Choice` encapsulates a decision. A `Choice` has a `yes()` and a `no()` method. The `choose(b)` method will invoke one of the two. `Choice`s also have names.

The `main()` method creates a specific `Choice`. `c` is not a immediate instance of `Choice` — as an abstract class, `Choice` has no immediate instances. `c` is an instance of an anonymous class which inherits from `Choice`, but supplies `yes()` and `no()` methods. These methods modify the contents of the `Cell[Int]` `n`. (Note that, as `n` is a local variable, it would take a few lines more coding to extract `c`'s class, name it, and make it an inner class.) The call to `c.choose(true)` will call `c.yes()`, incrementing `n()`, in a rather roundabout manner.

```
abstract class Choice(name: String) {
  def this(name:String) {property(name);}
  def choose(b:Boolean) { if (b) this.yes(); else this.no(); }
  abstract def yes():void;
  abstract def no():void;
}

class Example {
  static def main(Array[String]) {
    val n = new Cell[Int](0);
    val c = new Choice("Inc Or Dec") {
      def yes() { n() += 1; }
      def no()  { n() -= 1; }
    };
    c.choose(true);
    Console.OUT.println("n=" + n());
  }
}
```

Anonymous classes have many of the features of classes in general. A few features
are unavailable because they don't make sense.

- Anonymous classes don't have constructors. Since they don't have names,
  there's no way a constructor could get called in the ordinary way. Instead,
  the `new C(...)` expression must match a constructor of the parent class `C`,
  which will be called to initialize the newly-created object of the anonymous
  class.

- Certain modifiers don't make sense for anonymous classes: `public`, `private`,
  and `protected`. Anonymous classes, being anonymous, cannot be refer-
  enced at all.

- Anonymous classes cannot be `abstract`. Since they only exist in combi-
  nation with a constructor call, they must be constructable. The parent class
  of the anonymous class may be abstract, or may be an interface; in this case,
  the anonymous class must provide all the methods that the parent demands.

- Anonymous classes cannot have explicit `extends` or `implements` clauses;
  there's no place in the syntax for them. They have a single parent and that
  is that.

9 Structs

X10 objects are a powerful general-purpose programming tool. However, the power must be paid for in space and time. In space, a typical object implementation requires some extra memory for run-time class information, as well as a pointer for each reference to the object. In time, a typical object requires an extra indirection to read or write data, and some run-time computation to figure out which method body to call.

For high-performance computing, this overhead may not be acceptable for all objects. X10 provides structs, which are stripped-down objects. They are less powerful than objects; in particular they lack inheritance and mutable fields. Without inheritance, method calls do not need to do any lookup; they can be implemented directly. Accordingly, structs can be implemented and used more cheaply than objects, potentially avoiding the space and time overhead. (Currently, the C++ back end avoids the overhead, but the Java back end implements structs as Java objects and does not avoid it.)

Structs and classes are interoperable. Both can implement interfaces (in particular, like all X10 values they implement `Any`), and subprocedures whose arguments are defined by interfaces can take both structs and classes. (Some caution is necessary here: referring to a struct through an interface requires overhead similar to that required for an object.)

In many cases structs can be converted to classes or classes to structs, within the constraints of structs. If you start off defining a struct and decide you need a class instead, the code change required is simply changing the keyword `struct` to `class`. If you have a class that does not use inheritance or mutable fields, it can be converted to a struct by changing its keyword. Client code using the struct that was a class will need certain changes: *e.g.*, the `new` keyword must be added in constructor calls, and structs (unlike classes) do not have default values.

9.1 Struct declaration

StructDecl	::=	*Mods$^?$* struct *Id TypeParamsWithVariance$^?$*	*(20.33)*
		Properties$^?$ WhereClause$^?$ Interfaces$^?$ ClassBody	
TypeParamsWithVariance	:=	[*TypeParamWithVarianceList*]	*(20.24)*
Properties	::=	(*PropertyList*)	*(20.5)*
WhereClause	::=	*DepParams*	*(20.29)*
Interfaces	::=	implements *InterfaceTypeList*	*(20.115)*
ClassBody	::=	{ *ClassBodyDecls$^?$* }	*(20.117)*

All fields of a struct must be val.

A struct S cannot contain a field of type S, or a field of struct type T which, recursively, contains a field of type S. This restriction is necessary to permit S to be implemented as a contiguous block of memory of size equal to the sum of the sizes of its fields.

Values of a struct C type can be created by invoking a constructor defined in C. Unlike for classes, the new keyword is optional for struct constructors.

Example: *Leaving out* new *can improve readability in some cases:*

```
struct Polar(r:Double, theta:Double){
  def this(r:Double, theta:Double) {property(r,theta);}
  static val Origin = Polar(0,0);
  static val x0y1  = Polar(1, 3.14159/2);
  static val x1y0  = new Polar(1, 0);
}
```

When a struct and a method have the same name, new *may be used to resolve to the struct's constructor.*

```
struct Ambig(x:Int) {
  static def Ambig(x:Int) = "Breaks capitalization conventions!";
  static def example() {
    val useMethod      = Ambig(1);
    val useConstructor = new Ambig(2);
  }
}
```

Structs support the same notions of generics, properties, and constrained types that classes do.

Example:

```
struct Exam[T](nQuestions:Int){T <: Question} {
  public static interface Question {}
  // ...
}
```

9.2   Boxing of structs

If a struct S implements an interface I (*e.g.*, `Any`), a value `v` of type S can be assigned to a variable of type I. The implementation creates an object `o` that is an instance of an anonymous class implementing I and containing `v`. The result of invoking a method of I on `o` is the same as invoking it on `v`. This operation is termed *auto-boxing*. It allows full interoperability of structs and objects—at the cost of losing the extra efficiency of the structs when they are boxed.

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

9.3   Optional Implementation of `Any` methods

Two structs are equal (==) if and only if their corresponding fields are equal (==).

All structs implement `x10.lang.Any`. Structs are required to implement the following methods from `Any`. Programmers need not provide them; X10 will produce them automatically if the program does not include them.

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

A programmer who provides an explicit implementation of `equals(Any)` for a struct S should also consider supplying a definition for `equals(S):Boolean`. This will often yield better performance since the cost of an upcast to `Any` and then a downcast to S can be avoided.

9.4 Primitive Types

Certain types that might be built in to other languages are in fact implemented as structs in package `x10.lang` in X10. Their methods and operations are often provided with `@Native` (§18) rather than X10 code, however. These types are:

```
Boolean, Char, Byte, Short, Int, Long
Float, Double, UByte, UShort, UInt, ULong
```

9.4.1 Signed and Unsigned Integers

X10 has an unsigned integer type corresponding to each integer type: `UInt` is an unsigned `Int`, and so on. These types can be used for binary programming, or when an extra bit of precision for counters or other non-negative numbers is needed in integer arithmetic. However, X10 does not otherwise encourage the use of unsigned arithmetic.

9.5 Generic programming with structs

The programmer must be aware of the different interpretations of equality for structs and classes and ensure that the code is correctly written for both cases.

It is occasionally necessary to tell what kind of value x is: object, struct, or function. Three static methods on `x10.lang.System` provide this test: `System.isObject(x)` returns true if x is a value of `Object` type, including `null`; `System.isStruct(x)` returns true if x is a `struct`; `System.isFunction(x)` returns true if x is a closure value. Precisely one of these three functions returns true for any X10 value x.

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

9.6   Example structs

`x10.lang.Complex` provides a detailed example of a practical struct, suitable for use in a library. For a shorter example, we define the `Pair` struct—available in `x10.util.Pair`. A `Pair` packages two values of possibly unrelated type together in a single value, *e.g.*, to return two values from a function.

`divmod` computes the quotient and remainder of `a \div b` (naively). It returns both, packaged as a `Pair[UInt, UInt]`. Note that the constructor uses type inference, and that the quotient and remainder are accessed through the `first` and `second` fields.

```
struct Pair[T,U] {
    public val first:T;
    public val second:U;
    public def this(first:T, second:U):Pair[T,U] {
        this.first = first;
        this.second = second;
    }
    public def toString() = "(" + first + ", " + second + ")";
}
class Example {
  static def divmod(var a:UInt, b:UInt): Pair[UInt, UInt] {
     assert b > 0;
     var q : UInt = 0;
     while (a > b) {q++; a -= b;}
     return Pair(q, a);
  }
  static def example() {
     val qr = divmod(22, 7);
     assert qr.first == 3 && qr.second == 1;
  }
}
```

9.7   Nested Structs

Static nested structs may be defined, essentially as static nested classes except for making them structs (§8.10). Inner structs may be defined, essentially as inner

classes except making them structs (§8.11).

9.8   Converting Between Classes And Structs

Code written using structs can be modified to use classes, or vice versa. Caution must be used in certain places.

Class and struct *definitions* are syntactically nearly identical: change the `class` keyword to `struct` or vice versa. Of course, certain important class features can't be used with structs, such as inheritance and `var` fields.

Converting code that *uses* the class or struct requires a certain amount of caution. Suppose, in particular, that we want to convert the class `Class2Struct` to a struct, and `Struct2Class` to a class.

```
class Class2Struct {
  val a : Int;
  def this(a:Int) { this.a = a; }
  def m() = a;
}
struct Struct2Class {
  val a : Int;
  def this(a:Int) { this.a = a; }
  def m() = a;
}
```

1. Class constructors require the `new` keyword; struct constructors allow it but do not require it. `Struct2Class(3)` to will need to be converted to `new Struct2Class(3)`.

2. Objects and structs have different notions of ==. For objects, == means "same object"; for structs, it means "same contents". Before conversion, both `assert`s in the following program succeed. After converting and fixing constructors, both of them fail.

   ```
   val a = new Class2Struct(2);
   val b = new Class2Struct(2);
   assert a != b;
   val c = Struct2Class(3);
   ```

```
val d = Struct2Class(3);
assert c==d;
```

3. Objects can be set to `null`. Structs cannot. There is no default value like `null` available for structs.

10 Functions

10.1 Overview

Functions, the last of the three kinds of values in X10, encapsulate pieces of code which can be applied to a vector of arguments to produce a value. Functions, when applied, can do nearly anything that any other code could do: fail to terminate, throw an exception, modify variables, spawn activities, execute in several places, and so on. X10 functions are not mathematical functions: the `f(1)` may return `true` on one call and `false` on an immediately following call.

A *function literal* `(x1:T1,..,xn:Tn){c}:T=>e` creates a function of type `(x1:T1,...,xn:Tn){c}=>T` (§4.6). For example, `(x:Int):Int => x*x` is a function literal describing the squaring function on integers. `null` is also a function value.

Limitation: X10 functions cannot have type arguments or constraints.

Function application is written `f(a,b,c)`, following common mathematical usage.

The function body may be a block. To compute integer squares by repeated addition (inefficiently), one may write:

```
val sq: (Int) => Int
    = (n:Int) => {
        var s : Int = 0;
        val abs_n = n < 0 ? -n : n;
        for (i in 1..abs_n) s += abs_n;
        s
      };
```

A function literal evaluates to a function entity f. When f is applied to a suitable list of actual parameters `a1` through `an`, it evaluates `e` with the formal parameters

bound to the actual parameters. So, the following are equivalent, where e is an
expression involving x1 and x2[1]

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

and

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

This equivalence does not hold if the body is a statement rather than an expres-
sion. A few language features are forbidden (break or continue of a loop that
surrounds the function literal) or mean something different (return inside a func-
tion returns from the function, not the surrounding block).

The *method selector expression* e.m.(x1:T1,...,xn:Tn) (§10.3) permits the
specification of the function underlying the method m, which takes arguments of
type (x1:T1,..., xn:Tn). Within this function, this is bound to the result of
evaluating e.

Function types may be used in implements clauses of class definitions. Suitable
operator definitions must be supplied, with public operator this(x1:T1,
..., xn:Tn) declarations. Instances of such classes may be used as functions of
the given type. Indeed, an object may behave like any (fixed) number of functions,
since the class it is an instance of may implement any (fixed) number of function

---

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

types. *e.g.* Instances of the `Funny` class behave like two functions: a constant function on Booleans, and a linear function on pairs of `Int`s.

```
class Funny implements (Boolean) => Int,
                       (Int, Int) => Int
{
  public operator this(Boolean) = 1;
  public operator this(x:Int, y:Int) = 10*x+y;
  static def example() {
    val f <: Funny  = new Funny();
    assert f(true) == 1; // (Boolean)=>Int behavior
    assert f(1,2) == 12; // (Int,Int)=>Int behavior
  }
}
```

10.2   Function Literals

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

ClosureExp	::=	*FormalParams* *WhereClause*$^?$ *HasResultType*$^?$	*(20.92)*
		Offers$^?$ *=> ClosureBody*	
FormalParams	::=	(*FormalParamList*$^?$)	*(20.26)*
WhereClause	::=	*DepParams*	*(20.29)*
DepParams	::=	{ *ExistentialList*$^?$ *Conjunction*$^?$ }	*(20.23)*
ExistentialList	::=	*FormalParam*	*(20.30)*
	|	*ExistentialList* ; *FormalParam*	
HasResultType	::=	: *Type*	*(20.127)*
	|	<: *Type*	
ClosureBody	::=	*ConditionalExp*	*(20.94)*
	|	*Annotations*$^?$ { *BlockStatements*$^?$ *LastExp* }	
	|	*Annotations*$^?$ *Block*	

Functions have zero or more formal parameters and an optional return type. The body has the same syntax as a method body; it may be either an expression, a block of statements, or a block terminated by an expression to return. In particular, a value may be returned from the body of the function using a return statement (§12.13).

The type of a function is a function type as described in §4.6. In some cases the
return type T is also optional and defaults to the type of the body. If a formal xi
does not occur in any Tj, c, T or e, the declaration xi:Ti may be replaced by just
Ti. *E.g.*, (Int)=>7 is the integer function returning 7 for all inputs.

As with methods, a function may declare a guard to constrain the actual parame-
ters with which it may be invoked. The guard may refer to the type parameters,
formal parameters, and any vals in scope at the function expression.

Example:

```
val n = 3;
val f : (x:Int){x != n} => Int
      = (x:Int){x != n} => (12/(n-x));
Console.OUT.println("f(5)=" + f(5));
```

The body of the function is evaluated when the function is invoked by a call ex-
pression (§11.6), not at the function's place in the program text.

As with methods, a function with return type void cannot have a terminating
expression. If the return type is omitted, it is inferred, as described in §4.11.
It is a static error if the return type cannot be inferred. *E.g.*, (Int)=>null is
not well-specified; X10 does not know which type of null is intended. But
(Int):Array[Double](1) => null is legal.

Example 10.2.1 The following method takes a function parameter and uses it
to test each element of the list, returning the first matching element. It returns
absent if no element matches.


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

The method may be invoked thus:

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

□

10.2.1 Outer variable access

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

- All fields of the enclosing object(s) and class(es);

- All type parameters;

- All `val` variables;

`var` variables cannot be accessed.

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

e.g. The following is legal. However, the commented-out line would not be legal. Note that `a` is not a local `var` variable. It is a field of `this`. A reference to `a` is simply short for `this.a`, which is a use of a `val` variable (`this`).

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

10.3 Method selectors

A method selector expression allows a method to be used as a first-class function, without writing a function expression for it.

Example: *Consider a class* Span *defining ranges of integers.*

```
class Span(low:Int, high:Int) {
  def this(low:Int, high:Int) {property(low,high);}
  def between(n:Int) = low <= n && n <= high;
  static def example() {
    val digit = new Span(0,9);
    val isDigit : (Int) => Boolean = digit.between.(Int);
    assert isDigit(8);
  }
}
```

In example(), digit.between.(Int) *is a unary function testing whether its argument is between zero and nine. It could also be written* (n:Int) => digit.between(n).

MethodSelection	::=	*MethodName* . (*FormalParamList$^?$*)	(20.164)
	|	*Primary* . *Id* . (*FormalParamList$^?$*)	
	|	super . *Id* . (*FormalParamList$^?$*)	
	|	*ClassName* . super . *Id* . (*FormalParamList$^?$*)	

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

Thus, the method selector

```
e.m.(T₁, ..., Tₙ)
```

behaves as if it were the function

```
((v:V)=>
  (x₁: T₁, ..., xₙ: Tₙ){c}
  => v.m(x₁, ..., xₙ))
(e)
```

Because of overloading, a method name is not sufficient to uniquely identify a function for a given class. One needs the argument type information as well. The

selector syntax (dot) is used to distinguish `e.m()` (a method invocation on `e` of method named `m` with no arguments) from `e.m.()` (the function bound to the method).

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

There is no difference between using a function defined directly directly using the function syntax, or obtained via static or instance function selectors.

10.4 Operator functions

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

```
String.+   ≡ (x: String, y: String): String => x + y
Long.-     ≡ (x: Long, y: Long): Long => x - y
Float.-    ≡ (x: Float, y: Float): Float => x - y
Boolean.&  ≡ (x: Boolean, y: Boolean): Boolean => x & y
Int.<      ≡ (x: Int, y: Int): Boolean => x < y
```

Unary and binary promotion (§11.9) is not performed when invoking these operations; instead, the operands are coerced individually via implicit coercions (§11.27), as appropriate.

10.5 Functions as objects of type `Any`

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

---

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

`f == g`, then `f(1)` can be substituted for `g(1)` and the result will be identical. However, there is no guarantee that `f(1)==g(1)` will evaluate to true. Indeed, there is no guarantee that `f(1)==f(1)` will evaluate to true either, as `f` might be a function which returns n on its nth invocation. However, `f(1)==f(1)` and `f(1)==g(1)` are interchangeable.)

Every function type implements all the methods of `Any`. `f.equals(g)` is equivalent to `f==g`. `f.hashCode()`, `f.toString()`, and `f.typeName()` are implementation-dependent, but respect `equals` and the basic contracts of `Any`.

11 Expressions

X10 has a rich expression language. Evaluating an expression produces a value, or, in a few cases, no value. Expression evaluation may have side effects, such as change of the value of a `var` variable or a data structure, allocation of new values, or throwing an exception.

11.1 Literals

Literals denote fixed values of built-in types. The syntax for literals is given in §3.

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

11.2 `this`

Primary	::=	`this` *(20.157)*
	|	`this`
	|	*ClassName* `. this`

The expression `this` is a local `val` containing a reference to an instance of the lexically enclosing class. It may be used only within the body of an instance method, a constructor, or in the initializer of a instance field – that is, the places where there is an instance of the class under consideration.

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

Example: `Outer` *is a class containing* `Inner`. *Each instance of* `Inner` *has a reference* `Outer.this` *to the* `Outer` *involved in its creation.* `Inner` *has access to*

151

the fields of `Outer.this`, *as seen in the* `outerThree` *and* `alwaysTrue` *methods.*
Note that `Inner` *has its own* `three` *field, which is different from and not even the*
same type as `Outer.this.three`.

```
class Outer {
  val three = 3;
  class Inner {
    val three = "THREE";
    def outerThree() = Outer.this.three;
    def alwaysTrue() = outerThree() == 3;
  }
}
```

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

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

11.3   Local variables

Id	::=	*identifier*	*(20.146)*

A local variable expression consists simply of the name of the local variable, field
of the current object, formal parameter in scope, etc. It evaluates to the value of
the local variable.

Example: `n` *in the second line below is a local variable expression. The* `n` *in the*
first line is not; it is part of a local variable declaration.

```
val n = 22;
val m = n + 56;
```

11.4 Field access

FieldAccess	::=	*Primary . Id*	*(20.162)*
		`super` . *Id*	
		ClassName . `super` . *Id*	
		Primary . `class`	
		`super` . `class`	
		ClassName . `super` . `class`	

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

The type of the access is the declared type of the field with the actual target substituted for `this` in the type.

Example: *The declaration of* `b` *below has a constraint involving* `this`. *The use of an instance of it,* `f.b`, *has the same constraint involving* `f` *instead of* `this`, *as required.*

```
class Fielded {
  public val a : Int = 1;
  public val b : Int{this.a == b} = this.a;
  static def example() {
    val f : Fielded = new Fielded();
    val fb : Int{fb == f.a} = f.b;
  }
}
```

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

If the field target is given by the keyword `super`, the target's type is the superclass of the enclosing class. This form is used to access fields of the parent class shadowed by same-named fields of the current class.

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

Example: *This illustrates all four cases of field access.*

```
class Uncle {
```

```
  public static val f = 1;
}
class Parent {
  public val f = 2;
}
class Ego extends Parent {
  public val f = 3;
  class Child extends Ego {
    public val f = 4;
    def classNameDotId() =  Uncle.f;      // 1
    def cnDotSuperDotId() = Ego.super.f; // 2
    def superDotId() =      super.f;     // 3
    def expDotId() =        this.f;      // 4
  }
}
```

If the field target is `null`, a `NullPointerException` is thrown. If the field target is a class name, a static field is selected. It is illegal to access a field that is not visible from the current context. It is illegal to access a non-static field through a static field access expression. However, it is legal to access a static field through a non-static reference.

11.5   Function Literals

Function literals are described in §10.

11.6   Calls

MethodInvocation	::=	*MethodPrimaryPrefix* (*ArgumentList$^?$*)	*(20.163)*
	|	*MethodSuperPrefix* (*ArgumentList$^?$*)	
	|	*MethodClassNameSuperPrefix* (*ArgumentList$^?$*)	
	|	*MethodName TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	*Primary . Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	`super` . *Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	*ClassName* . `super` . *Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	*Primary TypeArguments$^?$* (*ArgumentList$^?$*)	
ArgumentList	::=	*Exp*	*(20.161)*
	|	*ArgumentList , Exp*	

A *MethodInvocation* may be to either a `static` method, an instance method, or a closure.

The syntax for method invocations is ambiguous. `ob.m()` could either be the invocation of a method named `m` on object `ob`, or the application of a function held in a field `ob.m`. The target `ob` must be type-checked to determine which of these it is. It is a static error if both cases are possible after type checking.

