Report on the Programming Language X10
Version 2.1

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

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

October 25, 2010

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

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

The X10 design team consists of Bard Bloom, David Cunningham, Stephen Fink, 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, Patrick Gallop, Christian Grothoff, Allan Kielstra, Sriram Krishnamoorthy, Bruce Lucas, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon,

Contents

3

1 Introduction

Background

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

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

One response to this problem has been to move to a *fragmented memory model*. Multiple processors are made to interact via a relatively language-neutral message-passing format such as MPI [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 processors, which is said to have *affinity* to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global *barriers* are used to ensure that processors remain roughly in lock-step.

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

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

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the PGAS model with *asynchrony* (yielding the *APGAS* programming model). X10 introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more *activities*. Activities are extremely lightweight threads of execution. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. 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 language 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 2.1.1 Normed describes entities with a norm() method. Normed is intended to be used for entities with a position in some coordinate system, and norm() gives the distance between the entity and the origin. A Slider is an object which can be moved around on a line; a PlanePoint is a fixed position in a plane. Both Sliders and PlanePoints 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);
}
```

$\square$

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);
```

They are used extensively in X10 programs. For example, the normal way to
construct an `Array[Int](1)` – that is, a fixed-length array of numbers, like an
`int[]` in Java – is to pass two arguments to a factory method: the first argument
being the length of the 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 un-
derstands 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, but an attempt to compute `axb1 + bxc` or `bxc
* axb1` would result in a compile-time type error:

```
static def example(a:Int, b:Int, c:Int) {
  val axb1 : Mat(a,b) = makeMat(a,b);
```

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

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

A constrained type looks like `Mat{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 thus possible to make a list of integers `List[Int]`, a list of non-zero integers

---

[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 *.

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

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

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

2.2 The sequential core of X10

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

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

2.3 Places and activities

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

of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of *place* (§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) can be used, to move part of the activity to q. It is easy to compute across multiple places, but the expensive operations (*e.g.*, those which require communication) are readily visible in the code.

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* (§**??**). 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 structure

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators, all of them composed of ASCII characters.

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.

Keywords X10 uses the following keywords:

abstract	false	offers	transient
as	final	operator	true
assert	finally	package	try
async	finish	private	var
ateach	for	property	when
break	goto	protected	while
case	if	public	at (*)

catch	implements	return	atomic (*)
class	import	self	await (*)
continue	in	static	clocked (*)
def	instanceof	struct	here (*)
default	interface	super	next (*)
do	native	switch	offer (*)
else	new	this	resume (*)
extends	null	throw	type (*)

Keywords marked with a (*) are *soft* keywords, which may also be used as identifiers. The other keywords may not be used as identifiers.

Note that the primitive types are not considered keywords.

Literals   Briefly, X10 v2.1 uses fairly standard syntax for its literals: integers, unsigned integers, floating point numbers, booleans, characters, strings, and null. The most exotic points are (1) unsigned numbers are marked by a u and cannot have a sign; (2) true and false are the literals for the booleans; and (3) floating point numbers are Double unless marked with an f for Float.

Less briefly, we use the following abbreviations:

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

- true and false are the Boolean literals.

- null is a literal for the null value. It has type Any{self==null}.

- Int literals have the form si; *e.g.*, 123, -321 are decimal Ints, 0123 and -0321 are octal Ints, and 0x123, -0X321, 0xBED, and 0XEBEC are hexadecimal Ints.

- Long literals have the form sil or siL. *E.g.*, 1234567890L and 0xBAGEL are Long literals.

- UInt literals have the form iu or iU. *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 sis or siS. *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 siy or siY. (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 sff or sfF. 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], sfD, and sfd. *E.g.*, 0.0, 0e100, 1.3D, 229792458d, and 314159265e-8 are Double literals.

- Char literals have one of the following forms:

    - 'c' where c is any printing ASCII character other than \ or ', representing the character c itself; *e.g.*, '!';
    - '\b', representing backspace;
    - '\t', representing tab;
    - '\n', representing newline;
    - '\f', representing form feed;
    - '\r', representing return;
    - '\'', representing single-quote;

---

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

    – `'\"'`, representing double-quote;

    – `'\\'`, representing backslash;

    – `'\dd'`, where *dd* is one or more octal digits, representing the one-byte character numbered *dd*; it is an error if *dd*> 0377.

- `String` literals consist of a double-quote `"`, followed by zero or more of the contents of a `Char` literal, followed by another double quote. *E.g.*, `"hi!"`, `""`.

Separators   X10 has the following separators and delimiters:

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

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

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

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 *arguments* and a *body*, which is an expression that can reference these arguments 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 to a set of arguments.

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), etc.

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.

4.0.4 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*	*(19.14)*
	|	*ConstrainedType*	
FunctionType	::=	*TypeParams$^?$ (FormalParamList$^?$) WhereClause$^?$*	*(19.15)*
		Offers$^?$ => Type	

The type *TypeName* refers to a defined type. 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 v2.1, 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) introduces a *class type* containing all instances of the class. The `Position` class below could describe the position of a slider control, for example.

```
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 via constructor calls. 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 (§14.2) and atomicity (§14.7) 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.

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*, public `val` instance fields that can be used in types. For example, the following code declares a class named

Coords with properties x and y and a move method. The properties are bound using the property statement in the constructor.

```
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 constraints. 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 parameters can be used as types, and will be bound to types on instantiation. For example, a generic stack class may be defined as Stack[T]{...}. Stacks can hold values of any type; *e.g.*, Stack[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, §8.4.1,§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 v2.1's C++ back end requires generic methods to be static or final; the Java back end can accomodate generic instance methods as well.

4.2.1   Variance of Type Parameters

Class, struct and interface definitions are permitted to specify a *variance* for each type parameter. There are three variance specifications: + indicates *covariance*, – indicates *contravariance* and the absence of + and – indicates *invariance*. For a class (or struct or interface) `S` specifying that a particular parameter position (say, `i`) is covariant means that if `Si <: Ti` then `S[S1,...,Sn]` `<: S[S1,..., Si-1,Ti,Si+1,... Sn]`. Similarly, saying that position `i` is is contravariant means that if `Si <: Ti` then `S[S1,..., Si-1,Ti,Si+1,... Sn]` `<: S[S1,...,Sn]`. If the position is invariant, then no such relationship is asserted between `Si <: Ti` and `S[S1,..., Si-1,Ti,Si+1,... Sn]`. The compiler must perform several checks on the body of the class (or struct or interface)

to establish that type parameters with a variance are used in a manner that is consistent with their semantics.

Limitation: The implementation of variance specifications suffers from various limitations in X10 v2.1. Users are strongly encouraged not to use variance. (Some classes, structs, and interfaces in the standard libraries use variance specifications in a careful way that circumvents these limitations.)

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*$^?$	*(19.4)*
		WhereClause$^?$ = *Type* ;	
TypeParams	::=	[*TypeParamList*]	*(19.25)*
FormalParams	::=	(*FormalParamList*$^?$)	*(19.26)*
WhereClause	::=	*DepParams*	*(19.29)*

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

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.

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]

---

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

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 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 assumes all built-in types have an infinite number of values. Therefore it does not realize that `a != b && a != c && b != c` is not satisfiable when `a`, `b`, and `c` are `Boolean` (since the `Boolean` type has only two values).

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 because `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.

4.6 Function types

FunctionType	::=	*TypeParams[?] (FormalParamList[?]) WhereClause[?]* (19.15)
		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.

$$(x_1\colon\ T_1,\ \ldots,\ x_n\colon\ T_n)\{c\}\ \Rightarrow\ T$$

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`$_1$`, ..., x`$_n$`: T`$_n$`){c} => T` is a subtype of `(x'`$_1$`: T'`$_1$`, ..., x'`$_n$`: T'`$_n$`){c'} => T'` if:

  1. Each `T`$_i$` <: T'`$_i$`;
  2. `c` 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.

- **Covariant Generic Types:** If `C` is a generic type whose ith type parameter is covariant, and `T'`$_i$` <: T`$_i$` and `T'`$_j$` == T`$_j$` for all $j \neq i$, then `C[T'`$_1$`, ..., T'`$_n$`] <: C[T'`$_1$`, ..., T'`$_n$`]`. *E.g.*, `class C[T1, +T2, T3]` with $i = 2$, and `U2 <: T2`, then `C[T1,U2,T3] <: C[T1,T2,T3]`.

- **Contravariant Generic Types:** If C is a generic type whose ith type parameter is contravariant, and T'_i <: T_i and T'_j == T_j for all $j \neq i$, then C[T'_1, ..., T'_n] :> C[T'_1, ..., T'_n]. *E.g.*, class C[T1, -T2, T3] with $i = 2$, and U2 <: T2, then C[T1,U2,T3] :> C[T1,T2,T3].

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 im-
  portant. 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.

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 v2.1 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.

---

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

4.11.1   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`.

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());
```

5 Variables

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

- **Type:** What sorts of values can be bound to the identifier;

- **Scope:** The region of code in which the identifier is associated with the entity;

- **Lifetime:** The interval of time in which the identifier is associated with the entity.

- **Visibility:** Which parts of the program can read or manipulate the value through the variable.

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

- Class variables, also known as the static fields of a class, which hold their values for the lifetime of the class.

- Instance variables, which hold their values for the lifetime of an object;

- Array elements, which are not individually named and hold their values for the lifetime of an array;

- Formal parameters to methods, functions, and constructors, which hold their values for the duration of method (etc.) invocation;

- Local variables, which hold their values for the duration of execution of a block.

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

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

In the following example, `n` is an instance variable, and `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`.

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

---

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

5.1 Immutable variables

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

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

a and b cannot be assigned to further.

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

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

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

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

5.2 Initial values of variables

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

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

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

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

Every class variable must be initialized before it is read, through the execution of an explicit initializer. Every instance variable must be initialized before it is read, through the execution of an explicit or implicit initializer or a constructor. Implicit initializers initialize `var`s to the default values of their types (§4.5). 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 [5, § 16].

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. (Future versions of X10 may allow destructuring of other types as well.) A point is a sequence of $r \geq 0$ `Int`-valued coordinates. It is often useful to get at the coordinates directly, in variables.

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.
```

A useful idiom for iterating over a range of numbers is:

```
var sum : Int = 0;
for ([i] in 1..100) sum += i;
```

The brackets in `[i]` introduce destructuring, making X10 treat `i` as an `Int`; without them, it would be a `Point`.

In general, a pattern of the form `[i`$_1$`,...,i`$_n$`]` matches a point with n coordinates, binding `i`$_j$ to coordinate j. A pattern of the form `p[i`$_1$`,...,i`$_n$`]` does the same, but also binds `p` to the point.

5.4   Formal parameters

FormalParams	::=	`(` *FormalParamList*$^?$ `)`	*(19.26)*
FormalParamList	::=	*FormalParam*	*(19.128)*
		FormalParamList `,` *FormalParam*	
FormalParam	::=	*Mods*$^?$ *FormalDeclarator*	*(19.131)*
		Mods$^?$ *VarKeyword FormalDeclarator*	
		Type	
FormalDeclarators	::=	*FormalDeclarator*	*(19.121)*
		FormalDeclarators `,` *FormalDeclarator*	
FormalDeclarator	::=	*Id ResultType*	*(19.151)*
		`[` *IdList* `]` *ResultType*	
		Id `[` *IdList* `]` *ResultType*	
ResultType	::=	`:` *Type*	*(19.126)*

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

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`. For example, the following gives `t` the type `Tub{self.p==3}` as desired:

```
        val t <: Tub = new Tub(3);
```

However, replacing `Tub` by `Int` would result in a compilation error.

Variables do not need to be initialized at the time of definition – not even `val`s. They must be initialized by the time of use, and `val`s may only be assigned to once. The X10 compiler performs static checks guaranteeing this restriction. The following is correct, albeit obtuse:

```
static def main(r: Array[String](1)):void {
  val a : Int;
```

```
a = r.size;
val b : String;
if (a == 5) b = "five?"; else b = "" + a + " args";
// ...
```

5.6   Fields

FieldDeclarators	::=	*FieldDeclarator*	*(19.122)*
	|	*FieldDeclarators* , *FieldDeclarator*	
FieldDecl	::=	*Mods$^?$ FieldKeyword FieldDeclarators* ;	*(19.38)*
	|	*Mods$^?$ FieldDeclarators* ;	
FieldDeclarator	::=	*Id HasResultType*	*(19.152)*
	|	*Id HasResultType$^?$ = VariableInitializer*	
HasResultType	::=	: *Type*	*(19.127)*
	|	<: *Type*	
FieldKeyword	::=	val	*(19.36)*
	|	var	
Mod	::=	abstract	*(19.2)*
	|	*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,global, or property; see §8.2 and §8.3. 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` files, like `var` local variables, must be declared with a type.