Example:

```
class Callsome {
  static val closure : () => Int = () => 1;
  static def method()            = 2;
  static def example() {
     assert Callsome.closure() == 1;
     assert Callsome.method()  == 2;
  }
}
```

However, adding a static method [mis]named `closure` *makes* `Callsome.closure()` *ambiguous: it could be a call to the closure, or to the static method:*

```
static def closure () = 3;
// ERROR: static errory = Callsome.closure();
```

The application form `e(f,g)`, when `e` evaluates to an object or struct, invokes the application `operator`, defined in the form

```
public operator this(f:F, g:G) = "value";
```

Method selection rules are given in §8.9.

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

Example: *In this example, a* DivideBy *object provides the service of dividing numbers by* denom — *so long as* denom *is not zero. In the* example *method,* this.div(100) *is not allowed; there is no guarantee that* denom != 0. *Casting* this *to a type whose constraint implies* denom != 0 *permits the method call.*

```
class DivideBy(denom:Int) {
  def div(numer:Int){denom != 0} = numer / denom;
  def example() {
     val thisCast = (this as DivideBy{self.denom != 0});
     thisCast.div(100);
     //ERROR: this.div(100);
  }
}
```

11.7   Assignment

Assignment	::=	*LeftHandSide AssignmentOperator AssignmentExp*	*(20.186)*
	|	*ExpName* (*ArgumentList*$^?$) *AssignmentOperator AssignmentExp*	
	|	*Primary* (*ArgumentList*$^?$) *AssignmentOperator AssignmentExp*	
LeftHandSide	::=	*ExpName*	*(20.187)*
	|	*FieldAccess*	
AssignmentOperator	::=	=	*(20.188)*
	|	*=	
	|	/=	
	|	%=	
	|	+=	
	|	-=	
	|	<<=	
	|	>>=	
	|	>>>=	
	|	&=	
	|	^=	
	|	|=	

The assignment expression `x = e` assigns a value given by expression `e` to a variable `x`. Most often, `x` is mutable, a `var` variable. The same syntax is used for delayed initialization of a `val`, but `val`s can only be initialized once.

```
var x : Int;
val y : Int;
x = 1;
y = 2; // Correct; initializes y
x = 3;
// ERROR: y = 4;
```

There are three syntactic forms of assignment:

1. `x = e;`, assigning to a local variable, formal parameter, field of `this`, etc.

2. `x.f = e;`, assigning to a field of an object.

3. `a(i`$_1$`,...,i`$_n$`) = v;`, where $n \geq 0$, assigning to an element of an array or some other such structure. This is an operator call (§8.7). For well-behaved classes it works like array assignment, mutatis mutandis, but there is no actual guarantee, and the compiler makes no assumptions about how this works for arbitrary `a`. Naturally, it is a static error if no suitable assignment operator for `a`.

For a binary operator ⋄, the ⋄-assignment expression `x ⋄= e` combines the current value of `x` with the value of `e` by ⋄, and stores the result back into `x`. `i += 2`, for example, adds 2 to `i`. For variables and fields, `x ⋄= e` behaves just like `x = x ⋄ e`.

The subscripting forms of `a(i) ⋄= b` are slightly subtle. Subexpressions of `a` and `i` are only evaluated once. However, `a(i)` and `a(i)=c` are each executed once—in particular, there is one call to the application operator, and one to the assignment operator. If subscripting is implemented strangely for the class of `a`, the behavior is *not* necessarily updating a single storage location. Specifically, `A()(I()) += B()` is tantamount to:

```
{
  val aa = A();  // Evaluate A() once
  val ii = I();  // Evaluate I() once
  val bb = B();  // Evaluate B() once
  val tmp = aa(ii) + bb; // read aa(ii)
  aa(ii) = tmp;  // write sum back to aa(ii)
}
```

Limitation: `+=` does not currently meet this specification.

11.8   Increment and decrement

The operators `++` and `--` increment and decrement a variable, respectively. `x++` and `++x` both increment `x`, just as the statement `x += 1` would, and similarly for `--`.

The difference between the two is the return value. `++x` and `--x` return the *new* value of `x`, after incrementing or decrementing. `x++` and `x--` return the *old* value of `x`, before incrementing or decrementing.

Limitation: This currently only works for numeric types.

11.9 Numeric Operations

Numeric types (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Complex`, and unsigned variants of fixed-point types) are normal X10 structs, though most of their methods are implemented via native code. They obey the same general rules as other X10 structs. For example, numeric operations, coercions, and conversions are defined by `operator` definitions, the same way you could for any struct.

Promoting a numeric value to a longer numeric type preserves the sign of the value. For example, `(255 as UByte) as UInt` is 255.

Most of these operations can be defined on user-defined types as well. While it is good practice to keep such operations consistent with the numeric operations whenever possible, the compiler neither enforces nor assumes any particular semantics of user-defined operations.

11.9.1 Conversions and coercions

Specifically, each numeric type can be converted or coerced into each other numeric type, perhaps with loss of accuracy.

```
val n : Byte = 123 as Byte; // explicit
val f : (Int)=>Boolean = (Int) => true;
val ok = f(n); // implicit
```

11.9.2 Unary plus and unary minus

The unary + operation on numbers is an identity function. The unary – operation on numbers is a negation function. On unsigned numbers, these are two's-complement. For example, `-(0x0F as UByte)` is `(0xF1 as UByte)`.

11.10 Bitwise complement

The unary ˜ operator, only defined on integral types, complements each bit in its operand.

11.11   Binary arithmetic operations

The binary arithmetic operators perform the familiar binary arithmetic operations:
+ adds, − subtracts, * multiplies, / divides, and % computes remainder.

On integers, the operands are coerced to the longer of their two types, and then operated upon. Floating point operations are determined by the IEEE 754 standard. The integer / and % throw an exception if the right operand is zero.

11.12   Binary shift operations

The operands of the binary shift operations must be of integral type. The type of the result is the type of the left operand. The right operand, describing a number of bits, must be unsigned: x << 1U.

If the promoted type of the left operand is Int, the right operand is masked with 0x1f using the bitwise AND (&) operator, giving a number at most the number of bits in an Int. If the promoted type of the left operand is Long, the right operand is masked with 0x3f using the bitwise AND (&) operator, giving a number at most the number of bits in a Long.

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

The >>> operator right-shifts the left operand by the number of bits given by the right operand. The result is not sign extended; that is, if the right operand is k, the most significant k bits of the result are set to 0. This operation is deprecated, and may be removed in a later version of the language.

11.13   Binary bitwise operations

The binary bitwise operations operate on integral types, which are promoted to the longer of the two types. The & operator performs the bitwise AND of the promoted operands. The | operator performs the bitwise inclusive OR of the promoted operands. The ˆ operator performs the bitwise exclusive OR of the promoted operands.

11.14   String concatenation

The + operator is used for string concatenation as well as addition. If either operand is of static type `x10.lang.String`, the other operand is converted to a `String` , if needed, and the two strings are concatenated. String conversion of a non-`null` value is performed by invoking the `toString()` method of the value. If the value is `null`, the value is converted to `"null"`.

The type of the result is `String`.

For example, `"one " + 2 + here` evaluates to `one 2(Place 0)`.

11.15   Logical negation

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

11.16   Boolean logical operations

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

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

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

11.17   Boolean conditional operations

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

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

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

11.18   Relational operations

The relational operations on numeric types compare numbers, producing `Boolean` results.

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

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

11.19   Conditional expressions

ConditionalExp	::=	*ConditionalOrExp*	*(20.184)*
	|	*ClosureExp*	
	|	*AtExp*	
	|	*FinishExp*	
	|	*ConditionalOrExp* ? *Exp* : *ConditionalExp*	

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

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

Example: `a == b ? 1 : 2` *evaluates to* `1` *if* a *and* b *are the same, and* `2` *if they are different. As the type of* `1` *is* `Int{self==1}` *and of* `2` *is* `Int{self==2}`, *the type of the conditional expression has the form* `Int{c}`, *where* `self==1` *and*

`self==2` *both imply* c. *For example, it might be* `Int{true}` *– or perhaps it might be* `Int{self != 8}`. *Note that this term has no most accurate type in the X10 type system.*

The subexpression not selected is not evaluated.

Example: *The following use of the conditional expression prevents division by zero. If* den==0, *the division is not performed at all.*

```
(den == 0) ? 0 : num/den
```

Similarly, the following code performs a method call if op *is non-null, and avoids the null pointer error if it is null. Defensive coding like this is quite common when working with possibly-null objects.*

```
(ob == null) ? null : ob.toString();
```

11.20 Stable equality

EqualityExp	::=	*RelationalExp*	*(20.178)*
	|	*EqualityExp* == *RelationalExp*	
	|	*EqualityExp* != *RelationalExp*	
	|	*Type* == *Type*	

The == and != operators provide a fundamental, though non-abstract, notion of equality. a==b is true if the values of a and b are extremely identical.

- If a and b are values of object type, then a==b holds if a and b are the same object.

- If one operand is `null`, then a==b holds iff the other is also `null`.

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

- The definition of equality for function types is specified in §10.5.

- No implicit coercions are performed by ==.

- It is a static error to have an expression a == b if the types of a and b are disjoint.

a != b is true iff a==b is false.

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

== provides a *stable* notion of equality. If two values are == at any time, they remain == forevermore, regardless of what happens to the mutable state of the program.

Example: *Regardless of the values and types of* a *and* b*, or the behavior of* any_code_at_all *(which may, indeed, be any code at all—not just a method call), the value of* a==b *does not change:*

```
val a = something();
val b = something_else();
val eq1 = (a == b);
any_code_at_all();
val eq2 = (a == b);
assert eq1 == eq2;
```

11.20.1   No Implicit Coercions

== is a primitive operation in X10 – one of very few. Most operations, like + and <=, are defined as operators. == and != are not. As non-operators, they need not and do not follow the general method resolution procedure of §8.9. In particular, while operators perform implicit conversions on their arguments, == and != do not.

The advantage of this restriction is that =='s behavior is as simple and efficient as possible. It never runs user-defined code, and the compiler can analyze and understand it in detail – and guarantee that it is efficient.

The disadvantage is that certain straightforward-looking idioms do not work. One may not, for example, write

```
// NOT ALLOWED
for(var i : Long = 0; i != 100; i++)
```

A Long like i can never == an Int like 100.

We can write `i = i + 1;`, adding an `Int` to `i`. This works because the expression uses +, an ordinary `operator`. There is an implicit coercion from `Int` to `Long`, so the `1` can be converted to `1L`, which can be added to `i`.

However, == does not permit implicit coercions, and so the `100` stays an `Int`. The loop must be written with a comparison of two `Long`s:

```
for(var i : Long = 0; i != 100L; i++)
```

Incidentally, it could also be written

```
for(var i : Long = 0; i <= 100; i++)
```

The operation <= is a regular operator, and thus uses coercions in its arguments, so `100` gets coerced to `100L`.

11.20.2   Non-Disjointness Requirement

It is a static error to have an expression `a==b` where `a` and `b` could not possibly be equal, based on their types. This is a practical codicil to §11.20.1. Consider the illegal code

```
// NOT ALLOWED
for(var i : Long = 0; i != 100; i++)
```

`100` and `100L` are different values; they are not ==. A coercion could make them equal, but == does not allow coercions. So, if `100 == 100L` were going to return anything, it would have to return `false`. This would have the unfortunate effect of making the `for` loop diverge.

Since this and related idioms are so common, and since so many programmers are used to languages which are less precise about their numeric types, X10 avoids the mistake by declaring it a static error in most cases. Specifically, `a==b` is not allowed if, by inspection of the types, `a` and `b` could not possibly be equal.

- Numbers of different base types cannot be compared for equality. `100==100L` is a static error. To compare numbers, explicitly cast them to the same type: `100 as Long == 100L`.

- Indeed, structs of different types cannot be equal, and so they cannot be compared for equality.

- For objects, the story is different. Unconstrained object types can always be compared for equality. Given objects `a:Person` and `b:Theory`, `a==b` could be true if `a==null` and `b==null`.

- Constrained object types may or may not be comparable. For example, if `Person` and `Theory` are both direct subclasses of `Object`, and `a:Person{self!=null}` and `b:Theory`, then `a==b` is not allowed, since the two could not possibly be equal.

- Explicit casts erase type information. If you wanted to have a comparison `a==b` for `a:Person{self!=null}` and `b:Theory`, you could write it as `a as Object == b as Object`. It would, of course, return `false`, but it would not be a compiler error.[1] A struct and an object may both be cast to `Any` and compared for equality, though they, too, will always be different.

11.21   Allocation

ClassInstCreationExp	::=	`new` *TypeName TypeArguments*$^?$ (*ArgumentList*$^?$) *(20.12)*
		ClassBody$^?$
	|	`new` *TypeName* [*Type*] [*ArgumentList*$^?$]
	|	*Primary* . `new` *Id TypeArguments*$^?$ (*ArgumentList*$^?$) *ClassBody*$^?$
	|	*AmbiguousName* . `new` *Id TypeArguments*$^?$ (*ArgumentList*$^?$) *ClassBody*$^?$

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

The allocation expression may have an optional class body. In this case, an anonymous subclass of the given class is allocated. An anonymous class allocation may also specify a single super-interface rather than a superclass; the superclass of the anonymous class is `x10.lang.Object`.

If the class is anonymous—that is, if a class body is provided—then the constructor is selected from the superclass. The constructor to invoke is selected using the same rules as for method invocation (§11.6).

---

[1]Code generators often find this trick too be useful.

The type of an allocation expression is the return type of the constructor invoked, with appropriate substitutions of actual arguments for formal parameters, as specified in §11.6.

It is illegal to allocate an instance of an `abstract` class. The usual visibility rules apply to allocations: it is illegal to allocate an instance of a class or to invoke a constructor that is not visible at the allocation expression.

Note that instantiating a struct type can use function application syntax; `new` is optional. As structs do not have subclassing, there is no need or possibility of a *ClassBody*.

11.22 Casts

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

CastExp	::=	*Primary*	*(20.86)*
			ExpName	
			CastExp `as` *Type*	

The result of this operation is a value of the given type if the cast is permissible at run time, and either a compile-time error or a runtime exception (`x10.lang.TypeCastException`) if it is not.

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

- If `T` is a primitive type, then `E`'s value is converted to type `T` according to the rules of §11.27.1.

- If `T` is a class, then the first half of the cast succeeds if the run-time value of `E` is an instance of class `T`, or of a subclass.

- If `T` is an interface, then the first half of the cast succeeds if the run-time value of `E` is an instance of a class implementing `T`.

- If `T` is a struct type, then the first half of the cast succeeds if the run-time value of `E` is an instance of `T`.

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

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

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

11.23 `instanceof`

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

RelationalExp	::=	*RangeExp*	*(20.177)*
	|	*SubtypeConstraint*	
	|	*RelationalExp* < *RangeExp*	
	|	*RelationalExp* > *RangeExp*	
	|	*RelationalExp* <= *RangeExp*	
	|	*RelationalExp* >= *RangeExp*	
	|	*RelationalExp* `instanceof` *Type*	
	|	*RelationalExp* `in` *ShiftExp*	

In the above expression, *Type* is any type. At run time, the result of `e instanceof T` is `true` if the value of `e` is an instance of type `T`. Otherwise the result is `false`. This determination may involve checking that the constraint, if any, associated with the type is true for the given expression.

For example, `3 instanceof Int{self==x}` is an overly-complicated way of saying `3==x`.

However, it is a static error if `e` cannot possibly be an instance of `C{c}`; the compiler will reject `1 instanceof Int{self == 2}` because 1 can never satisfy `Int{self == 2}`. Similarly, `1 instanceof String` is a static error, rather than an expression always returning false.

Limitation: X10 does not currently handle `instanceof` of generics in the way you might expect. For example, `r instanceof Array[Int{self != 0}]` does not test that every element of `r` is non-zero; instead, the compiler rejects it.

11.24 Subtyping expressions

SubtypeConstraint	::=	*Type* <: *Type*	*(20.28)*
	|	*Type* :> *Type*	

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

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

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

Example: *Subtyping expressions are particularly useful in giving constraints on generic types.* `x10.util.Ordered[T]` *is an interface whose values can be compared with values of type* `T`. *In particular,* `T <: x10.util.Ordered[T]` *is true if values of type* `T` *can be compared to other values of type* `T`. *So, if we wish to define a generic class* `OrderedList[T]`, *of lists whose elements are kept in the right order, we need the elements to be ordered. This is phrased as a constraint on* `T`:

```
class OrderedList[T]{T <: x10.util.Ordered[T]} {
  // ...
}
```

11.25 Contains expressions

RelationalExp	::=	*RelationalExp* `in` *ShiftExp*	*(20.177)*

`in` is a binary operator, definable in §8.7. It is conventionally used for checking containment.

Example: *The built-in type* `Region` *provides* `in`, *testing whether a* `Point` *is in the region:*

```
assert 3 in 1..10;
assert !(10 in 1..3);
```

Other types can provide them as well:

```
class Cont {
   operator this in (Int) = true;
   operator (String) in this = false;
   static operator (Cont) in (b:Boolean) = b;
   static def example() {
      val c:Cont = new Cont();
      assert c in 4 && !("odd" in c) && (c in true);
   }
}
```

11.26   Array Constructors

   Primary   ::=   [*ArgumentList*$^?$]

X10 includes short syntactic forms for constructing one-dimensional arrays. The shortest form is to enclose some expressions in brackets:

```
val ints <: Array[Int](1) = [1,3,7,21];
```

The expression [e_1, ..., e_n] produces an n-element `Array[T](1)`, where `T` is the computed common supertype (§4.9) of the **base types** of the expressions e_i.

Example: *The type of* [0,1,2] *is* `Array[Int](1)`. *More importantly, the type of* [0] *is also* `Array[Int](1)`. *It is* not `Array[Int{self==0}](1)`, *even though all the elements are all of type* `Int{self==0}`. *This is subtle but important. There are many functions that take* `Array[Int](1)`*s, such as conversions to* `Point`. *These functions do* not *take* `Array[Int{self==0}]`*'s.*

(Suppose that the function took a:`Array[Int](1)` *and did the operation* a(i)=100. *This operation is perfectly fine for an* `Array[Int](1)`, *which is all the compiler knows about* a. *However, it is invalid for an* `Array[Int{self==0}](1)`, *because it assigns a non-zero value to an element of the array, violating the type constraint which says that all the elements are zero. So,* `Array[Int{self==0}](1)` *is not and must not be a subtype of* `Array[Int](1)` — *the two types are simply unrelated. Since there are far more uses for* `Array[Int](1)` *than* `Array[Int{self==0}](1)`, *the compiler produces the former.)*

Example: *Occasionally one does actually need* `Array[Int{self==0}](1)`, *or, say,* `Array[Eel{self != null}](1)`, *an array of non-null* `Eel`*s. For these cases, cast one of the elements of the array to the desired type, and the array constructor will do the right thing.*

```
val zero <: Array[Int{self == 0}](1)
          = [0];
val non1 <: Array[Int{self != 1}](1)
          = [0 as Int{self != 1}];
val eels <: Array[Eel{self != null}](1)
          = [new Eel() as Eel{self != null}, new Eel(), new Eel()];
```

11.27 Coercions and conversions

X10 v2.2 supports the following coercions and conversions.

11.27.1 Coercions

> *CastExp*   ::=   *CastExp* `as` *Type*

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast. A *conversion* may change object identity if the type being converted to is not the same as the type converted from. X10 permits user-defined conversions (§11.27.2).

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

Example: *If* `Child <: Person` *and* `val rhys:Child`, *then* `rhys` *may be used in any context that expects a* `Person`. *For example,*

```
class Example {
  def greet(Person) = "Hi!";
  def example(rhys: Child) {
     greet(rhys);
  }
}
```

Similarly, 2 *(whose innate type is* Int{self==2}*) is usable in a context requiring a non-zero integer (*Int{self != 0}*).*

Explicit Coercion (Casting with as**)**   All classes and interfaces allow the use of the as operator for explicit type coercion. Any class or interface may be cast to any interface. Any interface may be cast to any class. Also, any interface can be cast to a struct that implements (directly or indirectly) that interface.

Example: *In the following code, a* Person *is cast to* Childlike*. There is nothing in the class definition of* Person *that suggests that a* Person *can be* Childlike*. However, the* Person *in question,* p*, is actually a* HappyChild *— a subclass of* Person *— and is, in fact,* Childlike*.*

Similarly, the Childlike *value* cl *is cast to* Happy*. Though these two interfaces are unrelated, the value of* cl *is, in fact,* Happy*. And the* Happy *value* hc *is cast to the class* Child*, though there is no relationship between the two, but the actual value is a* HappyChild*, and thus the cast is correct at runtime.*

Cyborg *is a struct rather than a class. So, it cannot have substructs, and all the interfaces of all* Cyborg*s are known: a* Cyborg *is* Personable*, but not* Childlike *or* Happy*. So, it is correct and meaningful to cast* r *to* Personable*. There is no way that a cast to* Childlike *could succeed, so* r as Childlike *is a static error.*

```
interface Personable {}
class Person implements Personable {}
interface Childlike extends Personable {}
class Child extends Person implements Childlike {}
struct Cyborg implements Personable {}
interface Happy {}
class HappyChild extends Child implements Happy {}
class Example {
  static def example() {
    var p : Person = new HappyChild();
    val cl : Childlike = p as Childlike; // class -> interface
    val hc : Happy = cl as Happy; //        interface -> interface
    val ch : Child = hc as Child; //        interface -> class
    var r : Cyborg = Cyborg();
    val rl : Personable = r as Personable;
    // ERROR: r as Childlike
```

```
        }
    }
```

If the value coerced is not an instance of the target type, and no coercion operators that can convert it to that type are defined, a `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

Limitation: It is currently a static error, rather than the specified `ClassCastException`, when the cast is statically determinable to be impossible.