6 Names and packages

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

6.1 Packages

A package is a named collection of top-level type declarations, *viz.*, class, interface, and struct declarations. Package names are sequences of identifiers, like `x10.lang` and `com.ibm.museum`. The multiple names are simply a convenience, though there is a tenuous relationship between packages `a`, `a.b`, and `a.c`.

Packages 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`.

6.1.1   Name Collisions

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

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

6.2   `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.

6.2.1   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.

6.2.2 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.

6.2.3 Implicit Imports

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

6.3 Conventions on Type Names

TypeName	::=	*Id*	*(19.98)*
	|	*TypeName* **.** *Id*	
PackageName	::=	*Id*	*(19.102)*
	|	*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 public methods, properties, `static vals`, and an invariant. It may extend several interfaces, giving X10 a large fraction of the power of multiple inheritance at a tiny fraction of the cost.

The following puny example illustrates all these features:

```
interface Pushable(text:String, prio:Int) {
  def push(): void;
  static val MAX_PRIO = 100;
}
class MessageButton(text:String, prio:Int)
  implements Pushable{self.prio==Pushable.MAX_PRIO} {
  public def push() {
    x10.io.Console.OUT.println(text + " pushed");
  }
}
```

`Pushable` defines two properties, a method, and a static value. `MessageButton` implements a constrained version of `Pushable`, *viz.* one with maximum priority. It also has `Pushable`'s properties. 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 properties that the interface requires.

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

NormalInterfaceDecl	::=	*Mods$^?$* `interface` *Id TypeParamsWithVariance$^?$*	*(19.11)*
		Properties$^?$ WhereClause$^?$ ExtendsInterfaces$^?$ InterfaceBody	
TypeParamsWithVariance	:=	[*TypeParamWithVarianceList*]	*(19.24)*
Properties	::=	(*PropertyList*)	*(19.5)*
PropertyList	::=	*Property*	*(19.6)*
	|	*PropertyList* , *Property*	
Property	::=	*Annotations$^?$ Id ResultType*	*(19.7)*
WhereClause	::=	*DepParams*	*(19.29)*
ExtendsInterfaces	::=	`extends` *Type*	*(19.140)*
	|	*ExtendsInterfaces* , *Type*	
InterfaceBody	::=	{ *InterfaceMemberDecls$^?$* }	*(19.141)*
InterfaceMemberDecls	::=	*InterfaceMemberDecl*	*(19.142)*
	|	*InterfaceMemberDecls InterfaceMemberDecl*	
InterfaceMemberDecl	::=	*MethodDecl*	*(19.143)*
	|	*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.

Each interface implicitly defines a nullary getter method `def p(): T` for each property `p: T`. The interface may not have another definition of a method `p()`.

A class `C` is said to implement an interface `I` if

- `I`, or a subtype of `I`, appears in the `implements` list of `C`,

- `C`'s properties include all the properties 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`.

Note that, by definition of $inv(\cdot)$ (see §8.7, `C`'s class invariant $inv(C)$ implies $inv(I)$.

7.1   Field Definitions

An interface may declare a `val` field, with a value. This field is implicitly `public static val`:

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

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

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

7.1.1   Fine Points of Fields

It can happen that two parent interfaces give fields of the same name. In that case, 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;
}
```

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, may implement multiple interfaces, may have static and instance fields, may have static and instance methods, may have constructors, may have static and instance nested classes and interfaces. X10 does not permit mutable static state.

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

An object exists in a single location: the place that it was created. One place cannot 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, properties, a constraint (the *class invariant*), a single superclass, zero or more interfaces, and a class body containing the the definition of fields, properties, methods, and member types. Each such declaration introduces a class type (§4.1).

NormalClassDecl	::=	*Mods$^?$* class *Id TypeParamsWithVariance$^?$*	*(19.32)*
		Properties$^?$ WhereClause$^?$ Super$^?$ Interfaces$^?$	
		ClassBody	
TypeParamsWithVariance	::=	[*TypeParamWithVarianceList*]	*(19.24)*
TypeParamWithVarianceList	::=	*TypeParamWithVariance*	*(19.87)*
	|	*TypeParamWithVarianceList* , *TypeParamWithVariance*	
TypeParamWithVariance	::=	*Id*	*(19.89)*
	|	*+ Id*	
	|	*− Id*	
Properties	::=	(*PropertyList*)	*(19.5)*
PropertyList	::=	*Property*	*(19.6)*
	|	*PropertyList* , *Property*	
Property	::=	*Annotations$^?$ Id ResultType*	*(19.7)*
WhereClause	::=	*DepParams*	*(19.29)*
DepParams	::=	{ *ExistentialList$^?$ Conjunction$^?$* }	*(19.23)*
Super	::=	extends *ClassType*	*(19.35)*
Interfaces	::=	implements *InterfaceTypeList*	*(19.115)*
InterfaceTypeList	::=	*Type*	*(19.116)*
	|	*InterfaceTypeList* , *Type*	
ClassBody	::=	{ *ClassBodyDecls$^?$* }	*(19.117)*
ClassBodyDecls	::=	*ClassBodyDecl*	*(19.118)*
	|	*ClassBodyDecls ClassBodyDecl*	
ClassBodyDecl	::=	*ClassMemberDecl*	*(19.119)*
	|	*CtorDecl*	
ClassMemberDecl	::=	*FieldDecl*	*(19.120)*
	|	*MethodDecl*	
	|	*PropertyMethodDecl*	
	|	*TypeDefDecl*	
	|	*ClassDecl*	
	|	*InterfaceDecl*	
	|	;	

8.2   Fields

Objects may have *instance fields*, or simply *fields*: 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 objects. See §8.5 for more information.

No two fields of the same class may have the same name.

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

```
class Crash {
  val f : (Int) => Boolean = (Int)=>true;
  def f(Int) = false;
}
```

Then `crash.f(3)` might either mean "call the function `crash.f` on argument 3", or "invoke the method `f` on argument 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`.

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

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

```
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 foo()
         = f            // 5
         + super.f    // 4
         + B.this.f   // 4 (the "f" of the outer instance)
         + B.super.f; // 3 (the "super.f" of the outer instance)
   }
}
```

8.2.3 Field qualifiers

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

`static` qualifier

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

`transient` **Qualifier**

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

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

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

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

---

[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.

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

It also defines a nullary getter method,

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

(As noted in §7, interfaces can define properties too. They define the same nullary getter methods, though they do not require fields.)

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

Properties are initialized by the invocation of a special `property` statement, which must be performed by each constructor of the class:

```
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.