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

- Casting a number to a *wider* numeric type is safe and effective, and can be done by an implicit conversion as well as an explicit coercion. For example, `4 as Long` produces the `Long` value of 4.

- Casting a floating-point value to an integer value truncates the digits after the decimal point, thereby rounding the number towards zero. `54.321 as Int` is 54, and `-54.321 as Int` is -54. If the floating-point value is too large to represent as that kind of integer, the coercion returns the largest or smallest value of that type instead: `1e110 as Int` is `Int.MAX_VALUE`, *viz.* `2147483647`.

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

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

- Casting an unsigned integer type to a signed integer type of the same size (*e.g.*, `UInt` to `Int`) preserves 2's-complement bit pattern (*e.g.*, `UInt.MAX_VALUE` as `Int == -1`. Casting an unsigned integer type to a signed integer type of a different size is equivalent to first casting to an unsigned integer type of the target size, and then casting to a signed integer type.

- Casting a signed integer type to an unsigned one is similar.

User-defined Coercions

Users may define coercions from arbitrary types into the container type B, and coercions from B to arbitrary types, by providing `static operator` definitions for the `as` operator in the definition of B.

Example:

```
class Bee {
  public static operator (x:Bee) as Int = 1;
  public static operator (x:Int) as Bee = new Bee();
  def example() {
    val b:Bee = 2 as Bee;
    assert (b as Int) == 1;
  }
}
```

11.27.2   Conversions

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

```
Byte < Short < Int < Long < Float < Double
UByte < UShort < UInt < ULong
```

Furthermore, an unsigned integer type may be implicitly coerced a signed type large enough to hold any value of the type: `UByte` to `Short`, `UShort` to `Int`, `UInt` to `Long`. There are no implicit conversions from signed to unsigned numbers, since they cannot treat negatives properly.

There are no implicit conversions in cases when overflow is possible. For example, there is no implicit conversion between `Int` and `UInt`. If it is necessary to convert between these types, use `n as Int` or `n as UInt`, generally with a test to ensure that the value will fit and code to handle the case in which it does not.

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

User defined conversions. The user may define implicit conversion operators from type `A` *to* a container type `B` by specifying an operator in `B`'s definition of the form:

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

The return type `T` should be a subtype of `B`. The return type need not be specified explicitly; it will be computed in the usual fashion if it is not. However, it is good practice for the programmer to specify the return type for such operators explicitly. The return type can be more specific than simply `B`, for cases when there is more information available.

Example: *The code for* `x10.lang.Point` *contains a conversion from one-dimensional* `Array`*s of integers to* `Point`*s of the same length:*

```
public operator (r: Array[Int](1)): Point(r.length) = make(r);
```

This conversion is used whenever an array of integers appears in a context that requires a `Point`*, such as subscripting. Note that* `a` *requires a* `Point` *of rank 2 as a subscript, and that a two-element* `Array` *(like* `[2,4]`*) is converted to a* `Point(2)`.

```
val a = new Array[String]((2..3) * (4..5), "hi!");
a([2,4]) = "converted!";
```

11.28 Parenthesized Expressions

If `E` is any expression, `(E)` is an expression which, when evaluated, produces the same result as `E`.

Example: *The main use of parentheses is to write complex expressions for which the standard precedence order of operations is not appropriate:* `1+2*3` *is 7, but* `(1+2)*3` *is 9.*

Similarly, but perhaps less familiarly, parentheses can disambiguate other expressions. In the following code, `funny.f` *is a field-selection expression, and so* `(funny.f)()` *means "select the* `f` *field from* `funny`*, and evaluate it". However,* `funny.f()` *means "evaluate the* `f` *method on object* `funny`*."*

```
class Funny {
  def f () = 1;
  val f = () => 2;
  static def example() {
    val funny = new Funny();
    assert funny.f() == 1;
    assert (funny.f)() == 2;
  }
}
```

Note that this does *not* mean that `E` and `(E)` are identical in all respects; for example, if `i` is an `Int` variable, `i++` increments `i`, but `(i)++` is not allowed. `++` is an assignment; it operates on variables, not merely values, and `(i)` is simply an expression whose *value* is the same as that of `i`.

12 Statements

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

12.1 Empty statement

The empty statement ; does nothing.

Example: *Sometimes, the syntax of X10 requires a statement in some position, but you do not actually want to do any computation there. The following code searches the array* a *for the value* v, *assumed to appear somewhere in* a, *and returns the index at which it was found. There is no computation to do in the loop body, so we use an empty statement there.*

```
static def search[T](a: Array[T](1), v: T):Int {
  var i : Int;
  for(i = a.region.min(0); a(i) != v; i++)
      ;
  return i;
}
```

12.2   Local variable declaration

LocalVariableDeclStatement	::=	*LocalVariableDecl* ;	*(20.155)*
VariableDeclaratorsWithType	::=	*VariableDeclaratorWithType*	*(20.123)*
		| *VariableDeclaratorsWithType* , *VariableDeclarator-WithType*	
VariableDeclarators	::=	*VariableDeclarator*	*(20.124)*
		| *VariableDeclarators* , *VariableDeclarator*	
VariableInitializer	::=	*Exp*	*(20.125)*
FormalDeclarators	::=	*FormalDeclarator*	*(20.121)*
		| *FormalDeclarators* , *FormalDeclarator*	

Short-lived variables are introduced by local variables declarations, as described in §5. Local variables may be declared only within a block statement (§12.3). The scope of a local variable declaration is the statement itself and the subsequent statements in the block.

```
if (a > 1) {
    val b = a/2;
    var c : Int = 0;
    // b and c are defined here
}
// b and c are not defined here.
```

Variables declared in such statements shadow variables of the same name declared elsewhere. A local variable of a given name, say x, cannot shadow another local variable or parameter named x unless there is an intervening method or closure declaration, or unless the inner x is declared inside an `async` or `at` statement and the outer variable is declared outside of that. Strictly, `at` introduces a new scope that does not share the variables of the external scope, so its variables do not actually shadow those outside.

Example: *The following code illustrates both legal and illegal uses of shadowing. Note that a shadowed* field *name x can still be accessed as* `this.x`.

```
class Shadow{
  var x : Int;
  def this(x:Int) {
     // Parameter can shadow field
     this.x = x;
```

```
    }
    def example(y:Int) {
       val x = "shadows a field";
       // ERROR: val y = "shadows a param";
       val z = "local";
       for (a in [1,2,3]) {
          // ERROR: val x = "can't shadow local var";
       }
       async {
          val x = "can shadow through async";
       }
       at (here;) {
          val x = "at gives a whole new namespace";
       }
       val f = () => {
         val x = "can shadow through closure";
          x
       };
    }
  }
```

12.3 Block statement

Block	::=	{ *BlockStatements*$^?$ }	*(20.147)*
BlockStatements	::=	*BlockStatement*	*(20.148)*
	|	*BlockStatements BlockStatement*	
BlockStatement	::=	*LocalVariableDeclStatement*	*(20.149)*
	|	*ClassDecl*	
	|	*TypeDefDecl*	
	|	*Statement*	

A block statement consists of a sequence of statements delimited by "{" and "}". When a block is evaluated, the statements inside of it are evaluated in order. Blocks are useful for putting several statements in a place where X10 asks for a single one, such as the consequent of an if, and for limiting the scope of local variables.

```
if (b) {
  // This is a block
  val v = 1;
  S1(v);
  S2(v);
}
```

12.4   Expression statement

ExpStatement	::=	*StatementExp* ;	*(20.48)*
StatementExp	::=	*Assignment*	*(20.49)*
		| *PreIncrementExp*	
		| *PreDecrementExp*	
		| *PostIncrementExp*	
		| *PostDecrementExp*	
		| *MethodInvocation*	
		| *ClassInstCreationExp*	

The expression statement evaluates an expression. The value of the expression is not used. Side effects of the expression occur, and may produce results used by following statements. Indeed, statement expressions which terminate without side effects cannot have any visible effect on the results of the computation.

Example:

```
class StmtEx {
  def this() { x10.io.Console.OUT.println("New StmtEx made");  }
  static def call() { x10.io.Console.OUT.println("call!");  }
  def example() {
     var a : Int = 0;
     a = 1; // assignment
     new StmtEx(); // allocation
     call(); // call
  }
}
```

Only selected forms of expression can appear in expression statements: assignments, increments, decrements, method invocations, and constructor invocations.

Most other forms of expression, such as literals and variable and field references, cannot have side effects and are thus pointless as expression statements.

12.5 Labeled statement

   LabeledStatement   ::=   *Id* : *Statement*

Statements may be labeled. The label may be used to describe the target of a `break` statement appearing within a substatement (which, when executed, ends the labeled statement), or, in the case of a loop, a `continue` as well (which, when executed, proceeds to the next iteration of the loop). The scope of a label is the statement labeled.

Example: *The label on the outer* `for` *statement allows* `continue` *and* `break` *statements to continue or break it. Without the label,* `continue` *or* `break` *would only continue or break the inner* `for` *loop.*

```
lbl : for (i in 1..10) {
   for (j in i..10) {
      if (a(i,j) == 0) break lbl;
      if (a(i,j) == 1) continue lbl;
      if (a(i,j) == a(j,i)) break lbl;
   }
}
```

In particular, a block statement may be labeled: `L:{S}`. This allows the use of `break L` within S to leave S, which can, if carefully used, avoid deeply-nested `if`s.

Example:

```
multiphase: {
  if (!exists(filename)) break multiphase;
  phase1(filename);
  if (!suitable_for_phase_2(filename)) break multiphase;
  phase2(filename);
  if (!suitable_for_phase_3(filename)) break multiphase;
  phase3(filename);
}
// Now the file has been phased as much as possible
```

Limitation: Blocks cannot currently be labeled.

12.6   Break statement

BreakStatement	::= break *Id*$^?$;	*(20.64)*

An unlabeled break statement exits the currently enclosing loop or switch state-ment. A labeled break statement exits the enclosing statement with the given label. It is illegal to break out of a statement not defined in the current method, constructor, initializer, or closure.

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

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

12.7   Continue statement

ContinueStatement	::= continue *Id*$^?$;	*(20.65)*

An unlabeled `continue` skips the rest of the current iteration of the innermost en-closing loop, and proceeds on to the next. A labeled `continue` does the same to the enclosing loop with that label. It is illegal to continue a loop not defined in the current method, constructor, initializer, or closure.

12.8   If statement

IfThenStatement	::= if (*Exp*) *Statement*	*(20.43)*
IfThenElseStatement	::= if (*Exp*) *Statement* else *Statement*	*(20.44)*

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

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

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

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

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

could be intended to mean either

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

or

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

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

12.9   Switch statement

SwitchStatement	::=	`switch (` *Exp* `)` *SwitchBlock*	*(20.51)*
SwitchBlock	::=	`{` *SwitchBlockStatementGroups*$^?$ *SwitchLabels*$^?$ `}`	*(20.52)*
SwitchBlockStatementGroups	::=	*SwitchBlockStatementGroup*	*(20.53)*
	|	*SwitchBlockStatementGroups* *SwitchBlockStatementGroup*	
SwitchBlockStatementGroup	::=	*SwitchLabels BlockStatements*	*(20.54)*
SwitchLabels	::=	*SwitchLabel*	*(20.55)*
	|	*SwitchLabels SwitchLabel*	
SwitchLabel	::=	`case` *ConstantExp* `:`	*(20.56)*
	|	`default :`	

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

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

The index expression must be of type `Int`. Case labels must be of type `Int`, `Byte`, `Short`, or `Char` and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

Example: *In this* `switch`*, case* 1 *falls through to case* 2. *The other cases are separated by* `break`*s.*

```
switch (i) {
  case 1: println("one, and ");
  case 2: println("two");
          break;
  case 3: println("three");
          break;
  default: println("Something else");
            break;
}
```

12.10   While statement

WhileStatement   ::=   while (*Exp*) *Statement*   *(20.57)*

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

Example: *A loop to execute the process in the Collatz conjecture (a.k.a. 3n+1 problem, Ulam conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, and Syracuse problem) can be written as follows:*

```
while (n > 1) {
    n = (n % 2 == 1) ? 3*n+1 : n/2;
}
```

12.11   Do–while statement

DoStatement	::=	do *Statement* `while` `(` *Exp* `)` `;`	*(20.58)*

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

12.12   For statement

ForStatement	::=	*BasicForStatement*	*(20.59)*
	|	*EnhancedForStatement*	
BasicForStatement	::=	`for` `(` *ForInit$^?$* `;` *Exp$^?$* `;` *ForUpdate$^?$* `)` *Statement*	*(20.60)*
ForInit	::=	*StatementExpList*	*(20.61)*
	|	*LocalVariableDecl*	
ForUpdate	::=	*StatementExpList*	*(20.62)*
StatementExpList	::=	*StatementExp*	*(20.63)*
	|	*StatementExpList* `,` *StatementExp*	
EnhancedForStatement	::=	`for` `(` *LoopIndex* `in` *Exp* `)` *Statement*	*(20.78)*
	|	`for` `(` *Exp* `)` *Statement*	

`for` statements provide bounded iteration, such as looping over a list. It has two forms: a basic form allowing near-arbitrary iteration, *a la* C, and an enhanced form designed to iterate over a collection.

A basic `for` statement provides for arbitrary iteration in a somewhat more organized fashion than a `while`. The loop `for(init; test; step)body` is equivalent to:

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

`init` is performed before the loop, and is traditionally used to declare and/or initialize the loop variables. It may be a single variable binding statement, such

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

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

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

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

An enhanced for statement is used to iterate over a collection, or other structure designed to support iteration by implementing the interface `Iterable[T]`. The loop variable must be of type `T`, or destructurable from a value of type `T` (§5). Each iteration of the loop binds the iteration variable to another element of the collection. The loop `for(x in c)`S behaves like:

```
val iterator: Iterator[T] = c.iterator();
while (iterator.hasNext()) {
  val x : T = iterator.next();
  S();
}
```

A number of library classes implement `Iterable`, and thus can be iterated over. For example, iterating over a `Region` iterates the `Point`s in the region, and iterating over an `Array` iterates over the `Point`s at which the array is defined.

The type of the loop variable may be supplied as `x <: T`. In this case the iterable `c` must have type `Iterable[U}` for some `U <: T`, and `x` will be given the type `U`.

Example: *This loop adds up the elements of a* `List[Int]`. *Note that iterating over a list yields the elements of the list, as specified in the* `List` *API.*

```
static def sum(a:x10.util.List[Int]):Int {
  var s : Int = 0;
  for(x in a) s += x;
```

```
    return s;
  }
```

The following code sums the elements of an integer array. Note that the `for` *loop iterates over the indices of the array, not the elements, as specified in the* `Array` *API.*

```
static def sum(a: Array[Int]): Int {
  var s : Int = 0;
  for(p in a) s += a(p);
  return s;
}
```

Iteration over an `IntRange` *(§16.2) is quite common. This allows looping while varying an integer index:*

```
var sum : Int = 0;
for(i in 1..10) sum += i;
assert sum == 55;
```

Iteration variables have the `for` statement as scope. They shadow other variables of the same names.

12.13   Return statement

| *ReturnStatement* | ::= | `return` $Exp^?$ `;` | *(20.66)* |

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

Example: *The following code illustrates returning values from a closure and a method. The* `return` *inside of* `closure` *returns from* `closure`, *not from* `method`.

```
def method(x:Int) {
  val closure = (y:Int) => {return x+y;};
  val res = closure(0);
  assert res == x;
  return res == x;
}
```

12.14   Assert statement

AssertStatement	::=	assert *Exp* ;	*(20.50)*
	|	assert *Exp* : *Exp* ;	

The statement `assert E` checks that the Boolean expression E evaluates to true, and, if not, throws an `x10.lang.Error` exception. The annotated assertion `assert E : F;` checks E, and, if it is false, throws an `x10.lang.Error` exception with F's value attached to it.

Example: *The following code compiles properly.*

```
class Example {
  public static def main(argv:Array[String](1)) {
    val a = 1;
    assert a != 1 : "Changed my mind about a";
  }
}
```

However, when run, it prints a stack trace starting with

```
x10.lang.Error: Changed my mind about a
```

12.15   Exceptions in X10

X10 programs can throw *Exceptions* to indicate unusual or problematic situations. Exceptions, as data values, are objects which which inherit from `x10.lang.Throwable`. Exceptions may be thrown intentionally with the `throw` statement. Many primitives and library functions throw exceptions if they encounter problems; *e.g.*, dividing by zero throws an instance of `x10.lang.ArithmeticException`.

When an exception is thrown, statically and dynamically enclosing `try-catch` blocks in the same activity can attempt to handle it. If the throwing statement in inside some `try` clause, and some matching `catch` clause catches that type of exception, the corresponding `catch` body will be executed, and the process of throwing is finished. If no statically-enclosing `try-catch` block can handle the exception, the current method call returns (abnormally), throwing the same exception from the point at which the method was called.

This process continues until the exception is handled or there are no more calling methods in the activity. In the latter case, the activity will terminate abnormally, and the exception will propagate to the activity's root; see §14.1 for details.

Unlike some statically-typed languages with exceptions, X10's exceptions are all *unchecked*. Methods do not declare which exceptions they might throw; any method can, potentially, throw any exception.

12.16 Throw statement

ThrowStatement	::=	throw *Exp* ;	*(20.67)*

`throw E` throws an exception whose value is E, which must be an instance of a subtype of `x10.lang.Throwable`.

Example: *The following code checks if an index is in range and throws an exception if not.*

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

12.17 Try–catch statement

TryStatement	::=	try *Block Catches*	*(20.68)*
	|	try *Block Catches*$^?$ *Finally*	
Catches	::=	*CatchClause*	*(20.69)*
	|	*Catches CatchClause*	
CatchClause	::=	catch (*FormalParam*) *Block*	*(20.70)*
Finally	::=	finally *Block*	*(20.71)*

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

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

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

The parameter of a `catch` block has the block as scope. It shadows other variables of the same name.

Example: *The* `example()` *method below executes without any assertion errors*

```
class Exn extends Throwable{}
class SubExn(n:Int) extends Exn{}
class Example {
  static def example() {
    var correct : Boolean = false;
    try {
        throw new SubExn(4);
    }
    catch (e : Exn)          { correct = true; }
    catch (e : SubExn)       { assert false; }
    catch (e : Throwable)    { assert false; }
    assert correct;
  }
}
```

Limitation: Constraints on exception types in `catch` blocks are not currently supported.

12.18   Assert

The `assert` statement `assert B;` checks that the Boolean expression B evaluates to true. If so, computation proceeds. If not, it throws `x10.lang.AssertionError`.

The extended form `assert B:A;` is similar, but provides more debugging information. The value of the expression A is available as part of the `AssertionError`, *e.g.*, to be printed on the console.

Example: `assert` *is useful for confirming properties that you believe to be true and wish to rely on. In particular, well-chosen* `assert`*s make a program robust in the face of code changes and unexpected uses of methods. For example, the following method compute percent differences, but asserts that it is not dividing by zero. If the mean is zero, it throws an exception, including the values of the numbers as potentially useful debugging information.*

```
static def percentDiff(x:Double, y:Double) {
  val diff = x-y;
  val mean = (x+y)/2;
  assert mean != 0.0  : [x,y];
  return Math.abs(100 * (diff / mean));
}
```

At times it may be considered important not to check `assert` statements; *e.g.*, if the test is expensive and the code is sufficiently well-tested. The `-noassert` command line option causes the compiler to ignore all `assert` statements.

13   Places

An X10 place is a repository for data and activities, corresponding loosely to a process or a processor. Places induce a concept of "local". The activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote places may take orders of magnitude longer. X10's system of places is designed to make this obvious. Programmers are aware of the places of their data, and know when they are incurring communication costs, but the actual operation to do so is easy. It's not hard to use non-local data; it's simply hard to to do so accidentally.

The set of places available to a computation is determined at the time that the program is started, and remains fixed through the run of the program. See the `README` documentation on how to set command line and configuration options to set the number of places.

Places are first-class values in X10, as instances of the built-in struct, `x10.lang.Place`. `Place` provides a number of useful ways to query places, such as `Place.places`, a `Sequence[Place]` of the places available to the current run of the program.

Objects and structs (with one exception) are created in a single place – the place that the constructor call was running in. They cannot change places. They can be *copied* to other places, and the special library struct `GlobalRef` allows values at one place to point to values at another.

13.1   The Structure of Places

Places are numbered 0 through `Place.MAX_PLACES-1`; the number is stored in the field `pl.id`. The `Sequence[Place]` `Place.places()` contains the places of the program, in numeric order. The program starts by executing a `main` method at `Place.FIRST_PLACE`, which is `Place.places()(0)`; see §14.4.

Operations on places include `pl.next()`, which gives the next entry (looping around) in `Place.places` and its opposite `pl.prev()`. In particular, `here.next()` means "a place other than `here`", except in single-place executions. There are also a number of tests, like `pl.isSPE()` and `pl.isCUDA()`, which test for particular kinds of processors.

13.2 `here`

The variable `here` is always bound to the place at which the current computation is running, in the same way that `this` is always bound to the instance of the current class (for non-static code), or `self` is bound to the instance of the type currently being constrained. `here` may denote different places in the same method body or even the same expression, due to place-shifting operations.

This is not unusual for automatic variables: `self` denotes two different values (one `List`, one `Int`) in `List[Int{self!=0}]{self!=null}`. In the following code, `here` has one value at `h0`, and a different one at `h1` (unless there is only one place).

```
val h0 = here;
at (here.next()) {
  val h1 = here;
  assert (h0 != h1);
}
```

(Similar examples show that `self` and `this` have the same behavior: `self` can be modified by constrained types appearing inside of type constraints, and `this` by inner classes.)

The following example looks through a list of references to `Thing`s. It finds those references to things that are `here`, and deals with them.

```
public static def deal(things: List[GlobalRef[Thing]]) {
    for(gr in things) {
        if (gr.home == here) {
            val grHere =
                gr as GlobalRef[Thing]{gr.home == here};
            val thing <: Thing = grHere();
            dealWith(thing);
```

```
      }
    }
  }
```

13.3   `at`: Place Changing

An activity may change place synchronously using the `at` statement or `at` expression. This is an expensive operation and must be used with care – but it provides the basis for multicore programming in X10.

The *PlaceExp* must be an expression of type `Place` or some subtype.

The `at`-statment `at(p;F)S` first evaluates `p` to a place, then copies information to that place as determined by `F`, and then executes `S` using the resulting copies. The `at`-*expression* `at(p;F)E` is similar, but it copies the result of the expression `E` and returns the copy as its result.

The clause `F` in `at(p;F)S` is a list of zero or more *copy specifiers*, explaining what values are to be copied to the place `p`, and how they are to be referred to at `p`.

Example: *The following example creates an array* a *located* here, *and copies it to another place, giving the copy the name* a2 *there. The copy is modified and examined. After the* at *finishes, the original is also examined, and (since only the copy, not the original, was modified) is observed to be unchanged.*

```
val a = [1,2,3];
at(here.next(); a2 = a) {
  a2(1) = 4;
  assert a2.equals([1,4,3]);
  // 'a' is not accessible here
}
assert a.equals([1,2,3]);
```


13.3.1   Copy Specifiers

A single copy specifier can be one of the following forms. Each copy specifier determines an *original-expression*, saying what value will be copied, and a *target variable*, saying what it will be called.

- `val x = E`, and its usual variants `val x:T = E`, `x : T = E`, and `val x <: T = E`, evaluate the expression `E` at the initial place, copy it to `p`, and bind `x` to the copy, as normal for a local `val` binding. If a type is supplied, it is checked statically in the usual way. The original-expression is `E`, and the target variable is `x`.

- `var x : T = E` evaluates `E` at the initial place, copies it to `p`, and binds `x` to a new `var` whose initial value is the copy, as normal for a local `var` binding. If a type is supplied, it is checked statically in the usual way. The original-expression is `E`, and the target variable is `x`. Note that, like a `var` parameter to a method, `x` is a local variable. Changes to `x` will not change anything else. In particular, even if `x` has the same name as a `var` variable outside, the two `var`s are unconnected. See §13.4 for the way to modify a variable from the surrounding scope.

- `x = E`, as a copy specifier, is equivalent to `val x = E`. Note that this abbreviated form is not available as a local variable definition, (because it is used as an assignment statement), but in a copy specifier there are no assignment statements. The original-expression is `E`, and the target variable is `x`.

- `x` alone, as a copy specifier, is equivalent to `val x = x`. It says that the variable `x` will be copied, and the copy will also be named `x`. The original-expression is `x`, and the target variable is `x`.

- A field assignment statements E_1`.fld` $= E_2$, evaluates E_1 and E_2 on the sending side to values v_1 and v_2. v_1 must be an object with a mutable field `fld`. v_1 and v_2 are sent to place `p`, and the field assignment is performed there. The compiler may optimize this, *e.g.*, by neglecting to deserialize v_1 `.fld`, and deserializing v_2 directly into that field rather than into a separate buffer.

- An array-element assignment E_0`(`E_1`, ..., `E_n`)` $= E_+$. Much like a field assignment, this evaluates all the expressions E_i at the sending side to values v_i, and transmits them. v_0 must admit a suitably-typed n-ary subscripting operation. That operation is applied after the values are deserialized at `p`. The compiler may optimize this, *e.g.*, by neglecting to deserialize one element of the array v_0, and deserializing v_+ directly into that location.

- `*` may appear as the last copy specifier in the list, indicating that all `val` variables from outside S which are used in S should be copied. Specifically, let $x_1, \ldots, x_n$ be all the `val` variables defined outside of S mentioned in S. The `*` copy specifier is equivalent to the list of variables $x_1, \ldots, x_n$.

As an important special case, `at(p;)S` copies *nothing* to S. This must not be confused with `at(p)S`, which copies *everything*.

Note that `at(p;x,*)use(x,y);` is equivalent to `at(p;*)use(x,y);`. In both statements, the `*` indicates that all variables used in the body are to be copied in. The former makes clear that `x` is one of the things being copied, but, from the `*`, there may be others.

However, other copy specifiers may be used to compute values in S which are not available (and thus need not be stored) outside of it. The following code may end up with a large object `c` in memory at `p` but not at the initial place:

```
at(p; c = a.largeFun(b), *) {
  use(a,b,c);
}
```

The blanket `at`-statement `at(p)S` copies everything. It is an abbreviation for `at(p;*)S`. When this manual refers to a generic `at`-statement as `at(p;F)S`, it should be understood as including the blanket `at` statement `at(p)S` with this interpretation.

13.3.2   Copying Values

An activity executing statement `at (q;F) S` at a place `p` evaluates `q` at `p` and then moves to `q` to execute S. The original-expressions of F are evaluated at `p`. Their values are copied (§13.3.3) to `q`, and bound to names there, as specified by F. S is evaluated in an environment containing the target variables of F, and `here` and *no* other variables. (In particular, if this statement appears in an instance method body and `this` is not copied, `this` is not accessible. This fact is important: it allows the programmer to control when `this` is copied, which may be expensive for large containers.)

Note that the value obtained by evaluating `q` is not necessarily distinct from `p` (*e.g.*, `q` may be `here`). This does not alter the behavior of `at`. `at(here;F)S` will

copy all the values specified by F, even though there is no actual change of place, and even though the original values already exist there.

On normal termination of S control returns to p and execution is continued with the statement following at (q;F) S. If S terminates abruptly with exception E, E is serialized into a buffer, the buffer is communicated to p where it is deserialized into an exception E1 and at (p) S throws E1.

Since at(p) S is a synchronous construct, usual control-flow constructs such as break, continue, return and throw are permitted in S. All concurrency related constructs – async, finish, atomic, when are also permitted.

The at-expression at(p)E is similar, except that, in the case of normal termination of E, the value that E produces is serialized into a buffer, transported to the starting place, and deserialized, and the value of the at-expression is the result of deserialization.

Limitation: X10 does not currently allow break, continue, or return to exit from an at.

13.3.3 How at Copies Values

The values of the original-expressions specified by F in at (p;F)S are copied to p, as follows.

First, the original-expressions are evaluated to give a vector of X10 values. Consider the graph of all values reachable from these values (without following transient fields (§13.3.6, GlobalRefs (§13.3.7); also custom serialization (§13.3.3 may alter this behavior)).

Second this graph is *serialized* into a buffer and transmitted to place q. Third, the vector of X10 values is re-created at q by deserializing the buffer at q. Fourth, S is executed at q, in an environment in which each variable v declared in F refers to the corresponding deserialized value.

Note that since values accessed across an at boundary are copied, the programmer may wish to adopt the discipline that either variables accessed across an at boundary contain only structs or stateless objects, or the methods invoked on them do not access any mutable state on the objects. Otherwise the programmer has to ensure that side effects are made to the correct copy of the object. For this the struct x10.lang.GlobalRef[T] is often useful.

Serialization and deserialization.

The X10 runtime provides a default mechanism for serializing/deserializing an object graph with a given set of roots. This mechanism may be overridden by the programmer on a per class or struct basiss as described in `x10.io.CustomSerialization`. The default mechanism performs a deep copy of the object graph (that is, it copies the object or struct and, recursively, the values contained in its fields), but does not traverse `transient` fields. Instead on serialization the default value §4.5 for the type of the field is used.

A struct `s` of type `x10.lang.GlobalRef[T]` 13.3.7 is serialized as a unique global reference to its contained object `o` (of type T). Please see the documentation of `x10.lang.GlobalRef[T]` for more details.

13.3.4 `at` and Activities

`at(p;F)S` does *not* start a new activity. It should be thought of as transporting the current activity to `p`, running `S` there, and then transporting it back. `async` is the only construct in the language that starts a new activity. In different contexts, each one of the following makes sense: (1) `async at(p;F) S` (spawn an activity locally to execute S at `p`; here `p` is evaluated by the spawned activity) , (2) `at(p;F) async S` (evaluate p and then at p spawn an activity to execute S), and, (3) `async at(p;F) async S`. In most cases, `async at(p;F) S` is preferred to `at(p;F) async S`, since the former returns instantly, but the latter blocks waiting for the remote activity to be spawned.

Since `at(p;F) S` does not start a new activity, S may contain constructs which only make sense within a single activity. For example,

```
for(x in globalRefsToThings)
  if (at(x.home; x) x().isNice())
    return x();
```

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

Limitation: X10 does not currently allow `break`, `continue`, or `return` to exit from an `at`.

13.3.5 Implicit copying from `at`

`at(p;F)S` copies data as specified by `F`, and sends it to place `p`, before executing
`S` there. The only things that are not copied are values only reachable through
`GlobalRef`s and `transient` fields, and data omitted by custom serialization.
If `x` is a variable name referring to some object `ob` outside of `S`, then the same
variable name `x` refers to a deep copy of `ob` inside of `S`. In this way the body `S` is
in a separate block with different variable bindings.

For example, consider the following program.

```
val c = new Cell[Int](9); // (1)
at (here;c) {              // (2)
   assert(c() == 9);       // (3)
   c.set(8);               // (4)
   assert(c() == 8);       // (5)
}
assert(c() == 9);          // (6)
```

The `at` statement copies the `Cell` and its contents. After (1), `c` is a `Cell` con-
taining 9; call that cell c_1 At (2), that cell is copied, resulting in another cell c_2
whose contents are also 9, as tested at (3). (Note that the copying behavior of `at`
happens *even when the destination place is the same as the starting place*— even
with `at(here)`.) At (4), the contents of c_2 are changed to 8, as confirmed at (5);
the contents of c_1 are of course untouched. Finally, at (c), outside the scope of
the `at` started at line (2), `c` refers to its original value c_1 rather than the copy c_2.

The `at` statement induces a *deep copy*. Not only does it copy the values of vari-
ables, it copies values that they refer to through zero or more levels of reference.
Structures are preserved as well: if two fields `x.f` and `x.g` refer to the same object
o_1 in the original, then `x.f` and `x.g` will both refer to the same object o_2 in the
copy.

For example, in the following variation of the preceding example, `a`'s original
value a_1 is an array with two references to the same `Cell[Int]` c_1. The fact that
$a_1(0)$ and $a_1(1)$ are both identical to c_1 is demonstrated in (A)-(C), as $a_1(0)$ is
modified and $a_1(1)$ is observed to change. In (D)-(F), the copy a_2 is tested in
the same way, showing that $a_2(0)$ and $a_2(1)$ both refer to the same `Cell[Int]` c_2.
However, the test at (G) shows that c_2 is a different cell from c_1, because changes
to c_2 did not propagate to c_1.

```
val c = new Cell[Int](5);
```

```
val a = new Array[c,c];
assert(a(0)() == 5 && a(1)() == 5);      // (A)
c.set(6);                                // (B)
assert(a(0)() == 6 && a(1)() == 6);      // (C)
at(here;a) {
  assert(a(0)() == 6 && a(1)() == 6);    // (D)
  c.set(7);                              // (E)
  assert(a(0)() == 7 && a(1)() == 7);    // (F)
}
assert(a(0)() == 6 && a(1)() == 6);      // (G)
```

13.3.6   Copying and Transient Fields

Recall that fields of classes and structs marked `transient` are not copied by `at`. Instead, they are set to the default values for their types. Types that do not have default values cannot be used in `transient` fields.

In the following example, every `Trans` object has an a-field equal to 1. However, despite the initializer on the b field, it is not the case that every `Trans` has b==2. Since b is `transient`, when the `Trans` value `this` is copied at `at(here){...}` in `example()`, its b field is not copied, and the default value for an `Int`, 0, is used instead. Note that we could not make a transient field `c : Int{c != 0}`, since the type has no default value, and copying would in fact set it to zero.

```
class Trans {
   val a : Int = 1;
   transient val b : Int = 2;
   //ERROR transient val c : Int{c != 0} = 3;
   def example() {
     assert(a == 1 && b == 2);
     at(here;a) {
         assert(a == 1 && b == 0);
     }
   }
}
```

13.3.7 Copying and GlobalRef

A `GlobalRef[T]` (say g) contains a reference to a value of type T, in a form which can be transmitted, and a `Place g.home` indicating where the value lives. When a `GlobalRef` is serialized an opaque, globally unique handle to the enclosed value o is passed.

The following example does not copy the value `huge`. However, `huge` would have been copied if it had been put into a `Cell`, or simply used directly.

```
val huge = "A potentially big thing";
val href = GlobalRef(huge);
at (here;href) {
  use(href);
  }
}
```

Values protected in `GlobalRef`s can be retrieved by the application operation `g()`. `g()` is guarded; it can only be called when `g.home == here`. If you want to do anything other than pass a global reference around or compare two of them for equality, you need to placeshift back to the home place of the reference, often with `at(g.home)`. For example, the following program modifies the command-line argument array.

```
public static def main(argv: Array[String](1)) {
  val argref = GlobalRef[Array[String](1)](argv);
  at(here.next(); argref)
      use(argref);
}
static def use(argref : GlobalRef[Array[String](1)]) {
  at(argref.home; argref) {
    val argv = argref();
    argv(0) = "Hi!";
  }
}
```

13.3.8 Warnings about at

There are two dangers involved with `at`:

- Careless use of `at` can result in copying and transmission of very large data structures. This is particularly an issue with the blanket `at` statement, `at(p)S`, where everything used in S is copied. In particular, it is very easy to capture `this` – a field reference will do it – and accidentally copy everything that `this` refers to, which can be very large. A disciplined use of copy specifiers to make explicit just what gets copied can ameliorate this issue.

- As seen in the examples above, a local variable reference `x` may refer to different objects in different nested `at` scopes. The programmer must either ensure that a variable accessed across an `at` boundary has no mutable state or be prepared to reason about which copy gets modified. A disciplined use of copy specifiers to give different names to variables can ameliorate this concern.

13.4 `athome`: **Returning Values from** `at`**-Blocks**

The `at(p;F)S` construct renders external variables unavailable within S. However, it is often useful to transmit values back from S, and store them in external variables.

The `athome(V;F)S` construct provides this ability. V is a list of variables, which must all be defined at the same place. `athome(V;F)S` goes to the place where the variables are defined, copying F as for `at(p;F)S`, and executes S — allowing reading, assignment and initialization of the listed variables in V.

V, the list of variables, may include one or more variables. It is a static error if X10 cannot determine that all the variables in the list are defined at the same place.

Example: `athome` *allows returning multiple pieces of information from an* `at`*-statement. In the following example, we return two data: one as a* `val` *named* `square`*, and the other as an addition in to a partially-computed polynomial named* `poly`*.*

```
static def example(a: Int, mathProc: Place) {
  val square : Int;
  var poly : Int = 1 + a; // will be 1+a+a*a
  at(mathProc; a) {
    val sq = a*a;
    athome(square, poly; sq) {
```

```
        square = sq;  // initialization
        poly += sq;   // read and update
    }
  return [square, poly];
  }
```

The abbreviated form `athome S` is equivalent to `athome(V;F)S`, where:

- `V` is the list of all variables appearing on the left-hand side of an assignment or update statement in `S`, excluding those which appear inside an `at` or `athome` statement in `S`;

- `F` is the same as for `at(p)S` (§13.3.1)

Example:

Much as the blanket `at` *construct* `at(p)S` *is convenient for executing a small code body at another place, the blanket* `athome` *construct* `athome S` *is convenient for returning a result or two. The preceding example could have been written using blanket statements.*

```
  static def example(a: Int, mathProc: Place) {
    val square : Int;
    var poly : Int = 1 + a; // will be 1+a+a*a
    at(mathProc) {
      val sq = a*a;
      athome {
          square = sq;  // initialization
          poly += sq;   // read and update
      }
    return [square, poly];
    }
```

Design: It is not fundamentally essential to distinguish `at` from `athome`. `at(p;F)S` could allow writing to variables whose homes are known at compile-time to be equal to `p`. Indeed, in earlier versions of X10, it did so. This required an idiom in which programmers had to manage the home locations of variables directly, and keep track of which home location corresponded to which variable. The `athome` construct makes this idiom more convenient.

14 Activities

An *activity* is a statement being executed, independently, with its own local variables; it may be thought of as a very light-weight thread. An X10 computation may have many concurrent activities executing at any give time. All X10 code runs as part of an activity; when an X10 program is started, the `main` method is invoked in an activity, called the *root activity*.

Activities coordinate their execution by various control and data structures. For example, `when(x==0);` blocks the current activity until some other activity sets `x` to zero. However, activities determine the places at which they may be blocked and resumed, by `when` and similar constructs. There are no means by which one activity can arbitrarily interrupt, block, or resume another, no method `activity.interrupt()`.

An activity may be *running*, *blocked* on some condition or *terminated*. If it is terminated, it is terminated in the same way that its statement is: in particular, if the statement terminates abruptly, the activity terminates abruptly for the same reason. (§14.1).

Activities can be long-running entities with a good deal of local state. In particular they can involve recursive method calls (and therefore have runtime stacks). However, activities can also be short-running light-weight entities, *e.g.*, it is reasonable to have an activity that simply increments a variable.

An activity may asynchronously and in parallel launch activities at other places. Every activity except the initial `main` activity is spawned by another. Thus, at any instant, the activities in a program form a tree.

X10 uses this tree in crucial ways. First is the distinction between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate locally when the activity has finished all its computation. (For instance the creation of an asynchronous activity terminates locally when the activity has been created.) It is said to terminate globally when it has

terminated locally and all activities that it may have spawned at any place have, recursively, terminated globally. For example, consider:

```
async {s1();}
async {s2();}
```

The primary activity spawns two child activities and then terminates locally, very quickly. The child activities may take arbitrary amounts of time to terminate (and may spawn grandchildren). When `s1()`, `s2()`, and all their descendants terminate locally, then the primary activity terminates globally.

The program as a whole terminates when the root activity terminates globally. In particular, X10 does not permit the creation of daemon threads—threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§14.4).

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

14.1 The X10 rooted exception model

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

The presence of a root activity and the concept of global termination permits X10 to adopt a more powerful exception model. In any state of the computation, say that an activity A is *a root of* an activity B if A is an ancestor of B and A is blocked at a statement (such as the `finish` statement §14.3) awaiting the termination of B (and possibly other activities). For every X10 computation, the *root-of* relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

---

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

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the *root-of* tree when the activity terminates (based on the semantics of the statement being executed by the activity).[2] There is always a good place to put a `try-catch` block to catch exceptions thrown by an asynchronous activity.

14.2   `async`: **Spawning an activity**

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

An activity is created by executing the `async` statement:

AsyncStatement	::=	`async` *ClockedClause*[?] *Statement*	*(20.73)*
	|	`clocked async` *Statement*	
ClockedClause	::=	`clocked (` *ClockList* `)`	*(20.72)*

The basic form of `async` is `async S`, which starts a new activity located `here` executing S. (For the clocked form, see §15.4.)

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the set of activities at the target place and will be executed based on the local scheduler's decisions. If some particular sequencing of events is needed, `when`, `atomic`, `finish`, clocks, and other X10 constructs can be used. X10 implementations are not required to have fair schedulers, though every implementation should make a best faith effort to ensure that every activity eventually gets a chance to make forward progress.

The statement in the body of an `async` is subject to the restriction that it must be acceptable as the body of a `void` method for an anonymous inner class declared at that point in the code. For example, it may reference `val` variables in lexically enclosing scopes, but not `var` variables.

---

[2]In X10 v2.2 the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

14.3   Finish

The statement `finish` S converts global termination to local termination.

| *FinishStatement* | ::= | `finish` *Statement* | *(20.79)* |
| | | | `clocked finish` *Statement* | |

An activity A executes `finish` S by executing S and then waiting for all activities spawned by S (directly or indirectly, here or at other places) to terminate. An activity may terminate normally, or abruptly, i.e. by throwing an exception. All exceptions thrown by spawned activities are caught and accumulated.

`finish` S terminates locally when all activities spawned by S terminate globally (either abruptly or normally). If S terminates normally, then `finish` S terminates normally and A continues execution with the next statement after `finish` S. If S or one of the activities spawned by it terminate abruptly, then `finish` S terminates abruptly and throws a single exception, `x10.lang.MultipleExceptions` formed from the collection of exceptions accumulated at `finish` S.

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

Note that repeatedly `finish`ing a statement has little effect after the first `finish`: `finish finish` S is indistinguishable from `finish` S if S throws no exceptions. (If S throws exceptions, `finish` S wraps them in one layer of `MultipleExceptions` and `finish finish` S in two layers.)

14.4   Initial activity

An X10 computation is initiated from the command line on the presentation of a class or struct name C. The class or struct must have a `public static def main(a: Array[String](1)):void` method, or a `public static def main(a: Array[String]):void` method, otherwise an exception is thrown and the computation terminates. The single statement

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

is executed where s is a one-dimensional `Array` of strings created from the command line arguments. This single activity is the root activity for the entire computation. (See §13 for a discussion of places.)

14.5   Ateach statements

Deprecated: The `ateach` construct is deprecated.

AtEachStatement	::=	`ateach` (*LoopIndex* `in` *Exp*) *ClockedClause*$^?$	*(20.77)*
		Statement	
	|	`ateach` (*Exp*) *Statement*	
LoopIndexDeclarator	::=	*Id HasResultType*$^?$	*(20.129)*
	|	[*IdList*] *HasResultType*$^?$	
	|	*Id* [*IdList*] *HasResultType*$^?$	
LoopIndex	::=	*Mods*$^?$ *LoopIndexDeclarator*	*(20.130)*
	|	*Mods*$^?$ *VarKeyword LoopIndexDeclarator*	

In `ateach(p in D)` S, D must be either of type `Dist` (see §16.5) or of type `DistArray[T]` (see §16), and p will be of type `Point` (see §16.1). If D is an `DistArray[T]`, then `ateach (p in D)S` is identical to `ateach(p in D.dist)S`; the iteration is over the array's underlying distribution.