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`.

8.4 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, 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*$^?$	*(19.3)*
	|	*MethMods* `property`	
	|	*MethMods Mod*	
MethodDecl	::=	*MethMods* `def` *Id TypeParams*$^?$ *FormalParams*	*(19.8)*
		WhereClause$^?$ *HasResultType*$^?$ *Offers*$^?$ *MethodBody*	
TypeParams	::=	[*TypeParamList*]	*(19.25)*
FormalParams	::=	(*FormalParamList*$^?$)	*(19.26)*
FormalParamList	::=	*FormalParam*	*(19.128)*
	|	*FormalParamList* , *FormalParam*	
HasResultType	::=	: *Type*	*(19.127)*
	|	<: *Type*	
MethodBody	::=	= *LastExp* ;	*(19.135)*
	|	= *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 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}` 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 exists only for those instances of `C` which satisfy `c`. It is illegal for code to invoke the method on objects whose static type is not a subtype of `C{c}`.

STATIC SEMANTICS RULE: The compiler checks that every method invocation `o.m(e`$_1$`, ..., e`$_n$`)` 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.

8.4.2 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.

The expression may contain invocations of other property methods. It is the re-
sponsibility of the programmer to ensure that the evaluation of a property ter-
minates 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.

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

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

and

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

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

8.4.3   Method overloading, overriding, hiding, shadowing and obscuring

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

Two or more methods of a class or interface may have the same name if they have a different number of type parameters, or they have formal parameters of different types. *E.g.*, the following is legal:

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

X10 v2.1 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.

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 *the same signature* if (a) they have the same number of type parameters, (b) they have the same number of formal (value) parameters, and (c) for each formal parameter the constraint erasure of its types are equivalent. It is a compile-time error for there to be two methods with the same name and same signature in a class (either defined in that class or in a superclass).

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

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

if it is the case that the constraint erasures of the types $T_1, \ldots, T_n$ are equivalent to the constraint erasures of the types $S_1, \ldots, T_n$ respectively.

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

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

A class C inherits from its direct superclass and superinterfaces all their methods visible according to the access modifiers of the superclass/superinterfaces that are not hidden or overridden. A method 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 Static initialization

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

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

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

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

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

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

```
    return C.f;
}
```

If all these activities terminate normally, all static 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.

8.6   User-Defined Operators

It is often convenient to have methods named by symbols rather than words. For example, suppose that we wish to define a `Poly` class of polynomials – for the sake of illustration, single-variable polynomials with `Int` coefficients. It would be very nice to be able to manipulate these polynomials by the usual operations: + to add, * to multiply, - to subtract, and `p(x)` to compute the value of the polynomial at argument `x`. We would like to write code thus:

```
public static def main(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.

8.6.1 Binary Operators

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

```
class Poly {
  public 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]) + 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]) + this;
```

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

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

This requires the following grammar:

MethodDecl	::=	*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*

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

operation is not uniquely determined, the compiler will report a static error.

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

8.6.2   Unary Operators

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

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

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

8.6.3   Type Conversions

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

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

8.6.4   Implicit Type Coercions

You may also define *implicit* type coercions to `T{c}` as static operators in class T. The syntax for this is `static operator (x:U) : T{c} = e`. Implicit coercions are used automatically by the compiler.

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]);

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));
}
```

8.6.5 `set` and `apply`

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

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

`a(i)=b` is short for the method call `a.set(b,i)`, with one or more indices `i`. (This has a possibly surprising consequence for the order of evaluation: in `a(i)=b`, as in `a.set(b,i)`, a is evaluated first, then b, and finally i.) Again, it is possible to overload `set` to provide a variety of subscripting operations. Each `set` method must have a corresponding `apply` method; that is, `a(i,j)=b` is only defined when `a(i,j)` is defined, despite the fact that `a(i,j)=b` does not evaluate `a(i,j)`.

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

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

8.7 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 `Pt`s[2]

```
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

---

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

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`$_n$
    `&& 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(`T`[`o`/`this`]) $\land$ `c`[`o`/`self`].

8.7.1   Invariants for `implements` and `extends` clauses

Consider a class definition

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

Each of the following static semantics rules must be satisfied:

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

STATIC SEMANTICS RULE (Super-extends): The return type c of each constructor in a class C must entail the invariant inv(C).

8.7.2 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(C), the class invariant associated with C (§8.7). 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* | *(19.34)* |
| | | *WhereClause$^?$ HasResultType$^?$ Offers$^?$ CtorBody* | |

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

Example 8.7.1 Here is another example, constructed as a simplified version of `x10.array.Region`. The `mockUnion` method has the type 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]);
    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)` (intended to represent the set of points in the rectangular parallelopiped between the origin and the `diag`.)

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

$\square$

STATIC SEMANTICS RULE (Super-invoke): Let C be a class with properties p_1: P_1, ..., p_n: P_n, invariant `c` extending the constrained type `D{d}` (where D is the name of a class).

For every constructor in C the compiler checks that the call to super invokes a constructor for D whose return type is strong enough to entail `d`. Specifically, if the call to super is of the form `super(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: $S_i \vdash x_i$: T_i (for $i \in \{1, \ldots, k\}$)
> x_1: S_1, ..., x_k: $S_k \vdash$ `c`
> d_1`[a/self]`, x_1: S_1, ..., x_k: $S_k \vdash$ `d[a/self]`

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

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

> p_1: T_1, ..., p_n: $T_n \vdash$ `t(C)` $\wedge$ `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.)

STATIC SEMANTICS RULE (Constructor invocation): 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.

8.7.3 Object Initialization

X10 does object initialization safely. It avoids a few classes of bad things:

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

2. `this` escaping from a constructor;

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.

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.

8.7.4 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`.[3]

---

[3]This is abominable behavior for `toString`, but nonetheless it is allowed.

```
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;
  }
}
```

8.7.5 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 second 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 omitted. 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 example, 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` object. (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 Classses

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 are valid if they were invoked (or inlined) at the place of creation. For example, `closure` below is acceptable because it only refers to fields defined at the point it was written. `badClosure` would not be acceptable, because it refers to fields that were not defined at that point, although they were defined later.