Instead of writing `ateach (p in D)` S the programmer should write `for(p in D) at(D(p);F) async` S to get the same effect. For each point p in D, at place D(p), transmitting information as specified by F, S is executed simultaneously.

However, this often results in excessive communication and parallelism. Instead the programmer may want to write:

```
for (place in D.places()) async at (place; p, D, *) {
    for (p in D|here) {
        S(p);
    }
}
```

If the programmer wishes to execute S in parallel at each place, S(p) may be replaced by `async S(p)` .

14.6 Atomic blocks

X10's `atomic` blocks provide a high-level construct for coordinating the mutation of shared data. A programmer may use atomic blocks to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple activities running in the same place.

An X10 program in which all accesses (both reads and writes) of shared variables appear in `atomic` or `when` blocks is guaranteed to use all shared variables atomically. Equivalently, if two accesses to some shared variable `v` could collide at runtime, and one is in an atomic block, then the other must be in an atomic block as well to guarantee atomicity of the accesses to `v`. If some accesses to shared variables are not protected by `atomic` or `when`, then race conditions or deadlocks may occur.

In particular, atomic sections are atomic with respect to each other. They may not be atomic with respect to non-atomic code.

X10 guarantees that atomic sections at the same place are mutually exclusive. That is, if one activity A at a given place p is executing an atomic section, then no other activity B at p will also be executing an atomic section. If such a B attempts to execute an `atomic` or `when` command, it will be blocked until A finishes executing its atomic section.

| *AtomicStatement* | ::= | `atomic` *Statement* | *(20.75)* |
| *WhenStatement* | ::= | `when` (*Exp*) *Statement* | *(20.76)* |

For example, consider a class `Redund[T]`, which encapsulates a list `list` and, (redundantly) keeps the size of the list in a second field `size`. Then `r:Redund[T]` has the invariant `r.list.size() == r.size`, which must be true at any point at which no method calls on `r` are active.

If the `add` method on `Redund` (which adds an element to the list) were defined as:

```
def add(x:T) { // Incorrect
  this.list.add(x);
  this.size = this.size + 1;
}
```

Then two activities simultaneously adding elements to the same `r` could break the invariant. Suppose that `r` starts out empty. Let the first activity perform the `list.add`, and compute `this.size+1`, which is 1, but not store it back into

`this.size` yet. (At this point, `r.list.size()==1` and `r.size==0`; the invariant expression is false, but, as the first call to `r.add()` is active, the invariant does not need to be true – it only needs to be true when the call finishes.) Now, let the second activity do its call to `add` to completion, which finishes with `r.size==1`. (As before, the invariant expression is false, but a call to `r.add()` is still active, so the invariant need not be true.) Finally, let the first activity finish, which assigns the 1 computed before back into `this.size`. At the end, there are two elements in `r.list`, but `r.size==1`. Since there are no calls to `r.add()` active, the invariant must be true, but it is not.

In this case, the invariant can be maintained by making the increment atomic. Doing so forbids that sequence of events; the `atomic` block cannot be stopped partway.

```
def add(x:T) {
  atomic {
    this.list.add(x);
    this.size = this.size + 1;
  }
}
```

14.6.1   Unconditional atomic blocks

The simplest form of an atomic block is the *unconditional atomic block*: `atomic S`. When `atomic S` is executing at some place `p`, no other activity at `p` may enter an atomic block. So, other activities may continue, even at the same place, but code protected by atomic blocks is not subject to interference from other code in atomic blocks.

If execution of the statement may throw an exception, it is the programmer's responsibility to wrap the atomic block within a `try/finally` clause and include undo code in the finally clause. Thus the `atomic` statement only guarantees atomicity on successful execution, not on a faulty execution.

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

Code executed inside of `atomic S` and `when S` is subject to certain restrictions. A violation of these restrictions causes an `IllegalOperationException` to be thrown at the point of the violation.

- S may not spawn another activity.

- S may not use any blocking statements; `when`, `next`, `finish`. (The use of a nested `atomic` is permitted.)

- S may not `force()` a `Future`.

- S may not use `at` expressions.

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

$$\text{atomic } \{s1; \text{ atomic } s2\} \ = \ \text{atomic } \{s1; \ s2\} \qquad (14.1)$$

Atomic blocks do not introduce deadlocks. They may exhibit all the bad behavior of sequential programs, including throwing exceptions and running forever, but they are guaranteed not to deadlock.

Example

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

```
var target:Object = null;
public atomic def CAS(old1: Object, new1: Object): Boolean {
   if (target.equals(old1)) {
      target = new1;
      return true;
   }
   return false;
}
```

14.6.2   Conditional atomic blocks

Conditional atomic blocks allow the activity to wait for some condition to be satisfied before executing an atomic block. For example, consider a `Redund` class holding a list `r.list` and, redundantly, its length `r.size`. A `pop` operation will delay until the `Redund` is nonempty, and then remove an element and update the length.

```
def pop():T {
  var ret : T;
  when(size>0) {
    ret = list.removeAt(0);
    size --;
    }
  return ret;
}
```

The execution of the test is atomic with the execution of the block. This is important; it means that no other activity can sneak in and make the condition be false before the block is executed. In this example, two `pop`s executing on a list with one element would work properly. Without the conditional atomic block – even doing the decrement atomically – one call to `pop` could pass the `size>0` guard; then the other call could run to completion (removing the only element of the list); then, when the first call proceeds, its `removeAt` will fail.

Note that `if` would not work here. `if(size>0) atomic{size--; return list.removeAt(0);}` allows another activity to act between the test and the atomic block. And `atomic{ if(size>0) {size--; ret = list.removeAt(0);}}` does not wait for `size>0` to become true.

Conditional atomic blocks are of the form `when(b)S`; `b` is called the *guard*, and `S` the *body*.

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

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

The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends.

The body `S` of `when(b)S` is subject to the same restrictions that the body of `atomic S` is. The guard is subject to the same restrictions as well. Furthermore, guards should not have side effects.

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

Example 14.6.1 The following class shows how to implement a bounded buffer of size 1 in X10 for repeated communication between a sender and a receiver. The call `buf.send(ob)` waits until the buffer has space, and then puts `ob` into it. Dually, `buf.receive()` waits until the buffer has something in it, and then returns that thing.

```
class OneBuffer[T] {
  var datum: T;
  def this(t:T) { this.datum = t; this.filled = true; }
  var filled: Boolean;
  public def send(v: T) {
    when (!filled) {
      this.datum = v;
      this.filled = true;
    }
  }
  public def receive(): T {
    when (filled) {
      v: T = datum;
      filled = false;
      return v;
    }
  }
}
```

□

14.7   Use of Atomic Blocks

The semantics of atomicity is chosen as a compromise between programming simplicity and efficient implementation. Unlike some possible definitions of "atomic", atomic blocks do not provide absolute atomicity.

Atomic blocks are atomic with respect to *each other*.

```
var n : Int = 0;
finish {
  async atomic n = n + 1; //(a)
  async atomic n = n + 2; //(b)
}
```

This program has only two possible interleavings: either (a) entirely precedes (b) or (b) entirely precedes (a). Both end up with n==3.

However, atomic blocks are not atomic with respect to non-atomic code. It we remove the atomics on (a), we get far messier semantics.

```
var n : Int = 0;
finish {
  // LEGAL BUT UNWISE
  async n = n + 1;            //(a)
  async atomic n = n + 2;    //(b)
}
```

If X10 had absolute atomic semantics, this program would be guaranteed to treat the atomic increment as a single statement. This would permit three interleavings: the two possible from the fully atomic program, or a third one with the events: (a)'s read of 0 from n, the entirety of (b), and then (a)'s write of 0+1 back to n. This interleaving results in n==1. So, with absolute atomic semantics, n==1 or n==3 are the possible results.

However, X10's semantics are weaker than that. Atomic statements are atomic with respect to each other — but there is no guarantee about how they interact with non-atomic statements at all. In particular, the following fourth interleaving is possible: (a)'s read of 0 from n, (b)'s read of 0 from n, (a)'s write of 1 to n, and (b)'s write of 2 to n. Thus, n==2 is permissible as a result in X10.

X10's semantics permit more efficient implementation than absolute atomicity. Absolute atomicity would, in principle, require all activities at place p to stop whenever one of them enters an atomic section, which would seriously curtail concurrency. X10 simply requires that, when one activity is in an atomic section, that other activities stop *when they are trying to enter an atomic section* — which is to say, they can continue computing on their own all they like. The difference can be substantial.

However, X10's semantics do impose a certain burden on the programmer. A sufficient rule of thumb is that, if *any* access to a variable is done in an atomic

section, then *all* accesses to it must be in atomic sections.

15 Clocks

Many concurrent algorithms proceed in phases: in phase k, several activities work independently, but synchronize together before proceeding on to phase $k + 1$. X10 supports this communication structure (and many variations on it) with a generalization of barriers called *clocks*. Clocks are designed so that programs which follow a simple syntactic discipline will not have either deadlocks or race conditions.

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

The program creates a clock `cl` to manage the phases. Each participating activity does the work of its first phase, and then executes `Clock.advanceAll();` to signal that it is finished with that work. `Clock.advanceAll();` is blocking, and causes the participant to wait until all participant have finished with the phase – as measured by the clock `cl` to which they are both registered. Then they do the second phase, and another `Clock.advanceAll();` to make sure that neither proceeds to the third phase until both are ready. This example uses `finish` to wait for both particiants to finish.

```
class ClockEx {
  static def say(s:String) =
     { atomic{x10.io.Console.OUT.println(s);} }
  public static def main(argv:Rail[String]) {
    finish async{
      val cl = Clock.make();
      async clocked(cl) {// Activity A
        say("A-1");
```

```
        Clock.advanceAll();
        say("A-2");
        Clock.advanceAll();
        say("A-3");
      }// Activity A

      async clocked(cl) {// Activity B
        say("B-1");
        Clock.advanceAll();
        say("B-2");
        Clock.advanceAll();
        say("B-3");
      }// Activity B
    }
  }
}
```

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

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

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

- It may spawn and simultaneously *register* a new activity on `c`, with the statement `async clocked(c)S`.

- It may *unregister* itself from `c`, with `c.drop()`. After doing so, it can no longer use most primary operations on `c`.

- It may *resume* the clock, with `c.resume()`, indicating that it has finished

with the current phase associated with `c` and is ready to move on to the next one.

- It may *wait* on the clock, with `c.advance()`. This first does `c.resume()`, and then blocks the current activity until the start of the next phase, *viz.*, until all other activities registered on that clock have called `c.resume()`.

- It may *block* on all the clocks it is registered with simultaneously, by the command `Clock.advanceAll();`. This, in effect, calls `c.advance()` simultaneously on all clocks `c` that the current activity is registered with.

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

15.1   Clock operations

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

15.1.1   Creating new clocks

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

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

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

15.1.2   Registering new activities on clocks

The statement

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

starts a new activity, initially registered with clocks `c1`, `c2`, and `c3`, and running S. The activity running this code must be registered on those clocks. Violations of these conditions are punished by the throwing of a `ClockUseException`.

If an activity a that has executed `c.resume()` then starts a new activity b also registered on `c` (*e.g.*, via `async clocked(c) S`), the new activity b starts out having also resumed `c`, as if it too had executed `c.resume()`.

```
// a
val c = Clock.make();
c.resume();
async clocked(c) {
  // b
  c.advance();
  b_phase_two();
}
c.advance();
a_phase_two();
```

In the proper execution, a and b both perform `c.advance()` and then their phase-2 actions. However, if b were not initially in the resume state for `c`, there would be a race condition; b could perform `c.advance()` and proceed to `b_phase_two` before a performed `c.advance()`.

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

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

15.1.3   Resuming clocks

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

An activity may invoke `resume()` only on a clock it is registered with, and has not yet dropped (§15.1.5). A `ClockUseException` is thrown if this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase.

15.1.4   Advancing clocks

An activity may execute the following method call to signal that it is done with the current phase.

```
Clock.advanceAll();
```

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

`Clock.advanceAll();` may be thought of as calling `c.advance()` in parallel for all clocks that the current activity is registered with. (The parallelism is conceptually important: if activities a and b are both registered on clocks `c` and `d`, and a executes `c.advance(); d.advance()` while b executes `d.advance(); c.advance()`, then the two will deadlock. However, if the two clocks are waited on in parallel, as `Clock.advanceAll();` does, a and b will not deadlock.)

Equivalently, `Clock.advanceAll();` sequentially calls `c.resume()` for each registered clock `c`, in arbitrary order, and then `c.wait()` for each clock, again in arbitrary order.

An activity blocked on `advance()` resumes execution once it is marked for progress by all the clocks it is registered with.

15.1.5   Dropping clocks

An activity may drop a clock by executing `c.drop()`.

The activity is no longer considered registered with this clock. A `ClockUseException` is thrown if the activity has already dropped `c`.

15.2 Deadlock Freedom

In general, programs using clocks can deadlock, just as programs using loops can fail to terminate. However, programs written with a particular syntactic discipline *are* guaranteed to be deadlock-free, just as programs which use only bounded loops are guaranteed to terminate. The syntactic discipline is:

- The `advance()` method may not be called on any clock. (The `Clock.advanceAll();` method is allowed.)

- Inside of `finish{S}`, all clocked `async`s must be in the scope an unclocked `async`.

The second clause prevents the following deadlock.

```
val c:Clock = Clock.make();
async clocked(c) {                    // (A)
      finish async clocked(c) {   // (B) Violates clause 2
            Clock.advanceAll();                   // (Bnext)
      }
      Clock.advanceAll();                         // (Anext)
}
```

`(A)`, first of all, waits for the `finish` containing `(B)` to finish. `(B)` will execute its `advance` at `(Bnext)`, and then wait for all other activities registered on `c` to execute their `advance()`s. However, `(A)` is registered on `c`. So, `(B)` cannot finish until `(A)` has proceeded to `(Anext)`, and `(A)` cannot proceed until `(B)` finishes. Thus, this causes deadlock.

15.3 Program equivalences

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

$$c.resume(); \ Clock.advanceAll(); \ = \ Clock.advanceAll(); \quad (15.1)$$

$$c.resume(); \ d.resume(); \ = \ d.resume(); \ c.resume(); \quad (15.2)$$

$$c.resume(); \ c.resume(); \ = \ c.resume(); \quad (15.3)$$

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

15.4   Clocked Finish

In the most common case of a single clock coordinating a few behaviors, X10 allows coding with an implicit clock. `finish` and `async` statements may be qualified with `clocked`.

A `clocked finish` introduces a new clock. It executes its body in the usual way that a `finish` does— except that, when its body completes, the activity executing the `clocked finish` drops the clock, while it waits for asynchronous spawned `async`s to terminate.

A `clocked async` registers its async with the implicit clock of the surrounding `clocked finish`.

Both the `clocked finish` and `clocked async` may use the `Clock.advanceAll()` method call to advance the implicit clock. Since the implicit clock is not available in a variable, it cannot be manipulated directly. (If you want to manipulate the clock directly, use an explicit clock.)

The following code starts two activities, each of which perform their first phase, wait for the other to finish phase 1, and then perform their second phase.

```
clocked finish {
  clocked async {
     phase("A", 1);
     Clock.advanceAll();
     phase("A", 2);
  }
  clocked async {
     phase("B", 1);
     Clock.advanceAll();
     phase("B", 2);
  }
}
```

Clocked finishes may be nested. The inner `clocked finish` operates in a single phase of the outer one.

16   Local and Distributed Arrays

Arrays provide indexed access to data at a single `Place`, *via* `Point`s—indices of any dimensionality. `DistArray`s is similar, but spreads the data across multiple `Place`s, *via* `Dist`s. We refer to arrays either sort as "general arrays".

This chapter provides an overview of the `x10.array` classes `Array` and `DistArray`, and their supporting classes `Point`, `IntRange`, `Region`, and `Dist`.

16.1   Points

General arrays are indexed by `Point`s, which are n-dimensional tuples of integers. The `rank` property of a point gives its dimensionality. Points can be constructed from integers or `Array[Int](1)`s by the `Point.make` factory methods:

```
val origin_1 : Point{rank==1} = Point.make(0);
val origin_2 : Point{rank==2} = Point.make(0,0);
val origin_5 : Point{rank==5} = Point.make([0,0,0,0,0]);
```

There is an implicit conversion from `Array[Int](1)` to `Point`, giving a convenient syntax for constructing points:

```
val p : Point = [1,2,3];
val q : Point{rank==5} = [1,2,3,4,5];
val r : Point(3) = [11,22,33];
```

The coordinates of a point are available by subscripting; `p(i)` is the `i`th coordinate of the point `p`. `Point(`n`)` is a `type`-defined shorthand for `Point{rank==`n`}`.

16.2   IntRange

An `IntRange` is a representation of a set of consecutive integers: `1..10` is the numbers 1 through 10. There is nothing special about `x10.lang.IntRange`, beyond its package. However, it appears frequently in idioms involving arrays and related constructs, especially rectangular arrays.

One notable idiom involving `IntRange` is the integer iteration idiom. `for(i in 1..10)use(i);` calls `use` on each number 1,2, ..., 10, in turn.

If `m > n`, the `IntRange m..n` is empty. It has no elements, and iterating over it will not execute the body of the loop.

16.3   Regions

A region is a set of points of the same rank. X10 provides a built-in class, `x10.array.Region`, to allow the creation of new regions and to perform operations on regions. Each region R has a property `R.rank`, giving the dimensionality of all the points in it.

```
val MAX_HEIGHT=20;
val Null = Region.makeUnit();  // Empty 0-dimensional region
val R1 = 1..100; // IntRange
val R2 = R1 as Region(1);
val R3 = (0..99) * (-1..MAX_HEIGHT);
val R4 = Region.makeUpperTriangular(10);
val R5 = R4 && R3; // intersection of two regions
```

The `IntRange` value `1..100` can be implicitly or explicitly coerced to a one-dimensional `Region` consisting of the points $\{[m], \ldots, [n]\}$. `IntRange`s are useful in building up regions, especially rectangular regions. In general, we ignore the distinction between an `IntRange` and a rank-one `Region`, except for those occasional situations where the compiler requires attending to the distinction.

By a special dispensation, the compiler knows that, if `r : Region(m)` and `s : Region(n)`, then `r*s : Region(m+n)`. (The X10 type system ordinarily could not specify the sum; the best it could do would be `r*s : Region`, with the rank of the region unknown.) This feature allows more convenient use of arrays; in particular, one does not need to keep track of ranks nearly so much.

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

- `Region.makeEmpty(n)` returns an empty region of rank `n`.

- `Region.makeFull(n)` returns the region containing all points of rank `n`.

- `Region.makeUnit()` returns the region of rank 0 containing the unique point of rank 0. It is useful as the identity for Cartesian product of regions.

- `Region.makeHalfspace(normal:Point, k:Int)` returns the unbounded half-space of rank `normal.rank`, consisting of all points p satisfying p·`normal` ≤ `k`.

- `Region.makeRectangular(min, max)`, where `min` and `max` are `Int` rails or valrails of length `n`, returns a `Region(n)` equal to: `[min(0) .. max(0), ..., min(n-1)..max(n-1)]`.

- `Region.make(regions)` constructs the Cartesian product of the `Region(1)`s in `regions`.

- `Region.makeBanded(size, upper, lower)` constructs the banded `Region(2)` of size `size`, with `upper` bands above and `lower` bands below the diagonal.

- `Region.makeBanded(size)` constructs the banded `Region(2)` with just the main diagonal.

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

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

- If `R` is a region, and `p` a Point of the same rank, then `R+p` is `R` translated forwards by `p` – the region whose points are `r+p` for each `r` in `R`.

- If `R` is a region, and `p` a Point of the same rank, then `R-p` is `R` translated backwards by `p` – the region whose points are `r-p` for each `r` in `R`.

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of the region `(1..2)*(1..2)` are ordered as

    (1,1), (1,2), (2,1), (2,2)

Sequential iteration statements such as `for` (§12.12) iterate over the points in a region in the canonical order.

A region is said to be *rectangular* if it is of the form $(T_1 * \cdots * T_k)$ for some set of intervals $T_i = l_i \text{ .. } h_i$. In particular an `IntRange` turned into a `Region` is rectangular: `(1..10) as Region(1)`. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them (that is, it is *convex*). (Banded and triangular regions are not rectangular.) The operation `R.boundingBox()` gives the smallest rectangular region containing R.

16.3.1   Operations on regions

Let `R` be a region. A *sub-region* is a subset of `R`.

Let `R1` and `R2` be two regions whose types establish that they are of the same rank. Let `S` be another region; its rank is irrelevant.

`R1 && R2` is the intersection of `R1` and `R2`, *viz.*, the region containing all points which are in both `R1` and `R2`. For example, `1..10 && 2..20` is `2..10`.

`R1 * S` is the Cartesian product of `R1` and `S`, formed by pairing each point in `R1` with every point in `S`. Thus, `(1..2)*(3..4)*(5..6)` is the region of rank 3 containing the eight points with coordinates `[1,3,5]`, `[1,3,6]`, `[1,4,5]`, `[1,4,6]`, `[2,3,5]`, `[2,3,6]`, `[2,4,5]`, `[2,4,6]`.

For a region `R` and point `p` of the same rank, `R+p` and `R-p` represent the translation of the region forward and backward by `p`. That is, `R+p` is the set of points `p+q` for all `q` in `R`, and `R-p` is the set of `q-p`.

More `Region` methods are described in the API documentation.

16.4   Arrays

Arrays are organized data, arranged so that it can be accessed by subscript. An `Array[T] A` has a `Region A.region`, telling which `Points` are in `A`. For each point `p` in `A.region`, `A(p)` is the datum of type `T` associated with `p`. X10 implementations should attempt to store `Arrays` efficiently, and to make array element accesses quick—*e.g.*, avoiding constructing `Points` when unnecessary.

This generalizes the concepts of arrays appearing in many other programming languages. A `Point` may have any number of coordinates, so an `Array` can have, in effect, any number of integer subscripts.

Indeed, it is possible to write code that works on `Array`s regardless of dimension. For example, to add one `Array[Int] src` into another `dest`,

```
static def addInto(src: Array[Int], dest:Array[Int])
  {src.region == dest.region}
  = {
    for (p in src.region)
        dest(p) += src(p);
  }
```

Since `p` is a `Point`, it can hold as many coordinates as are necessary for the arrays `src` and `dest`.

The basic operation on arrays is subscripting: if `A` is an `Array[T]` and `p` a point with the same rank as `A.region`, then `A(p)` is the value of type `T` associated with point `p`.

Array elements can be changed by assignment. If `t:T`,

```
A(p) = t;
```

modifies the value associated with `p` to be `t`, and leaves all other values in `A` unchanged.

An `Array[T]` `A` has:

- `A.region`: the `Region` upon which `A` is defined.

- `A.size`: the number of elements in `A`.

- `A.rank`, the rank of the points usable to subscript `A`. Identical to `A.region.rank`.

16.4.1 Array Constructors

To construct an array whose elements all have the same value `init`, call `new Array[T](R, init)`. For example, an array of a thousand `"oh!"`s can be made by: `new Array[String](1..1000, "oh!")`.

To construct and initialize an array, call the two-argument constructor. `new Array[T](R, f)` constructs an array of elements of type T on region R, with A(p) initialized to `f(p)` for each point p in R. `f` must be a function taking a point of rank `R.rank` to a value of type T. *E.g.*, to construct an array of a hundred zero values, call `new Array[Int](1..100, (Point(1))=>0)`. To construct a multiplication table, call `new Array[Int]((0..9)*(0..9), (p:Point(2)) => p(0)*p(1))`.

Other constructors are available; see the API documentation and §11.26.

16.4.2   Array Operations

The basic operation on `Array`s is subscripting. If `A:Array[T]` and `p:Point{rank == A.rank}`, then `a(p)` is the value of type T appearing at position p in A. The syntax is identical to function application, and, indeed, arrays may be used as functions. `A(p)` may be assigned to, as well, by the usual assignment syntax `A(p)=t`. (This uses the application and setting syntactic sugar, as given in §8.7.5.)

Sometimes it is more convenient to subscript by integers. Arrays of rank 1-4 can, in fact, be accessed by integers:

```
val A1 = new Array[Int](1..10, 0);
A1(4) = A1(4) + 1;
val A4 = new Array[Int]((1..2)*(1..3)*(1..4)*(1..5), 0);
A4(2,3,4,5) = A4(1,1,1,1)+1;
```

Iteration over an `Array` is defined, and produces the `Point`s of the array's region. If you want to use the values in the array, you have to subscript it. For example, you could double every element of an `Array[Int]` by:

```
for (p in A) A(p) = 2*A(p);
```


16.5   Distributions

Distributed arrays are spread across multiple `Place`s. A *distribution*, a mapping from a region to a set of places, describes where each element of a distributed array is kept. Distributions are embodied by the class `x10.array.Dist`. This class is `final` in X10 v2.2; future versions of the language may permit user-definable

distributions. The *rank* of a distribution is the rank of the underlying region, and thus the rank of every point that the distribution applies to.

```
val R  <: Region = 1..100;
val D1 <: Dist = Dist.makeBlock(R);
val D2 <: Dist = Dist.makeConstant(R, here);
```

D1 distributes the region R in blocks, with a set of consecutive points at each place, as evenly as possible. D2 maps all the points in R to here.

Let D be a distribution. `D.region` denotes the underlying region. Given a point p, the expression `D(p)` represents the application of D to p, that is, the place that p is mapped to by D. The evaluation of the expression `D(p)` throws an `ArrayIndexOutofBoundsException` if p does not lie in the underlying region.

16.5.1 `PlaceGroups`

A `PlaceGroup` represents an ordered set of `Place`s. `PlaceGroup`s exist for performance and scaleability: they are more efficient, in certain critical places, than general collections of `Place`. `PlaceGroup` implements `Sequence[Place]`, and thus provides familiar operations – `pg.size()` for the number of places, `pg.iterator()` to iterate over them, etc.

`PlaceGroup` is an abstract class. The concrete class `SparsePlaceGroup` is intended for a small group of places: in particular, `new SparsePlaceGroup(somePlace)` is a good `PlaceGroup` containing one place. `new SparsePlaceGroup(seqPlaces)` constructs a sparse place group from a sorted sequence of places.

16.5.2 Operations returning distributions

Let R be a region, Q a `PlaceGroup`, and P a place.

Unique distribution   The distribution `Dist.makeUnique(Q)` is the unique distribution from the region `(1..k) as Region(1)` to Q mapping each point i to p_i.

Constant distributions.   The distribution `Dist.makeConstant(R,P)` maps every point in region R to place P. The special case `Dist.makeConstant(R)` maps every point in R to here.

Block distributions.   The distribution `Dist.makeBlock(R)` distributes the elements of R, in approximately-even blocks, over all the places available to the program. There are other `Dist.makeBlock` methods capable of controlling the distribution and the set of places used; see the API documentation.

Domain Restriction.   If `D` is a distribution and R is a sub-region of `D.region`, then `D | R` represents the restriction of D to R—that is, the distribution that takes each point `p` in R to `D(p)`, but doesn't apply to any points but those in R.

Range Restriction.   If `D` is a distribution and P a place expression, the term `D | P` denotes the sub-distribution of D defined over all the points in the region of D mapped to P.

Note that `D | here` does not necessarily contain adjacent points in `D.region`. For instance, if `D` is a cyclic distribution, `D | here` will typically contain points that differ by the number of places. An implementation may find a way to still represent them in contiguous memory, *e.g.*, using a complex arithmetic function to map from the region index to an index into the array.

16.6   Distributed Arrays

Distributed arrays, instances of `DistArray[T]`, are very much like `Array`s, except that they distribute information among multiple `Place`s according to a `Dist` value passed in as a constructor argument. For example, the following code creates a distributed array holding a thousand cells, each initialized to 0.0, distributed via a block distribution over all places.

```
val R <: Region = 1..1000;
val D <: Dist = Dist.makeBlock(R);
val da <: DistArray[Float] = DistArray.make[Float](D, (Point(1))=>0.0f);
```

16.7   Distributed Array Construction

`DistArray`s are instantiated by invoking one of the `make` factory methods of the `DistArray` class. A `DistArray` creation must take either an `Int` as an argument

or a `Dist`. In the first case, a distributed array is created over the distribution `Dist.makeConstant([0:N-1],here)`; in the second over the given distribution.

A distributed array creation operation may also specify an initializer function. The function is applied in parallel at all points in the domain of the distribution. The construction operation terminates locally only when the `DistArray` has been fully created and initialized (at all places in the range of the distribution).

For instance:

```
val data : DistArray[Int]
    = DistArray.make[Int](Dist.makeConstant(1..1000), ([i]:Point(1)) => i);
val blocked = Dist.makeBlock((1..1000)*(1..1000));
val data2 : DistArray[Int]
    = DistArray.make[Int](blocked, ([i,j]:Point(2)) => i*j);
```

The first declaration stores in `data` a reference to a mutable distributed array with `1000` elements each of which is located in the same place as the array. The element at `[i]` is initialized to its index `i`.

The second declaration stores in `data2` a reference to a mutable two-dimensional distributed array, whose coordinates both range from 1 to 1000, distributed in blocks over all `Place`s, initialized with `i*j` at point `[i,j]`.

16.8   Operations on Arrays and Distributed Arrays

Arrays and distributed arrays share many operations. In the following, let `a` be an array with base type T, and `da` be an array with distribution D and base type T.

16.8.1   Element operations

The value of `a` at a point `p` in its region of definition is obtained by using the indexing operation `a(p)`. The value of `da` at `p` is similarly `da(p)` This operation may be used on the left hand side of an assignment operation to update the value: `a(p)=t;` and `da(p)=t;` The operator assignments, `a(i) += e` and so on, are also available.

It is a runtime error to use either `da(p)` or `da(p)=v` at a place other than `da.dist(p)`, *viz.* at the place that the element exists.

16.8.2   Constant promotion

For a region R and a value v of type T, the expression `new Array[T](R, v)` pro-
duces an array on region R initialized with value v Similarly, for a distribution D
and a value v of type T the expression `DistArray.make[T](D, (Point(D.rank))=>v)`
constructs a distributed array with distribution D and base type T initialized with v
at every point.

Note that `Array`s are constructed by constructor calls, but `DistArray`s are con-
structed by calls to the factory methods `DistArray.make`. This is because `Array`s
are fairly simple objects, but `DistArray`s may be implemented by different classes
for different distributions. The use of the factory method gives the library writer
the freedom to select appropriate implementations.

16.8.3   Restriction of an array

Let R be a sub-region of `da.region`. Then `da | R` represents the sub-`DistArray`
of da on the region R. That is, `da | R` has the same values as da when sub-
scripted by a point in region `R && da.region`, and is undefined elsewhere. '
Recall that a rich set of operators are available on distributions (§16.5) to obtain
sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

16.8.4   Operations on Whole Arrays

Pointwise operations   The unary `map` operation applies a function to each ele-
ment of a distributed or non-distributed array, returning a new distributed array
with the same distribution, or a non-distributed array with the same region. For
example, the following produces an array of cubes:

```
val A = new Array[Int](1..10, (p:Point(1))=>p(0) );
// A = 1,2,3,4,5,6,7,8,9,10
val cube = (i:Int) => i*i*i;
val B = A.map(cube);
// B = 1,8,27,64,216,343,512,729,1000
```

A variant operation lets you specify the array B into which the result will be stored.

```
val A = new Array[Int](1..10, (p:Point(1))=>p(0) );
// A = 1,2,3,4,5,6,7,8,9,10
```

```
val cube = (i:Int) => i*i*i;
val B = new Array[Int](A.region); // B = 0,0,0,0,0,0,0,0,0,0
A.map(B, cube);
// B = 1,8,27,64,216,343,512,729,1000
```

This is convenient if you have an already-allocated array lying around unused. In particular, it can be used if you don't need A afterwards and want to reuse its space:

```
val A = new Array[Int](1..10, (p:Point(1))=>p(0) );
// A = 1,2,3,4,5,6,7,8,9,10
val cube = (i:Int) => i*i*i;
A.map(A, cube);
// A = 1,8,27,64,216,343,512,729,1000
```

The binary map operation takes a binary function and another array over the same region or distributed array over the same distribution, and applies the function pointwise to corresponding elements of the two arrays, returning a new array or distributed array of the same shape. The following code adds two distributed arrays:

```
static def add(da:DistArray[Int], db: DistArray[Int]{da.dist==db.dist})
    = da.map(db, Int.+);
```

Reductions   Let f be a function of type (T,T)=>T. Let a be an array over base type T. Let unit be a value of type T. Then the operation a.reduce(f, unit) returns a value of type T obtained by combining all the elements of a by use of f in some unspecified order (perhaps in parallel). The following code gives one method which meets the definition of reduce, having a running total r, and accumulating each value a(p) into it using f in turn. (This code is simply given as an example; Array has this operation defined already.)

```
def oneWayToReduce[T](a:Array[T], f:(T,T)=>T, unit:T):T {
  var r : T = unit;
  for(p in a.region) r = f(r, a(p));
  return r;
}
```

For example, the following sums an array of integers. f is addition, and unit is zero.

```
val a = [1,2,3,4];
val sum = a.reduce(Int.+, 0);
assert(sum == 10); // 10 == 1+2+3+4
```

Other orders of evaluation, degrees of parallelism, and applications of `f(x,unit)` and `f(unit,x)` are also correct. In order to guarantee that the result is precisely determined, the function `f` should be associative and commutative, and the value `unit` should satisfy `f(unit,x) == x == f(x,unit)` for all `x:T`.

`DistArray`s have the same operation. This operation involves communication between the places over which the `DistArray` is distributed. The X10 implementation guarantees that only one value of type `T` is communicated from a place as part of this reduction process.

Scans   Let `f:(T,T)=>T`, `unit:T`, and a be an `Array[T]` or `DistArray[T]`. Then `a.scan(f,unit)` is the array or distributed array of type `T` whose ith element in canonical order is the reduction by `f` with unit `unit` of the first i elements of a.

This operation involves communication between the places over which the distributed array is distributed. The X10 implementation will endeavour to minimize the communication between places to implement this operation.

Other operations on arrays, distributed arrays, and the related classes may be found in the `x10.array` package.

17 Annotations

X10 provides an an annotation system system for to allow the compiler to be
extended with new static analyses and new transformations.

Annotations are interface types that decorate the abstract syntax tree of an X10
program. The X10 type-checker ensures that an annotation is a legal interface
type. In X10, interfaces may declare both methods and properties. Therefore, like
any interface type, an annotation may instantiate one or more of its interface's
properties.

17.1 Annotation syntax

The annotation syntax consists of an "@" followed by an interface type.

Annotations	::=	*Annotation*	*(20.144)*
	|	*Annotations Annotation*	
Annotation	::=	@ *NamedType*	*(20.145)*

Annotations can be applied to most syntactic constructs in the language including
class declarations, constructors, methods, field declarations, local variable decla-
rations and formal parameters, statements, expressions, and types. Multiple occur-
rences of the same annotation (i.e., multiple annotations with the same interface
type) on the same entity are permitted.

Recall that interface types may have dependent parameters.

The following examples illustrate the syntax:

- Declaration annotations:

```
// class annotation
@Value
class Cons { ... }

// method annotation
@PreCondition(0 <= i && i < this.size)
public def get(i: Int): Object { ... }

// constructor annotation
@Where(x != null)
def this(x: T) { ... }

// constructor return type annotation
def this(x: T): C@Initialized { ... }

// variable annotation
@Unique x: A;
```

- Type annotations:

```
List@Nonempty

Int@Range(1,4)

Array[Array[Double]]@Size(n * n)
```

- Expression annotations:

```
m() : @RemoteCall
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(0)
@MaxIterations(n)
for (var i: Int = 0; i < n; i++) { ... }

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

17.2 Annotation declarations

Annotations are declared as interfaces. They must be subtypes of the interface `x10.lang.annotation.Annotation`. Annotations on particular static entities must extend the corresponding `Annotation` subclasses, as follows:

- Expressions—`ExpressionAnnotation`

- Statements—`StatementAnnotation`

- Classes—`ClassAnnotation`

- Fields—`FieldAnnotation`

- Methods—`MethodAnnotation`

- Imports—`ImportAnnotation`

- Packages—`PackageAnnotation`

18 Native Code Integration

At times it becomes necessary to call non-X10 code from X10, perhaps to make use of specialized libraries in other languages or to write more precisely controlled code than X10 generally makes available. The `@Native(lang,code)` `Phrase` annotation from `x10.compiler.Native` in X10 can be used to tell the X10 compiler to generate `code` for certain kinds of `Phrase`, instead of what it would normally compile to, when compiling to the `lang` back end.

18.1 Native `static` Methods

`static` methods can be given native implementations. Note that these implementations are syntactically *expressions*, not statements, in C++ or Java. Also, it is possible (and common) to provide native implementations into both Java and C++ for the same method.

```
import x10.compiler.Native;
class Son {
  @Native("c++", "printf(\"Hi!\")")
  @Native("java", "System.out.println(\"Hi!\")")
  static def printNatively():void = {};
}
```

If only some back-end languages are given, the X10 code will be used for the remaining back ends:

```
import x10.compiler.Native;
class Land {
  @Native("c++", "printf(\"Hi from C++!\")")
  static def example():void = {
```

```
    x10.io.Console.OUT.println("Hi from X10!");
  };
}
```

The `native` modifier on methods indicates that the method must not have an X10 code body, and `@Native` implementations must be given for all back ends:

```
import x10.compiler.Native;
class Plants {
  @Native("c++", "printf(\"Hi!\")")
  @Native("java", "System.out.println(\"Hi!\")")
  static native def printNatively():void;
}
```

Values may be returned from external code to X10. Scalar types in Java and C++ correspond directly to the analogous types in X10.

```
import x10.compiler.Native;
class Return {
  @Native("c++", "1")
  @Native("java", "1")
  static native def one():Int;
}
```

Parameters may be passed to external code. `(#1)` is the first parameter, `(#2)` the second, and so forth. (`(#0)` is the name of the enclosing class.) Be aware that this is macro substitution rather than normal parameter passing; *e.g.*, if the first actual parameter is `i++`, and `(#1)` appears twice in the external code, `i` will be incremented twice. For example, a (ridiculous) way to print the sum of two numbers is:

```
import x10.compiler.Native;
class Species {
  @Native("c++", "printf(\"Sum=%d\", ((#1)+(#2)) )")
  @Native("java", "System.out.println(\"\" + ((#1)+(#2)))")
  static native def printNatively(x:Int, y:Int):void;
}
```

Static variables in the class are available in the external code.

18.2   Native Blocks

Any block may be annotated with `@Native(lang,stmt)`, indicating that, in the given back end, it should be implemented as `stmt`. All value variables from the surrounding context are available inside `stmt`. For example, the method call `born.example(10)`, if compiled to Java, changes the field `y` of a `Born` object to 10. If compiled to C++ (for which there is no `@Native`), it sets it to 3.

```
import x10.compiler.Native;
class Born {
  var y : Int = 1;
  public def example(x:Int):Int{
    @Native("java", "y=x;")
    {y = 3;}
    return y;
  }
}
```

Note that the code being replaced is a statement – the block `{y = 3;}` in this case – so the replacement should also be a statement.

Other X10 constructs may or may not be available in Java and/or C++ code. For example, type variables do not correspond exactly to type variables in either language, and may not be available there. The exact compilation scheme is *not* fully specified. You may inspect the generated Java or C++ code and see how to do specific things, but there is no guarantee that fancy extern coding will continue to work in later versions of X10.

The full facilities of C++ or Java are available in native code blocks. However, there is no guarantee that advanced features behave sensibly. You must follow the exact conventions that the code generator does, or you will get unpredictable results. Furthermore, the code generator's conventions may change without notice or documentation from version to version. In most cases the code should either be a very simple expression, or a method or function call to external code.

18.3   External Java Code

When X10 is compiled to Java, mentioning a Java class name in native code will cause the Java compiler to find it in the sourcepath or classpath, in the usual way.

This requires no particular extra work from the programmer.

18.4 External C++ Code

C++ code can be linked to X10 code, either by writing auxiliary C++ files and adding them with suitable annotations, or by linking libraries.

18.4.1 Auxiliary C++ Files

Auxiliary C++ code can be written in `.h` and `.cc` files, which should be put in the same directory as the the X10 file using them. Connecting with the library uses the `@NativeCPPInclude(dot_h_file_name)` annotation to include the header file, and the `@NativeCPPCompilationUnit(dot_cc_file_name)` annotation to include the C++ code proper. For example:

MyCppCode.h:

```
void foo();
```

MyCppCode.cc:

```
#include <cstdlib>
#include <cstdio>
void foo() {
    printf("Hello World!\n");
}
```

Test.x10:

```
import x10.compiler.Native;
import x10.compiler.NativeCPPInclude;
import x10.compiler.NativeCPPCompilationUnit;

@NativeCPPInclude("MyCPPCode.h")
@NativeCPPCompilationUnit("MyCPPCode.cc")
public class Test {
    public static def main (args:Array[String](1)) {
        { @Native("c++","foo();") {} }
    }
}
```

18.4.2   C++ System Libraries

If we want to additionally link to more libraries in `/usr/lib` for example, it is necessary to adjust the post-compilation directly. The post-compilation is the compilation of the C++ which the X10-to-C++ compiler `x10c++` produces.

The mechanism used for this is the `-post` command line parameter to `x10c++`. The following example shows how to compile `blas` into the executable via post compiler parameters.

```
x10c++ Test.x10 -post '# # -I /usr/local/blas # -L /usr/local/blas -lblas'
```

- The first # means to use the default compiler for the architecture (from x10rt properties file).

- The second # is substituted for the .cc files and CXXFLAGS that would ordinarily be used.

- The third # is substituted for the libraries and LDFLAGS that would ordinarily used.

- For the second and third, if a % is used instead of a # then the the substitution does not occur in that position. The % is erased. The desired parameter value should appear after the % on the line. This allows a complete override of the postcompiler behaviour.

19 Definite Assignment

X10 requires, reasonably enough, that every variable be set before it is read. Sometimes this is easy, as when a variable is declared and assigned together:

```
var x : Int = 0;
assert x == 0;
```

However, it is convenient to allow programs to make decisions before initializing variables.

```
def example(a:Int, b:Int) {
  val max:Int;
  // max cannot be read here.
  if (a > b) max = a;
  else max = b;
  assert max >= a && max >= b;
}
```

This is particularly useful for `val` variables. `var`s could be initialized to a default value and then reassigned with the right value, but `val`s must be initialized once correctly and cannot be changed.

However, one must be careful – and the X10 compiler enforces this care. Without the `else` clause, the preceding code might not give `max` a value by the `assert`.

This leads to the concept of *definite assignment* [**?**]. A variable is definitely assigned at a point in code if, no matter how that point in code is reached, the variable has been assigned to. In X10, variables must be definitely assigned before they can be read.

As X10 requires that `val` variables *not* be initialized twice, we need the dual concept as well. A variable is *definitely unassigned* at a point in code if it cannot have

been assigned there. For example, immediately after `val x:Int`, x is definitely unassigned.

Finally, we need the concept of *singly* and *multiply assigned*. A variable is singly assigned in a block if it is assigned precisely once; it is multiply assigned if it could possibly be assigned more than once. `var`s can multiply assigned as desired. `val`s must be singly assigned. For example, the code `x = 1; x = 2;` is perfectly fine if x is a `var`, but incorrect (even in a constructor) if x is a `val`.

At some points in code, a variable might be neither definitely assigned nor definitely unassigned. Such states are not always useful.