```
class C {
  val a = 3;
  val closure = () => a*10; // This is OK
  //val badClosure = () => b*10;
  val b = 4;
}
```

8.7.6   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 §8.7.5.

8.7.7   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	yes	no	yes	no
Constructor guard	(K)	no	yes	no	no	no
Constructor ret. type	(L)	yes	yes	yes	yes	no
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)
}
```

8.8   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)`.

```
class Resolve {
  static def zap(Object) = "Object";
  static def zap(Resolve) = "Resolve";
  public static def main(argv:Array[String](1)) {
    Console.OUT.println(zap(1..4));
    Console.OUT.println(zap(new Resolve()));
  }
}
```

The basic concept of method resolution is quite straightforward:

1. List all the methods that could possibly be used;

2. Pick the most specific one;

3. Fail if there is not a unique most specific one.

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.

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.

1. The method name `m`;

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

3. The types $x_1:T_1, \ldots,\ x_f:T_f$ 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.[4]

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';

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

3. The names and types $x_1:T'_1, \ldots,\ x_f:T'_f$ 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

---

[4]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})=....

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 m';

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

    - 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: T'_i `<:` Ti_i for each i.

    - 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; otherwise, it returns false.

8.9   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. 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;
  }
}
```

8.10   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 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");
    }
  }
}
```

8.10.1   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:

```
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();
  }
}
```

8.11   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.

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 referenced at all.

- Anonymous classes cannot be `abstract`. Since they only exist in combination 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 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

A struct declaration has the structure:

```
StructModifiers?
struct C[X₁, ..., Xₙ](p₁:T₁, ..., pₙ:Tₙ){c}
    implements I₁, ..., Iₖ {
StructBody
}
```

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`, but without prefixing it with `new`:

```
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);
}
```

Structs support the same notions of generics, properties, and constrained types that classes do. For example, the `Pair` type below provides pairs of values; the `diag()` method applies only when the two elements of the pair are equal, and returns that common value:

```
struct Pair[T,U](t:T, u:U) {
  def this(t:T, u:U) { property(t,u); }
  def diag(){T==U && t==u} = t;
}
```

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
```

9.5   Generic programming with structs

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

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

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

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

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.

```
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():String {
        return "(" + first + ", " + second + ")";
```

```
      }
  }
```

9.7 Nested Structs

Static nested structs may be defined, essentially as static nested classes except for making them structs (§8.9). Inner structs may be defined, essentially as inner classes except making them structs (§8.10).

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 forbid it. `new Class2Struct(3)` will need to be converted to `Class2Struct(3)`, and `Struct2Class(3)` 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 asserts 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.

It is a limitation of X10 v2.1 that functions do not support type arguments. This limitation may be removed in future versions of the language.

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

Limitation: Function literals do not currently support guards.

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

```
{
  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 doesn't quite work if the body is a statement rather than an expression. A few language features are forbidden (`break` or `continue` of a loop that surrounds the function literal) or mean something different (`return` inside a function returns from the function).

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

_____

[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.

10.2 Function Literals

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

ClosureExp	::=	*FormalParams WhereClause? HasResultType?*	*(19.92)*
		Offers? => ClosureBody	
FormalParams	::=	(*FormalParamList?*)	*(19.26)*
WhereClause	::=	*DepParams*	*(19.29)*
DepParams	::=	{ *ExistentialList? Conjunction?* }	*(19.23)*
ExistentialList	::=	*FormalParam*	*(19.30)*
	|	*ExistentialList* ; *FormalParam*	
HasResultType	::=	: *Type*	*(19.127)*
	|	<: *Type*	
ClosureBody	::=	*ConditionalExp*	*(19.94)*
	|	*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: (Int)=>7 is the integer function returning 7 for all inputs.

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

The body of the function is evaluated when the function is invoked by a call expression (§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-defined; 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 $(x_1: T_1, \ldots, x_n: T_n)\{c\} => \{ s \}$ the types T_i, the guard `c` and the body `s` may access many, though not all, sorts of variables from outer scopes. Specifically, they can access:

- All fields of the enclosing object and class;

- All type parameters;

- All `val` variables;

`var` variables 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.

For example, 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. For 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;
  def example() {
    val digit = new Span(0,9);
    val isDigit : (Int) => Boolean = digit.between.(Int);
    if (isDigit(8)) Console.OUT.println("8 is!");
  }
}
```

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	::=	*Primary . Id . (FormalParamList$^?$)*	*(19.164)*
	|	*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.(T`$_1$`, ..., T`$_n$`)`

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 `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`.

---

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

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` | *(19.157)* |
| | | | *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. For 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

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*	*(19.146)*

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. `n` in the second line below is a local variable expression.

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

11.4   Field access

FieldAccess	::=	*Primary* . *Id*	*(19.162)*
		| `super` . *Id*	
		| *ClassName* . `super` . *Id*	

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 sub-stituted for `this` in the type.

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

If the field target is given by the keyword `super`, the target's type is the super-class 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. The following example illustrates all four cases:

```
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 expDotId() = this.f; // 4
    def superDotId() = super.f; // 3
    def classNameDotId() = Uncle.f; // 1;
    def cnDotSuperDotId() = Ego.super.f; // 2
  }
}
```

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.

11.5   Function Literals

Function literals are described in §10.

11.6   Calls

MethodInvocation	::=	*MethodName TypeArguments$^?$ (ArgumentList$^?$)*	*(19.163)*
	|	*Primary . Id TypeArguments$^?$ (ArgumentList$^?$)*	
	|	`super` *. Id TypeArguments$^?$ (ArgumentList$^?$)*	
	|	*ClassName .* `super` *. Id TypeArguments$^?$ (ArgumentList$^?$)*	
	|	*Primary TypeArguments$^?$ (ArgumentList$^?$)*	
ArgumentList	::=	*Exp*	*(19.161)*
	|	*ArgumentList , Exp*	

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

The syntax is ambiguous; the target must be type-checked to determine if it is the name of a method or if it refers to a field containing a closure. It is a static error if a call may resolve to both a closure call or to a method call.

```
class Callsome {
  static val closure = () => 1;
  static def method () = 2;
  static val closureEvaluated = Callsome.closure();
  static val methodEvaluated = Callsome.method();
}
```

However, adding a static method called `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();
```

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

Method selection rules are given in §8.8.