```
def example(flag : Boolean) {
  var x : Int;
  if (flag) x = 1;
  // x is neither def. assigned nor unassigned.
  x = 2;
  // x is def. assigned.
```

This shows that we cannot simply define "definitely unassigned" as "not definitely assigned". If x had been a `val` rather than a `var`, the previous example would not be allowed.

Unfortunately, a completely accurate definition of "definitely assigned" or "definitely unassigned" is undecidable – impossible for the compiler to determine. So, X10 takes a *conservative approximation* of these concepts. If X10's definition says that x is definitely assigned (or definitely unassigned), then it will be assigned (or not assigned) in every execution of the program.

However, there are programs which X10's algorithm says are incorrect, but which actually would behave properly if they were executed. In the following example, `flag` is either `true` or `false`, and in either case x will be initialized. However, X10's analysis does not understand this — thought it *would* understand if the example were coded with an `if-else` rather than a pair of `if`s. So, after the two `if` statements, x is not definitely assigned, and thus the `assert` statement, which reads it, is forbidden.

```
def example(flag:Boolean) {
  var x : Int;
  if (flag) x = 1;
  if (!flag) x = 2;
  // Not Allowed: assert x < 3;
```

```
    }
```

19.1 Asynchronous Definite Assignment

Local variables (but not fields) allow *asynchronous assignment*. A local variable can be assigned in an `async` statement, and, when the `async` is `finish`ed, the variable is definitely assigned.

Example:

```
val a : Int;
finish {
  async {
    a = 1;
  }
  // a is not definitely assigned here
}
// a is definitely assigned after 'finish'
assert a==1;
```

This concept supports a core X10 programming idiom. A `val` variable may be initialized asynchronously, thereby providing a means for returning a value from an `async` to be used after the enclosing `finish`.

19.2 Characteristics of Definite Assignment

The properties "definitely assigned", "singly assigned", and "definitely unassigned" are computed by a conservative approximation of X10's evaluation rules.

The precise details are up to the implementation. Many basic cases must be handled accurately; *e.g.*, `x=1;` definitely and singly assigns `x`.

However, in more complicated cases, a conforming X10 may mark as invalid some code which, when executed, would actually be correct. For example, the following program fragment will always result in `x` being definitely and singly assigned:

```
val x : Int;
var b : Boolean = mysterious();
if (b) {
    x = cryptic();
}
if (!b) {
    x = unknown();
}
```

However, most conservative approximations of program execution won't mark x as properly initialized. For this to be correct, precisely one of the two assignments to x must be executed. If b were true initially, it would still be true after the call to cryptic() — since methods cannot modify their caller's local variables – and so the first but not the second assignment would happen. If b were false initially, it would still be false when !b is tested, and so the second but not the first assignment would happen. Either way, x is definitely and singly assigned.

However, for a slightly different program, this analysis would be wrong. *E.g.*, if b were a field of this rather than a local variable, cryptic() could change b; if b were true initially, both assignments might happen, which is incorrect for a val.

This sort of reasoning is beyond most conservative approximation algorithms. (Indeed, many do not bother checking that !b late in the program is the opposite of b earlier.) Algorithms that pay attention to such details and subtleties tend to be fairly expensive, which would lead to very slow compilation for X10 – for the sake of obscure cases.

X10's analysis provides at least the following guarantees. We describe them in terms of a statement S performing some collection of possible numbers of assignments to variables — on a scale of "0", "1", and "many". For example, if(b) x=1; else {x=1;x=2;y=2;} might assign to x one or many times, and might assign to y zero or one time. Hence, after it, x is definitely assigned and may be multiply assigned, and y is neither definitely assigned nor definitely unassigned.

These descriptions are combined in natural ways. For example, if R says that x will be assigned 0 or 1 times, and S says it will be assigned precisely once, then R;S will assign it one or many times. If only one or R or S will occur, as from if(b)R; else S;, then x may be assigned 0 or 1 times.

This information is sufficient for the tests X10 makes. If x can only be assigned to once in S, it is definitely assigned. It is an error if x is ever read at a point where it have been assigned zero times. It is an error if a val may be assigned many times.

We do not guarantee that any particular X10 compiler uses this algorithm; indeed, as of the time of writing, the X10 compiler uses a somewhat more precise one. However, any conformant X10 compiler must provide results which are at least as accurate as this analysis.

Assignment: `x = e`

`x = e` assigns to `x`, in addition to whatever assignments `e` makes. For example, if `this.setX(y)` sets a field `x` to `y` and returns `y`, then `x = this.setX(y)` definitely and multiply assigns `x`.

`async` **and** `finish`

By itself, `async S` provides few guarantees. After `async{x=1;}` finishes, we know that there is a separate activity which will, when the scheduler lets it, set `x` to 1. We do not know that anything has happened yet.

However, if there is a `finish` around the `async`, the situation is clearer. After `finish{ async{ x=1; }}`, `x` has definitely been assigned.

In general, if an `async S` appears in the body of a `finish` in a way that guarantees that it will be executed, then, after the `finish`, the assignments made by S will have occurred. For example, if S definitely assigns to `x`, and the body of the `finish` guarantees that `async S` will be executed, then `finish{...async S...}` definitely assigns `x`.

`if` **and** `switch`

When `if(E) S else T` finishes, it will have performed the assignments of E, together with those of either S or T but not both. For example, `if(b) x=1; else x=2;` definitely assigns x, but `if(b) x=1;` does not.

`switch` is more complex, but follows the same principles as `if`. For example, `switch(E){case 1: A; break; case 2: B; default: C;}` performs the assignments of E, and those of precisely one of A, or B;C, or C. Note that case 2 falls through to the default case, so it performs the same assignments as B;C.

Sequencing

When `R;S` finishes, it will have performed the assignments of `R` and those of `S`. For example, `x=1;y=2;` definitely assigns `x` and `y`, and `x=1;x=2;` multiply assigns `x`.

Loops

`while(E)S` performs the assignments of `E` one or more times, and those of `S` zero or more times. For example, if `while(b()){x=1;}` might assign to `x` zero, one, or many times. `do S while(E)` performs the assignments of `E` one or more times, and those of `S` one or more times.

`for(A;B;C)D` performs the assignments of `A` once, those of `B` one or more times, and those of `C` and `D` one or more times. `for(x in E)S` performs the assignments of `E` once and those of `S` zero or more times.

`continue` and `break` inside of a loop may be ignored conservatively.

Loops are of very little value for providing definite assignments, since X10 does not in general know how many times they will be executed.

Method Calls

A method call `E.m(A,B)` performs the assignments of `E`, `A`, and `B` once each, and also those of `m`. This implies that X10 must be aware of the possible assignments performed by each method.

If X10 has complete information about `m` (as when `m` is a `private` or `final` method), this is straightforward. When such information is fundamentally impossible to acquire, as when `m` is a non-final method invocation, X10 has no choice but to assume that `m` might do anything that a method can do.

- `m` cannot assign to local fields of the caller; methods have no such power.

- `m` can assign to `var` fields of `this` freely, unless this is prohibited by an annotation; see §**??**

- `m` cannot initialize `val` fields of `this`, unless permitted to do so by an annotation; see §**??**.

`at` **and** `athome`

`at(E)S` performs the assignments of E. Within S, only those assignments to variables `x` from the surrounding environment which take place within a suitable `athome(x)R` are counted.

Example: *In the following code, the outer variable named* `a` *is definitely assigned once, by the assignment* `a = 3;`. *The inner variable (also named* `a`*) is definitely multiply assigned by the two assignments* `a = 1;` *and* `a = 2;` *between the* `at` *and the* `athome`.

```
var a : Int;
at(here.next(); var a : Int = a;) {
  a = 1;
  a = 2;
  athome(a) a = 3;
}
```

`atomic`

`atomic S` performs the assignments of S, and `when(E)S` performs those of E and S.

`try`

`try S catch(x:T1) E1 ... catch(x:Tn) En  finally F` performs some or all of the assignments of S, plus all the assignments of zero or one of the E's, plus those of F. For example,