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

```
class DivideBy(denom:Int) {
  def doIt(numer:Int){denom != 0} = numer / denom;
  def example() {
     //ERROR: denom might be zero: this.doIt(100);
     (this as DivideBy{self.denom != 0}).doIt(100);
  }
}
```

11.7   Assignment

Assignment	::=	*LeftHandSide AssignmentOperator AssignmentExp*	*(19.186)*
	|	*ExpName* (*ArgumentList$^?$*) *AssignmentOperator AssignmentExp*	
	|	*Primary* (*ArgumentList$^?$*) *AssignmentOperator AssignmentExp*	
LeftHandSide	::=	*ExpName*	*(19.187)*
	|	*FieldAccess*	
AssignmentOperator	::=	=	*(19.188)*
	|	*=	
	|	/=	
	|	%=	
	|	+=	
	|	-=	
	|	<<=	
	|	>>=	
	|	>>>=	
	|	&=	
	|	^=	
	|	|=	

The assignment expression `x = e` assigns a value given by expression `e` to a variable `x`. Most often, `x` is a mutable (`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;
    // Incorrect: 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 syntactic sugar for a method call: `a.set(v,i`$_1$`,...,i`$_n$`)`. Naturally, it is a static error if no suitable `set` method exists for `a`.

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

The subscripting forms of `a(i)` $\diamond$`= 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 `a.apply(i)` and one to `a.set(i,c)`, the desugared forms of `a(i)` and `a(i)=c`. 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` returns the *new* value of x, after incrementing. `x++` returns the *old* value of `, before incrementing.`

Limitation: This currently only works for numeric types.

11.9   Numeric Operations

Numeric types (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, 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 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.

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.

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 ^ 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`; other-
wise, the operator evaluates to `false`.

The `|` operator evaluates to `false` if both of its operands evaluate to `false`; oth-
erwise, 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`; oth-
erwise, 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 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* ? *Exp* : *ConditionalExp* | *(19.184)* |

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.

For 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.

11.20   Stable equality

EqualityExp	::=	*RelationalExp*	*(19.178)*
	|	*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.

- If the operands have numeric types, they are coerced into the larger of the two types (see §**??**) and then compared for numeric equality.

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

The predicates == and != may not be overridden by the programmer. Note that a==b is a form of *stable equality*; that is, the result of the equality operation is not affected by the mutable state of the program, after evaluation of a and b.

11.21   Allocation

ClassInstCreationExp	::=	new *TypeName* *TypeArguments*$^?$ (*ArgumentList*$^?$) *ClassBody*$^?$	*(19.12)*
	|	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).

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. 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 uses function application syntax, not `new`. 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	::=	*CastExp* `as` *Type*	*(19.86)*

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 precisely that type.

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

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

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	::=	*RelationalExp* `instanceof` *Type*	*(19.177)*

In the above expression, *Type* is any type. At run time, the result of this operator is `true` if the *RelationalExpression* can be coerced to *Type* without a `TypeCastException` being thrown or static error occurring. 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

SubtypbeConstraint	::=	*Type* <: *Type*	*(19.28)*
	|	*Type* :> *Type*	

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

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

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

Subtyping expressions are 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*	*(19.177)*

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

11.26   Array Constructors

Primary	::=	[*ArgumentList$^?$*]	*(19.157)*
ClassInstCreationExp	::=	new *TypeName* [*Type*] [*ArgumentList$^?$*]	*(19.12)*

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 `[e1,e2,e3, ..., en]` produces an n-element `Array[T](1)`, where `T` is the least common supertype of the **base types** of the expressions `ei`. For 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 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.

Still, 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, X10 provides an array constructor which does allow specification of the element type: `new Array[T][e1...en]`. Each element `ei` must be of type `T`. The resulting array is of type `Array[T](1)`.

```
val zero <: Array[Int{self == 0}](1) = new Array[Int{self == 0}][0];
val non1 <: Array[Int{self != 1}](1) = new Array[Int{self != 1}][0];
val eels <: Array[Eel{self != null}](1) =
    new Array[Eel{self != null}][ new Eel() ];
```

11.27 Coercions and conversions

X10 v2.1 supports the following coercions and conversions.

11.27.1 Coercions

CastExp	::=	*CastExp* `as` *Type*	*(19.86)*

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 subtype may be implicitly coerced to any supertype.

Explicit coercion (casting with `as`) An object of any class may be explicitly coerced to any other class type using the `as` operation. If `Child <: Person` and `rhys:Child`, then

```
rhys as Person
```

is an expression of type `Person`.

If the value coerced is not an instance of the target type, a `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

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`, `2147483647`.

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

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

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

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 conversion operators from type A *to* a container type B by specifying a method on B as follows:

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

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

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

```
public operator (r: Array[int](1)): Point(r.length) = make(r);
```

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

Example 11.27.1 Consider the following code:

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

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

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. It is useful when a loop header is evaluated for its side effects. For example, the following code sums the elements of a 1-dimensional array.

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

12.2   Local variable declaration

LocalVariableDeclStatement	::=	*LocalVariableDecl* ;	*(19.155)*
LocalVariableDecl	::=	*Mods$^?$ VarKeyword VariableDeclarators*	*(19.156)*
	|	*Mods$^?$ VariableDeclaratorsWithType*	
	|	*Mods$^?$ VarKeyword FormalDeclarators*	
VariableDeclaratorsWithType	::=	*VariableDeclaratorWithType*	*(19.123)*
	|	*VariableDeclaratorsWithType , VariableDeclarator-WithType*	
VariableDeclarators	::=	*VariableDeclarator*	*(19.124)*
	|	*VariableDeclarators , VariableDeclarator*	
VariableInitializer	::=	*Exp*	*(19.125)*
FormalDeclarators	::=	*FormalDeclarator*	*(19.121)*
	|	*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.
```

12.3   Block statement

Block	::=	{ *BlockStatements$^?$* }	*(19.147)*
BlockStatements	::=	*BlockStatement*	*(19.148)*
	|	*BlockStatements BlockStatement*	
BlockStatement	::=	*LocalVariableDeclStatement*	*(19.149)*
	|	*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* ;	*(19.48)*
StatementExp	::=	*Assignment*	*(19.49)*
	|	*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.

```
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.