```
try {
  x = boomy();
  x = 0;
}
catch(e:Boom) { y = 1; }
finally { z = 1; }
```

assigns `x` zero, one, or many times[1], assigns `y` zero or one time, and assigns `z` exactly once.

---

[1]A more precise analysis could discover that `x` cannot be initialized only once.

Expression Statements

Expression statements `E;`, and other statements that execute an expression and do something innocuous with it (local variable declaration and `assert`) have the same effects as `E`.

`return,` `throw`

Statements that do not finish normally, such as `return` and `throw`, don't initialize anything (though the computation of the return or thrown value may). They also terminate a line of computation. For example, `if(b) {x=1; return;}  x=2;` definitely and singly assigns `x`.

20 Grammar

In this grammar, $X^?$ denotes an optional X element.

MethodInvocation	::=	*MethodPrimaryPrefix* (*ArgumentList$^?$*)	*(1)*
	|	*MethodSuperPrefix* (*ArgumentList$^?$*)	
	|	*MethodClassNameSuperPrefix* (*ArgumentList$^?$*)	
Mod	::=	`abstract`	*(2)*
	|	*Annotation*	
	|	`atomic`	
	|	`final`	
	|	`native`	
	|	`private`	
	|	`protected`	
	|	`public`	
	|	`static`	

		$\mid$ `transient`	
		$\mid$ `clocked`	
MethMods	::=	*Mods*$^?$	*(3)*
		$\mid$ *MethMods* `property`	
		$\mid$ *MethMods Mod*	
TypeDefDecl	::=	*Mods*$^?$ `type` *Id TypeParams*$^?$ *FormalParams*$^?$ *WhereClause*$^?$ `=` *Type* `;`	*(4)*
Properties	::=	`(` *PropertyList* `)`	*(5)*
PropertyList	::=	*Property*	*(6)*
		$\mid$ *PropertyList* `,` *Property*	
Property	::=	*Annotations*$^?$ *Id ResultType*	*(7)*
MethodDecl	::=	*MethMods* `def` *Id TypeParams*$^?$ *FormalParams WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	*(8)*
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `(` *FormalParam* `)` *BinOp* `(` *FormalParam* `)` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ *PrefixOp* `(` *FormalParam* `)` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `this` *BinOp* `(` *FormalParam* `)` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `(` *FormalParam* `)` *BinOp* `this` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ *PrefixOp* `this` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` `this` *TypeParams*$^?$ *FormalParams WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` `this` *TypeParams*$^?$ *FormalParams* `=` `(` *FormalParam* `)` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `(` *FormalParam* `)` `as` *Type WhereClause*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `(` *FormalParam* `)` `as` `?` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
		$\mid$ *MethMods* `operator` *TypeParams*$^?$ `(` *FormalParam* `)` *WhereClause*$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	

PropertyMethodDecl	::=	*MethMods* *Id* *TypeParams$^?$* *FormalParams WhereClause$^?$ HasResultType$^?$ MethodBody*	*(9)*
	|	*MethMods* *Id* *WhereClause$^?$* *HasResultType$^?$ MethodBody*	
ExplicitCtorInvocation	::=	`this` *TypeArguments$^?$* (*ArgumentList$^?$*) ;	*(10)*
	|	`super` *TypeArguments$^?$* (*ArgumentList$^?$*) ;	
	|	*Primary* . `this` *TypeArguments$^?$* (*ArgumentList$^?$*) ;	
	|	*Primary* . `super` *TypeArguments$^?$* (*ArgumentList$^?$*) ;	
NormalInterfaceDecl	::=	*Mods$^?$* `interface` *Id* *TypeParamsWithVariance$^?$ WhereClause$^?$ ExtendsInterfaces$^?$ InterfaceBody*	*(11)*
ClassInstCreationExp	::=	`new` *TypeName* *TypeArguments$^?$* (*ArgumentList$^?$*) *ClassBody$^?$*	*(12)*
	|	`new` *TypeName* [*Type*] [*ArgumentList$^?$*]	
	|	*Primary* . `new` *Id* *TypeArguments$^?$* (*ArgumentList$^?$*) *ClassBody$^?$*	
	|	*AmbiguousName* . `new` *Id* *TypeArguments$^?$* (*ArgumentList$^?$*) *ClassBody$^?$*	
AssignPropertyCall	::=	`property` *TypeArguments$^?$* (*ArgumentList$^?$*) ;	*(13)*
Type	::=	*FunctionType*	*(14)*
	|	*ConstrainedType*	
FunctionType	::=	*TypeParams$^?$* (*FormalParamList$^?$*) *WhereClause$^?$ Offers$^?$* => *Type*	*(15)*
ClassType	::=	*NamedType*	*(16)*
AnnotatedType	::=	*Type Annotations*	*(17)*
ConstrainedType	::=	*NamedType*	*(18)*
	|	*AnnotatedType*	
	|	(*Type*)	
PlaceType	::=	*PlaceExp*	*(19)*
SimpleNamedType	::=	*TypeName*	*(20)*

	|	*Primary . Id*	
	|	*DepNamedType . Id*	
DepNamedType	::=	*SimpleNamedType DepParams*	(21)
	|	*SimpleNamedType Arguments*	
	|	*SimpleNamedType Arguments DepParams*	
	|	*SimpleNamedType TypeArguments*	
	|	*SimpleNamedType TypeArguments DepParams*	
	|	*SimpleNamedType TypeArguments Arguments*	
	|	*SimpleNamedType TypeArguments Arguments Dep-Params*	
NamedType	::=	*SimpleNamedType*	(22)
	|	*DepNamedType*	
DepParams	::=	{ *ExistentialList$^?$ Conjunction$^?$* }	(23)
TypeParamsWithVariance	:=	[*TypeParamWithVarianceList*]	(24)
TypeParams	::=	[*TypeParamList*]	(25)
FormalParams	::=	(*FormalParamList$^?$*)	(26)
Conjunction	::=	*Exp*	(27)
	|	*Conjunction , Exp*	
SubtypeConstraint	::=	*Type <: Type*	(28)
	|	*Type :> Type*	
WhereClause	::=	*DepParams*	(29)
ExistentialList	::=	*FormalParam*	(30)
	|	*ExistentialList ; FormalParam*	
ClassDecl	::=	*StructDecl*	(31)
	|	*NormalClassDecl*	
NormalClassDecl	::=	*Mods$^?$* class *Id TypeParamsWithVariance$^?$ Properties$^?$ WhereClause$^?$ Super$^?$ Interfaces$^?$ ClassBody*	(32)
StructDecl	::=	*Mods$^?$* struct *Id TypeParamsWithVariance$^?$ Properties$^?$ WhereClause$^?$ Interfaces$^?$ ClassBody*	(33)
CtorDecl	::=	*Mods$^?$* def this *TypeParams$^?$ FormalParams WhereClause$^?$ HasResultType$^?$ Offers$^?$ CtorBody*	(34)

Super	::=	`extends` *ClassType*	*(35)*
FieldKeyword	::=	`val`	*(36)*
	|	`var`	
VarKeyword	::=	`val`	*(37)*
	|	`var`	
FieldDecl	::=	*Mods$^?$ FieldKeyword FieldDeclarators* `;`	*(38)*
	|	*Mods$^?$ FieldDeclarators* `;`	
Statement	::=	*AnnotationStatement*	*(39)*
	|	*ExpStatement*	
AnnotationStatement	::=	*Annotations$^?$ NonExpStatement*	*(40)*
NonExpStatement	::=	*Block*	*(41)*
	|	*EmptyStatement*	
	|	*AssertStatement*	
	|	*SwitchStatement*	
	|	*DoStatement*	
	|	*BreakStatement*	
	|	*ContinueStatement*	
	|	*ReturnStatement*	
	|	*ThrowStatement*	
	|	*TryStatement*	
	|	*LabeledStatement*	
	|	*IfThenStatement*	
	|	*IfThenElseStatement*	
	|	*WhileStatement*	
	|	*ForStatement*	
	|	*AsyncStatement*	
	|	*AtStatement*	
	|	*AtomicStatement*	
	|	*WhenStatement*	
	|	*AtEachStatement*	
	|	*FinishStatement*	
	|	*NextStatement*	
	|	*ResumeStatement*	
	|	*AssignPropertyCall*	
	|	*OfferStatement*	
OfferStatement	::=	`offer` *Exp* `;`	*(42)*
IfThenStatement	::=	`if (` *Exp* `)` *Statement*	*(43)*
IfThenElseStatement	::=	`if (` *Exp* `)` *Statement* `else` *Statement*	*(44)*

EmptyStatement	::=	;	*(45)*
LabeledStatement	::=	*Id* : *LoopStatement*	*(46)*
LoopStatement	::=	*ForStatement*	*(47)*
	|	*WhileStatement*	
	|	*DoStatement*	
	|	*AtEachStatement*	
ExpStatement	::=	*StatementExp* ;	*(48)*
StatementExp	::=	*Assignment*	*(49)*
	|	*PreIncrementExp*	
	|	*PreDecrementExp*	
	|	*PostIncrementExp*	
	|	*PostDecrementExp*	
	|	*MethodInvocation*	
	|	*ClassInstCreationExp*	
AssertStatement	::=	assert *Exp* ;	*(50)*
	|	assert *Exp* : *Exp* ;	
SwitchStatement	::=	switch (*Exp*) *SwitchBlock*	*(51)*
SwitchBlock	::=	{ *SwitchBlockStatementGroups*$^?$ *SwitchLabels*$^?$ }	*(52)*
SwitchBlockStatementGroups	::=	*SwitchBlockStatementGroup*	*(53)*
	|	*SwitchBlockStatementGroups SwitchBlockStatementGroup*	
SwitchBlockStatementGroup	::=	*SwitchLabels BlockStatements*	*(54)*
SwitchLabels	::=	*SwitchLabel*	*(55)*
	|	*SwitchLabels SwitchLabel*	
SwitchLabel	::=	case *ConstantExp* :	*(56)*
	|	default :	
WhileStatement	::=	while (*Exp*) *Statement*	*(57)*
DoStatement	::=	do *Statement* while (*Exp*) ;	*(58)*

ForStatement	::=	*BasicForStatement*	*(59)*
	|	*EnhancedForStatement*	
BasicForStatement	::=	for (*ForInit$^?$* ; *Exp$^?$* ; *ForUpdate$^?$*) *Statement*	*(60)*
ForInit	::=	*StatementExpList*	*(61)*
	|	*LocalVariableDecl*	
ForUpdate	::=	*StatementExpList*	*(62)*
StatementExpList	::=	*StatementExp*	*(63)*
	|	*StatementExpList* , *StatementExp*	
BreakStatement	::=	break *Id$^?$* ;	*(64)*
ContinueStatement	::=	continue *Id$^?$* ;	*(65)*
ReturnStatement	::=	return *Exp$^?$* ;	*(66)*
ThrowStatement	::=	throw *Exp* ;	*(67)*
TryStatement	::=	try *Block Catches*	*(68)*
	|	try *Block Catches$^?$ Finally*	
Catches	::=	*CatchClause*	*(69)*
	|	*Catches CatchClause*	
CatchClause	::=	catch (*FormalParam*) *Block*	*(70)*
Finally	::=	finally *Block*	*(71)*
ClockedClause	::=	clocked (*ClockList*)	*(72)*
AsyncStatement	::=	async *ClockedClause$^?$ Statement*	*(73)*
	|	clocked async *Statement*	
AtStatement	::=	at *PlaceExpSingleList Statement*	*(74)*
AtomicStatement	::=	atomic *Statement*	*(75)*

WhenStatement	::=	`when` (*Exp*) *Statement*	*(76)*
AtEachStatement	::=	`ateach` (*LoopIndex* `in` *Exp*) *ClockedClause*$^?$	*(77)*
		Statement	
	|	`ateach` (*Exp*) *Statement*	
EnhancedForStatement	::=	`for` (*LoopIndex* `in` *Exp*) *Statement*	*(78)*
	|	`for` (*Exp*) *Statement*	
FinishStatement	::=	`finish` *Statement*	*(79)*
	|	`clocked finish` *Statement*	
PlaceExpSingleList	::=	(*PlaceExp*)	*(80)*
PlaceExp	::=	*Exp*	*(81)*
NextStatement	::=	`next` ;	*(82)*
ResumeStatement	::=	`resume` ;	*(83)*
ClockList	::=	*Clock*	*(84)*
	|	*ClockList* , *Clock*	
Clock	::=	*Exp*	*(85)*
CastExp	::=	*Primary*	*(86)*
	|	*ExpName*	
	|	*CastExp* `as` *Type*	
TypeParamWithVarianceList	::=	*TypeParamWithVariance*	*(87)*
	|	*TypeParamWithVarianceList* , *TypeParamWithVariance*	
TypeParamList	::=	*TypeParam*	*(88)*
	|	*TypeParamList* , *TypeParam*	
TypeParamWithVariance	::=	*Id*	*(89)*
	|	+ *Id*	
	|	– *Id*	
TypeParam	::=	*Id*	*(90)*
AssignmentExp	::=	*Exp* -> *Exp*	*(91)*
ClosureExp	::=	*FormalParams* *WhereClause*$^?$ *HasResultType*$^?$	*(92)*
		Offers$^?$ => *ClosureBody*	

LastExp	::=	*Exp*	*(93)*
ClosureBody	::=	*ConditionalExp*	*(94)*
	|	*Annotations$^?$ { BlockStatements$^?$ LastExp }*	
	|	*Annotations$^?$ Block*	
AtExp	::=	at *PlaceExpSingleList ClosureBody*	*(95)*
FinishExp	::=	finish (*Exp*) *Block*	*(96)*
identifier	::=	IDENTIFIER	*(97)*
TypeName	::=	*Id*	*(98)*
	|	*TypeName . Id*	
ClassName	::=	*TypeName*	*(99)*
TypeArguments	::=	[*TypeArgumentList*]	*(100)*
TypeArgumentList	::=	*Type*	*(101)*
	|	*TypeArgumentList , Type*	
PackageName	::=	*Id*	*(102)*
	|	*PackageName . Id*	
ExpName	::=	*Id*	*(103)*
	|	*AmbiguousName . Id*	
MethodName	::=	*Id*	*(104)*
	|	*AmbiguousName . Id*	
PackageOrTypeName	::=	*Id*	*(105)*
	|	*PackageOrTypeName . Id*	
AmbiguousName	::=	*Id*	*(106)*
	|	*AmbiguousName . Id*	
CompilationUnit	::=	*PackageDecl$^?$ TypeDecls$^?$*	*(107)*
	|	*PackageDecl$^?$ ImportDecls TypeDecls$^?$*	
	|	*ImportDecls PackageDecl ImportDecls$^?$ TypeDecls$^?$*	
	|	*PackageDecl ImportDecls PackageDecl ImportDecls$^?$ TypeDecls$^?$*	
ImportDecls	::=	*ImportDecl*	*(108)*
	|	*ImportDecls ImportDecl*	

TypeDecls	::=	*TypeDecl*	*(109)*
	|	*TypeDecls TypeDecl*	
PackageDecl	::=	*Annotations$^?$* `package` *PackageName* ;	*(110)*
ImportDecl	::=	*SingleTypeImportDecl*	*(111)*
	|	*TypeImportOnDemandDecl*	
SingleTypeImportDecl	::=	`import` *TypeName* ;	*(112)*
TypeImportOnDemandDecl	::=	`import` *PackageOrTypeName* . * ;	*(113)*
TypeDecl	::=	*ClassDecl*	*(114)*
	|	*InterfaceDecl*	
	|	*TypeDefDecl*	
	|	;	
Interfaces	::=	`implements` *InterfaceTypeList*	*(115)*
InterfaceTypeList	::=	*Type*	*(116)*
	|	*InterfaceTypeList* , *Type*	
ClassBody	::=	{ *ClassBodyDecls$^?$* }	*(117)*
ClassBodyDecls	::=	*ClassBodyDecl*	*(118)*
	|	*ClassBodyDecls ClassBodyDecl*	
ClassBodyDecl	::=	*ClassMemberDecl*	*(119)*
	|	*CtorDecl*	
ClassMemberDecl	::=	*FieldDecl*	*(120)*
	|	*MethodDecl*	
	|	*PropertyMethodDecl*	
	|	*TypeDefDecl*	
	|	*ClassDecl*	
	|	*InterfaceDecl*	
	|	;	
FormalDeclarators	::=	*FormalDeclarator*	*(121)*
	|	*FormalDeclarators* , *FormalDeclarator*	
FieldDeclarators	::=	*FieldDeclarator*	*(122)*
	|	*FieldDeclarators* , *FieldDeclarator*	
VariableDeclaratorsWithType	::=	*VariableDeclaratorWithType*	*(123)*

			VariableDeclaratorsWithType , *VariableDeclarator-WithType*	
VariableDeclarators	::=	*VariableDeclarator*	*(124)*
			VariableDeclarators , *VariableDeclarator*	
VariableInitializer	::=	*Exp*	*(125)*
ResultType	::=	: *Type*	*(126)*
HasResultType	::=	: *Type*	*(127)*
			<: *Type*	
FormalParamList	::=	*FormalParam*	*(128)*
			FormalParamList , *FormalParam*	
LoopIndexDeclarator	::=	*Id HasResultType$^?$*	*(129)*
			[*IdList*] *HasResultType$^?$*	
			Id [*IdList*] *HasResultType$^?$*	
LoopIndex	::=	*Mods$^?$ LoopIndexDeclarator*	*(130)*
			Mods$^?$ VarKeyword LoopIndexDeclarator	
FormalParam	::=	*Mods$^?$ FormalDeclarator*	*(131)*
			Mods$^?$ VarKeyword FormalDeclarator	
			Type	
Offers	::=	offers *Type*	*(132)*
ExceptionTypeList	::=	*ExceptionType*	*(133)*
			ExceptionTypeList , *ExceptionType*	
ExceptionType	::=	*ClassType*	*(134)*
MethodBody	::=	= *LastExp* ;	*(135)*
			= *Annotations$^?$* { *BlockStatements$^?$ LastExp* }	
			= *Annotations$^?$ Block*	
			Annotations$^?$ Block	
			;	
CtorBody	::=	= *CtorBlock*	*(136)*
			CtorBlock	
			= *ExplicitCtorInvocation*	
			= *AssignPropertyCall*	
			;	
CtorBlock	::=	{ *ExplicitCtorInvocation$^?$ BlockStatements$^?$* }	*(137)*

Arguments	::=	(*ArgumentList*$^?$)	*(138)*
InterfaceDecl	::=	*NormalInterfaceDecl*	*(139)*
ExtendsInterfaces	::=	**extends** *Type*	*(140)*
	|	*ExtendsInterfaces* , *Type*	
InterfaceBody	::=	{ *InterfaceMemberDecls*$^?$ }	*(141)*
InterfaceMemberDecls	::=	*InterfaceMemberDecl*	*(142)*
	|	*InterfaceMemberDecls InterfaceMemberDecl*	
InterfaceMemberDecl	::=	*MethodDecl*	*(143)*
	|	*PropertyMethodDecl*	
	|	*FieldDecl*	
	|	*ClassDecl*	
	|	*InterfaceDecl*	
	|	*TypeDefDecl*	
	|	;	
Annotations	::=	*Annotation*	*(144)*
	|	*Annotations Annotation*	
Annotation	::=	@ *NamedType*	*(145)*
Id	::=	*identifier*	*(146)*
Block	::=	{ *BlockStatements*$^?$ }	*(147)*
BlockStatements	::=	*BlockStatement*	*(148)*
	|	*BlockStatements BlockStatement*	
BlockStatement	::=	*LocalVariableDeclStatement*	*(149)*
	|	*ClassDecl*	
	|	*TypeDefDecl*	
	|	*Statement*	
IdList	::=	*Id*	*(150)*
	|	*IdList* , *Id*	
FormalDeclarator	::=	*Id ResultType*	*(151)*
	|	[*IdList*] *ResultType*	
	|	*Id* [*IdList*] *ResultType*	
FieldDeclarator	::=	*Id HasResultType*	*(152)*

		| *Id HasResultType$^?$ = VariableInitializer*	
VariableDeclarator	::=	*Id HasResultType$^?$ = VariableInitializer*	*(153)*
		| *[IdList] HasResultType$^?$ = VariableInitializer*	
		| *Id [IdList] HasResultType$^?$ = VariableInitializer*	
VariableDeclaratorWithType	::=	*Id HasResultType = VariableInitializer*	*(154)*
		| *[IdList] HasResultType = VariableInitializer*	
		| *Id [IdList] HasResultType = VariableInitializer*	
LocalVariableDeclStatement	::=	*LocalVariableDecl ;*	*(155)*
LocalVariableDecl	::=	*Mods$^?$ VarKeyword VariableDeclarators*	*(156)*
		| *Mods$^?$ VariableDeclaratorsWithType*	
		| *Mods$^?$ VarKeyword FormalDeclarators*	
Primary	::=	`here`	*(157)*
		| *[ArgumentList$^?$]*	
		| *Literal*	
		| `self`	
		| `this`	
		| *ClassName* . `this`	
		| *(Exp)*	
		| *ClassInstCreationExp*	
		| *FieldAccess*	
		| *MethodInvocation*	
		| *MethodSelection*	
		| *OperatorFunction*	
OperatorFunction	::=	*TypeName* . +	*(158)*
		| *TypeName* . -	
		| *TypeName* . *	
		| *TypeName* . /	
		| *TypeName* . %	
		| *TypeName* . &	
		| *TypeName* . |	
		| *TypeName* . ^	
		| *TypeName* . <<	
		| *TypeName* . >>	
		| *TypeName* . >>>	
		| *TypeName* . <	
		| *TypeName* . <=	

	|	*TypeName* . >=	
	|	*TypeName* . >	
	|	*TypeName* . ==	
	|	*TypeName* . !=	
Literal	::=	`IntegerLiteral`	*(159)*
	|	`LongLiteral`	
	|	`UnsignedIntegerLiteral`	
	|	`UnsignedLongLiteral`	
	|	`FloatingPointLiteral`	
	|	`DoubleLiteral`	
	|	*BooleanLiteral*	
	|	`CharacterLiteral`	
	|	`StringLiteral`	
	|	`null`	
BooleanLiteral	::=	`true`	*(160)*
	|	`false`	
ArgumentList	::=	*Exp*	*(161)*
	|	*ArgumentList* , *Exp*	
FieldAccess	::=	*Primary* . *Id*	*(162)*
	|	`super` . *Id*	
	|	*ClassName* . `super` . *Id*	
	|	*Primary* . `class`	
	|	`super` . `class`	
	|	*ClassName* . `super` . `class`	
MethodInvocation	::=	*MethodName TypeArguments$^?$* (*ArgumentList$^?$*)	*(163)*
	|	*Primary* . *Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	`super` . *Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	*ClassName* . `super` . *Id TypeArguments$^?$* (*ArgumentList$^?$*)	
	|	*Primary TypeArguments$^?$* (*ArgumentList$^?$*)	
MethodSelection	::=	*MethodName* . (*FormalParamList$^?$*)	*(164)*
	|	*Primary* . *Id* . (*FormalParamList$^?$*)	
	|	`super` . *Id* . (*FormalParamList$^?$*)	
	|	*ClassName* . `super` . *Id* . (*FormalParamList$^?$*)	
PostfixExp	::=	*CastExp*	*(165)*

		PostIncrementExp	
	|	*PostDecrementExp*	
PostIncrementExp	::=	*PostfixExp* ++	*(166)*
PostDecrementExp	::=	*PostfixExp* --	*(167)*
UnannotatedUnaryExp	::=	*PreIncrementExp*	*(168)*
	|	*PreDecrementExp*	
	|	+ *UnaryExpNotPlusMinus*	
	|	– *UnaryExpNotPlusMinus*	
	|	*UnaryExpNotPlusMinus*	
UnaryExp	::=	*UnannotatedUnaryExp*	*(169)*
	|	*Annotations UnannotatedUnaryExp*	
PreIncrementExp	::=	++ *UnaryExpNotPlusMinus*	*(170)*
PreDecrementExp	::=	-- *UnaryExpNotPlusMinus*	*(171)*
UnaryExpNotPlusMinus	::=	*PostfixExp*	*(172)*
	|	˜ *UnaryExp*	
	|	! *UnaryExp*	
MultiplicativeExp	::=	*UnaryExp*	*(173)*
	|	*MultiplicativeExp* * *UnaryExp*	
	|	*MultiplicativeExp* / *UnaryExp*	
	|	*MultiplicativeExp* % *UnaryExp*	
AdditiveExp	::=	*MultiplicativeExp*	*(174)*
	|	*AdditiveExp* + *MultiplicativeExp*	
	|	*AdditiveExp* – *MultiplicativeExp*	
ShiftExp	::=	*AdditiveExp*	*(175)*
	|	*ShiftExp* << *AdditiveExp*	
	|	*ShiftExp* >> *AdditiveExp*	
	|	*ShiftExp* >>> *AdditiveExp*	
RangeExp	::=	*ShiftExp*	*(176)*
	|	*ShiftExp* .. *ShiftExp*	
RelationalExp	::=	*RangeExp*	*(177)*
	|	*SubtypeConstraint*	
	|	*RelationalExp* < *RangeExp*	
	|	*RelationalExp* > *RangeExp*	

	|	*RelationalExp* <= *RangeExp*	
	|	*RelationalExp* >= *RangeExp*	
	|	*RelationalExp* `instanceof` *Type*	
	|	*RelationalExp* `in` *ShiftExp*	
EqualityExp	::=	*RelationalExp*	*(178)*
	|	*EqualityExp* == *RelationalExp*	
	|	*EqualityExp* != *RelationalExp*	
	|	*Type* == *Type*	
AndExp	::=	*EqualityExp*	*(179)*
	|	*AndExp* & *EqualityExp*	
ExclusiveOrExp	::=	*AndExp*	*(180)*
	|	*ExclusiveOrExp* ^ *AndExp*	
InclusiveOrExp	::=	*ExclusiveOrExp*	*(181)*
	|	*InclusiveOrExp* | *ExclusiveOrExp*	
ConditionalAndExp	::=	*InclusiveOrExp*	*(182)*
	|	*ConditionalAndExp* && *InclusiveOrExp*	
ConditionalOrExp	::=	*ConditionalAndExp*	*(183)*
	|	*ConditionalOrExp* || *ConditionalAndExp*	
ConditionalExp	::=	*ConditionalOrExp*	*(184)*
	|	*ClosureExp*	
	|	*AtExp*	
	|	*FinishExp*	
	|	*ConditionalOrExp* ? *Exp* : *ConditionalExp*	
AssignmentExp	::=	*Assignment*	*(185)*
	|	*ConditionalExp*	
Assignment	::=	*LeftHandSide AssignmentOperator AssignmentExp*	*(186)*
	|	*ExpName* (*ArgumentList*$^?$) *AssignmentOperator AssignmentExp*	
	|	*Primary* (*ArgumentList*$^?$) *AssignmentOperator AssignmentExp*	
LeftHandSide	::=	*ExpName*	*(187)*
	|	*FieldAccess*	
AssignmentOperator	::=	=	*(188)*
	|	*=	
	|	/=	
	|	%=	
	|	+=	
	|	-=	

| | <<=
| | >>=
| | >>>=
| | &=
| | ^=
| | |=

Exp	::=	*AssignmentExp*	*(189)*
ConstantExp	::=	*Exp*	*(190)*
PrefixOp	::=	+	*(191)*

| | –
| | !
| | ~

| *BinOp* | ::= | + | *(192)* |

| | –
| | *
| | /
| | %
| | &
| | |
| | ^
| | &&
| | ||
| | <<
| | >>
| | >>>
| | >=
| | <=
| | >
| | <
| | ==
| | !=

References

[1] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[2] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.

[3] William Carlson, Tarek El-Ghazawi, Bob Numrich, and Kathy Yelick. Programming in the Partitioned Global Address Space Model, 2003. Presentation at SC 2003, http://www.gwu.edu/ upc/tutorials.html.

[4] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[5] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[7] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[8] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[9] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishna-murthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

A Change Log

A.1 Changes from X10 v2.0.6

This document summarizes the main changes between X10 2.0.6 and X10 2.1. The descriptions are intended to be suggestive rather than definitive; see the language specification for full details.

A.1.1 Object Model

1. Objects are now local rather than global.

   (a) The `home` property is gone.

   (b) `at(P)S` produces deep copies of all objects reachable from lexically exposed variables in S when it executes S. (**Warning:** They are copied even in `at(here)S`.)

2. The `GlobalRef[T]` struct is the only way to produce or manipulate cross-place references.

   (a) `GlobalRef`'s have a `home` property.

   (b) Use `GlobalRef[Foo](foo)` to make a new global reference.

   (c) Use `myGlobalRef()` to access the object referenced; this requires `here == myGlobalRef.home`.

3. The `!` type modifier is no longer needed or present.

4. `global` modifiers are now gone:

    (a) `global` methods in *interfaces* are now the default.

    (b) `global` *fields* are gone. In some cases object copying will produce the same effect as global fields. In other cases code must be rewritten. It may be desirable to mark nonglobal fields `transient` in many cases.

    (c) `global` *methods* are now marked `@Global` instead. Methods intended to be non-global may be marked `@Pinned`.

A.1.2 Constructors

1. `proto` types are gone.

2. Constructors and the methods they call must satisfy a number of static checks.

    (a) Constructors can only invoke `private` or `final` methods, or methods annotated `@NonEscaping`.

    (b) Methods invoked by constructors cannot read fields before they are written.

    (c) The compiler ensures this with a detailed protocol.

3. It is still impossible for X10 constructors to leak references to `this` or observe uninitialized fields of an object. Now, however, the mechanisms enforcing this are less obtrusive than in 2.0.6; the burden is largely on the compiler, not the programmer.

A.1.3 Implicit clocks for each finish

Most clock operations can be accomplished using the new implicit clocks.

1. A `finish` may be qualified with `clocked`, which gives it a clock.

2. An `async` in a `clocked finish` may be marked `clocked`. This registers it on the same clock as the enclosing `finish`.

3. `clocked async S` and `clocked finish S` may use `next` in the body of S to advance the clock.

4. When the body of a `clocked finish` completes, the `clocked finish` is dropped form the clock. It will still wait for spawned asyncs to terminate, but such asyncs need to wait for it.

A.1.4   Asynchronous initialization of val

`vals` can be initialized asynchronously. As always with `vals`, they can only be read after it is guaranteed that they have been initialized. For example, both of the `prints` below are good. However, the commented-out `print` in the `async` is bad, since it is possible that it will be executed before the initialization of `a`.

```
val a:Int;
finish {
  async {
     a = 1;
     print("a=" + a);
  }
  async {
     // WRONG: print("a=" + a);
  }
}
print("a=" + a);
```

A.1.5   Main Method

The signature for the `main` method is now:

```
def main(Array[String]) {..}
```

or, if the arguments are actually used,

```
def main(argv: Array[String](1)) {..}
```

A.1.6   Assorted Changes

1. The syntax for destructuring a point now uses brackets rather than braces: `for( [i] in 1..10 )`, rather than the prior `(i)`.

A.1.7 Safety of atomic and when blocks

1. Static effect annotations (`safe`, `sequential`, `nonblocking`, `pinned`) are no longer used. They have been replaced by dynamic checks.

2. Using an inappropriate operation in the scope of an `atomic` or `when` construct will throw `IllegalOperationException`. The following are inappropriate:

   - `when`
   - `resume()` or `next` on clocks
   - async
   - `Future.make()`, or `Future.force()`.
   - `at`

A.1.8 Removed Topics

The following are gone:

1. `foreach` is gone.

2. All `var`s are effectively `shared`, so `shared` is gone.

3. The place clause on `async` is gone. `async (P) S` should be written `at(P) async S`.

4. Checked exceptions are gone.

5. `future` is gone.

6. `await ... or ...` is gone.

7. `const` is gone.

A.1.9   Deprecated

The following constructs are still available, but are likely to be replaced in a future version:

1. `ValRail`.

2. `Rail`.

3. `ateach`

4. `offers`. The `offers` concept was experimental in 2.1, but was determined inadequate. It has not been removed from the compiler yet, but it will be soon. In the meantime, traces of it are still visible in the grammar. They should not be used and can safely be ignored.

A.2   Changes from X10 v2.0

- `Any` is now the top of the type hierarchy (every object, struct and function has a type that is a subtype of `Any`). `Any` defines `home`, `at`, `toString`, `typeName`, `equals` and `hashCode`. `Any` also defines the methods of `Equals`, so `Equals` is not needed any more.

- Revised discussion of incomplete types.

- The manual has been revised and brought into line with the current implementation.

A.3   Changes from X10 v1.7

The language has changed in the following way:

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

  Class and struct types are called *container types* in that they specify a collection of fields and methods. Container types have a name and a signature

(the collection of members accessible on that type). Collection types support primitive equality == and may support user-defined equality if they implement the `x10.lang.Equals` interface.

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

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

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

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

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

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

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

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

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

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

B   Options

B.0.1   Compiler Options

The X10 compilers have many useful options.

B.0.2   Optimization: `-O` or `-optimize`

This flag causes the compiler to generate optimized code.

B.0.3   Debugging: `-DEBUG=boolean`

This flag, if true, causes the compiler to generate debugging information. It is false by default.

B.0.4   Call Style: `-STATIC_CALLS=boolean`, `-VERBOSE_CALLS=boolean`

By default, if a method call *could* be correct but is not *necessarily* correct, the X10 compiler generates a dynamic check to ensure that it is correct before it is performed. For example, the following code:

```
def use(n:Int{self == 0}) {}
def test(x:Int) {
   use(x); // creates a dynamic cast
}
```

compiles with `-STATIC_CALLS=false`, even though it is possible that `x!=0` when `use(x)` is called. In this case, the compiler inserts a cast, which has the effect of checking that the call is correct before it happens:

```
def use(n:Int{self == 0}) {}
def test(x:Int) {
    use(x as Int{self == 0});
}
```

The compiler produces a warning that it inserted some dynamic casts. If you then want to see what it did, use -VERBOSE_CALLS.

You may also turn on static checking, with the -STATIC_CALLS flag. With static checking, calls that cannot be proved correct statically will be marked as errors. The program without the dynamic cast fails to compile with -STATIC_CALLS.

B.0.5 Help: -help and -- -help

These options cause the compiler to print a list of all command-line options.

B.0.6 Source Path: -sourcepath *path*

This option tells the compiler where to look for X10 source code.

B.0.7 Class Path: -classpath *path*

This option tells the compiler where to look for compiled code in class files.

B.0.8 Output Directory: -d *directory*

This option tells the compiler to produce its output files in the specified directory.

B.0.9 Runtime -x10rt *impl*

This option tells which runtime implementation to use. The choices are lapi, pgp, sockets, mpi, and standalone.

B.1 Execution Options: Java

The Java execution command x10 has a number of options as well.

B.1.1   `-NUMBER_OF_LOCAL_PLACES=`*`number`*

This option controls how many `Place`s the system will run on. The default is four, but you may control it as you wish.

B.1.2   Heap Size: `-mx` *`size`*

Sets the maximum size of the heap.

B.1.3   Help: `-h`

Prints a listing of all execution options.

The X10 language has been developed as part of the IBM PERCS Project, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.