12.5   Labeled statement

LabeledStatement	::=	*Id* **:** *LoopStatement*	*(19.46)*
LoopStatement	::=	*ForStatement*	*(19.47)*
	|	*WhileStatement*	
	|	*DoStatement*	
	|	*AtEachStatement*	

Loop statements (`for`, `while`, `do`, `ateach`) may be labeled. The label may be used as the target of a break or continue statement. The scope of a label is the statement labeled.

```
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;
   }
}
```

12.6   Break statement

BreakStatement	::=	break *Id$^?$* **;**	*(19.64)*

An unlabeled break statement exits the currently enclosing loop or switch statement. An labeled break statement exits the enclosing loop or switch statement with the given label. It is illegal to break out of a loop not defined in the current method, constructor, initializer, or closure.

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

```
var found: Boolean = false;
```

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

12.7   Continue statement

ContinueStatement	::=	continue $Id^?$;	*(19.65)*

An unlabeled `continue` skips the rest of the current iteration of the innermost enclosing 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*	*(19.43)*
IfThenElseStatement	::=	if (*Exp*) *Statement* else *Statement*	*(19.44)*

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

As is traditional in languages derived from Algol, the if-statement is 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*	*(19.51)*
SwitchBlock	::=	{ *SwitchBlockStatementGroups*? *SwitchLabels*? }	*(19.52)*
SwitchBlockStatementGroups	::=	*SwitchBlockStatementGroup*	*(19.53)*
	|	*SwitchBlockStatementGroups* *SwitchBlockStatementGroup*	
SwitchBlockStatementGroup	::=	*SwitchLabels BlockStatements*	*(19.54)*
SwitchLabels	::=	*SwitchLabel*	*(19.55)*
	|	*SwitchLabels SwitchLabel*	
SwitchLabel	::=	`case` *ConstantExp* `:`	*(19.56)*
	|	`default :`	

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

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

The index expression must be of type `Int`. Case labels must be of type `Int`, `Byte`, `Short`, or `Char` and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

In the following example, 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*	*(19.57)*

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.

```
while (n > 1) {
   n = (n % 2 == 1) ? 3*n+1 : n/2;
}
```

12.11   Do–while statement

DoStatement	::= do *Statement* while (*Exp*) ;	*(19.58)*

A do-while statement executes a 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*	*(19.59)*
	|	*EnhancedForStatement*	
BasicForStatement	::=	`for` (*ForInit$^?$* ; *Exp$^?$* ; *ForUpdate$^?$*) *Statement*	*(19.60)*
ForInit	::=	*StatementExpList*	*(19.61)*
	|	*LocalVariableDecl*	
ForUpdate	::=	*StatementExpList*	*(19.62)*
StatementExpList	::=	*StatementExp*	*(19.63)*
	|	*StatementExpList* , *StatementExp*	
EnhancedForStatement	::=	`for` (*LoopIndex* `in` *Exp*) *Statement*	*(19.78)*
	|	`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`. `for(init; test; step)`body is equivalent to:

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

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

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

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

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

An enhanced for statement is used to iterate over a collection, 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; in practice, this means that `for ([i] in 1..10)` iterates over numbers from 1 to 10, while `for (i in 1..10` iterates over `Point`s from 1 to 10). Each iteration of the loop binds the iteration variable to another element of the collection.

```
var sum : Int = 0;
for ([i] in 1..n) sum += i;
```

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 a `DistArray` iterates over the `Point`s at which the distributed array is defined.

12.13   Return statement

ReturnStatement	::=	`return` *Exp*$^?$ `;`	*(19.66)*

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

12.14   Assert statement

AssertStatement	::=	`assert` *Exp* `;`	*(19.50)*
	|	`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.

For example,

```
class Example {
  public static def main(argv:Array[String](1)) {
    val a = 1;
    assert a != 1 : "Changed my mind about a";
  }
}
```

compiles properly, but 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* ;	*(19.67)*

throw E throws an exception whose value is E, which must be an instance of a subtype of x10.lang.Throwable.

For example, the following code checks if an index is in range and throws an exception if not.

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

12.17   Try–catch statement

TryStatement	::=	try *Block Catches*	*(19.68)*
	|	try *Block Catches*$^?$ *Finally*	
Catches	::=	*CatchClause*	*(19.69)*
	|	*Catches CatchClause*	
CatchClause	::=	catch (*FormalParam*) *Block*	*(19.70)*
Finally	::=	finally *Block*	*(19.71)*

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.

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 `Places.MAX_PLACES-1`, 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 §14.5.

151

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, due to place-shifting operations. In the following code, `here` has one value at `h0`, and a different one at `h1`.

```
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);
        }
    }
}
```

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* (§14.5).

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

14.1   The X10 rooted exception model

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

The presence of a root activity 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 §14.4) awaiting the termination of B (and possibly other activities). For every X10 computation, the *root-of* relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

---

[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 `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.

AtStatement	::=	*at PlaceExpSingleList Statement*	*(19.74)*
AtExp	::=	*at PlaceExpSingleList ClosureBody*	*(19.95)*

An activity executing statement `at (q) S` at a place `p` evaluates `q` at `p` and then moves to `q` to execute `S` as follows. Note that the value obtained by evaluating `q` is not necessarily distinct from `p` (e.g. `q` may be `here`).

First, the runtime identifies the object graph at `p` reachable from lexically scoped initialized variables (say, `v1`,..., `vn`) referenced in `S`. Second this graph is *serialized* into a buffer and transmitted to place `q`. Third, the variables `v1`,..., `vn` are re-created at `q` and initialized with the objects obtained by deserializing the buffer at `q`. Fourth, `S` is executed at `q` (in an environment in which the variable `vi` refers to the re-created variable at `q`).

On normal termination of `S` control returns to `p` and execution is continued with the statement following `at (q) 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 (e) S` throws `E1`.

Since `at(e) S` is a synchronous constructs 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.

Limitation: In `at(e) S`, `S` is permitted to access only those immutable variables in the surrounding scope that are definitely initialized.

---

[2]In X10 v2.1 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.

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]` 14.2.2 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.

14.2.1   Discussion

`at(p)S` does *not* start a new activity. It should be thought of as transporting the current activity to `p`, running S there, and then transporting it back. `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) S` (spawn an activity locally to execute S at p; here p is evaluated by the spawned activity), (2) `at(p) async S` (evaluate p and then at p spawn an activity to execute S), and, (3) `async at(p) async S`.

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

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

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

14.2.2 Implicit copying from at

`at(p)S` copies nearly all data that `S` might reference, 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. 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 `ob{$'$}` 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) {                // (2)
   assert(c() == 9);       // (3)
   c.set(8);               // (4)
   assert(c() == 8);       // (5)
}
assert(c() == 9);          // (6)
```

The `at` statement implicitly copies the `Cell` and its contents. After `(1)`, `c` is a `Cell` containing 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 variables, 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[Cell[Int]][c,c];
assert(a(0)() == 5 && a(1)() == 5);      // (A)
c.set(6);                                // (B)
assert(a(0)() == 6 && a(1)() == 6);      // (C)
at(here) {
  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)
```

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) {
         assert(a == 1 && b == 0);
     }
   }
}
```

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 uniqe 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) {
   use(href);
  }
}
```

Values protected in `GlobalRef`s can be retrieved by the g.apply() method, often abbreviated simply g(). g.apply() 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()) use(argref);
}
static def use(argref : GlobalRef[Array[String](1)]) {
  at(argref.home) {
    val argv = argref();
    argv(0) = "Hi!";
  }
}
```

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. 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.

- 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.

`at` and Variables

`var` variables can be set at the place that they are declared, but not from other places. This allows a straightforward way to transmit several pieces of information back from an `at(somewhereElse)`, as shown by the following example which computes the roots of the quadratic equation $ax^2 + bx + c = 0$. For the sake of the example, we assume that `mathProcessor()` returns a Place suitable for calculation. In practice it would rarely if ever make sense to send such a small calculation to another place.

The starting place has local `vars` r and s, which will hold the roots. (They are initialized to $0 + 0i$ because of §**??**.) The computation at the math processor evaluates the roots of the equation. These roots are bound to math processor local `val` variables rr and ss. An inner place-shifting `at(h)` moves the computation back to the initial place, stored for that purpose in h, at which values can be assigned to the variables r and s at the starting place. Note that we take care to send only the minimal information across `at` boundaries; in particular we do not send the discriminant back to h. When the intermediate results are large and the desired data small, the savings can be substantial.

```
def printRootsOfQuadratic(a:Complex, b:Complex, c:Complex) {
  var r : Complex = Complex(0,0);
  var s : Complex = Complex(0,0);
  val h = here;
  at(mathProcessor()) {
    val disc = Math.sqrt(b*b - 4*a*c);
    val rr = (-b + disc) / (2*a);
    val ss = (-b - disc) / (2*a);
    at(h) {
```

```
        r = rr; s = ss;
      }
    }
    Console.OUT.println("r = " + r + "; s = " + s);
  }
```

14.3 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*	*(19.73)*
	|	clocked async *Statement*	
ClockedClause	::=	clocked (*ClockList*)	*(19.72)*

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.

STATIC SEMANTICS RULE: The statement in the body of an async is subject to the restriction that it must be acceptable as the body of a void method for an anonymous inner class declared at that point in the code. As such, it may reference variables in lexically enclosing scopes.

14.4 **Finish**

The statement finish S converts global termination to local termination.

| *FinishStatement* | ::= | `finish` *Statement* | *(19.79)* |
| | | | `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.)

14.5   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.)

14.6 Ateach statements

Deprecated: The `ateach` construct is deprecated.

AtEachStatement	::=	`ateach (` *LoopIndex* `in` *Exp* `)` *ClockedClause*$^?$	*(19.77)*
		Statement	
	|	`ateach (` *Exp* `)` *Statement*	
LoopIndexDeclarator	::=	*Id HasResultType*$^?$	*(19.129)*
	|	`[` *IdList* `]` *HasResultType*$^?$	
	|	*Id* `[` *IdList* `]` *HasResultType*$^?$	
LoopIndex	::=	*Mods*$^?$ *LoopIndexDeclarator*	*(19.130)*
	|	*Mods*$^?$ *VarKeyword LoopIndexDeclarator*	

In `ateach(p in D) S`, D must be either of type `Dist` (see §16.4) 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)) async S` to get the same effect. For each point p in D, at place `D(p)`, 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) {
    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)` .

14.7 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.

| *AtomicStatement* | ::= | atomic *Statement* | *(19.75)* |
| *WhenStatement* | ::= | when (*Exp*) *Statement* | *(19.76)* |

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) {
  this.list.add(x);
  atomic { this.size = this.size + 1; }
}
```

14.7.1 Unconditional atomic blocks

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

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:

$$\text{atomic \{s1; atomic s2\}} \;\; = \;\; \text{atomic \{s1; s2\}} \qquad (14.1)$$

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;
}
```

14.7.2   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 `await 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 14.7.1 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;
      }
    }
  }
```

□

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 `next;` to signal that it is finished with that work. `next;` 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 `next;` 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.

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

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

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.next()`. 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 `next;`. This, in effect, calls `c.next()` 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. `next;` 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.next();
  b_phase_two();
}
c.next();
a_phase_two();
```

In the proper execution, a and b both perform c.next() 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.next() and proceed to b_phase_two before a performed c.next().

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 statement

```
    next;
```

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

`next;` may be thought of as calling `c.next()` 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.next(); d.next()` while b executes `d.next(); c.next()`, then the two will deadlock. However, if the two clocks are waited on in parallel, as `next;` does, a and b will not deadlock.)

Equivalently, `next;` 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 `next` resumes execution once it is marked for progress by all the clocks it is registered with.

15.1.5   Dropping clocks

An activity may drop a clock by executing `c.drop()`.

The activity is no longer considered registered with this clock. A `ClockUseException` is thrown if the activity has already dropped `c`.

15.2   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 `next()` **method** may not be called on any clock. (The `next;` statement 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
          next;                   // (Bnext)
      }
      next;                       // (Anext)
}
```

(A), first of all, waits for the `finish` containing (B) to finish. (B) will execute its
`next` at (Bnext), and then wait for all other activities registered on `c` to execute
their `next`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:

$$c.resume(); \ next; \ = \ next; \tag{15.1}$$
$$c.resume(); \ d.resume(); \ = \ d.resume(); \ c.resume(); \tag{15.2}$$
$$c.resume(); \ c.resume(); \ = \ c.resume(); \tag{15.3}$$

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

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 qual-
ified 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 `next` statement to advance 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);
      next;
      phase("A", 2);
  }
  clocked async {
      phase("B", 1);
      next;
      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`, `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 `ValRail[Int]` 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 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; // 1-dim region with extent 1..100
val R2 = (1..100) as Region(1); // same as R1
val R3 = (0..99) * (-1..MAX_HEIGHT);
val R4 = Region.makeUpperTriangular(10);
val R5 = R4 && R3; // intersection of two regions
```

The expression `m..n`, for integer expressions `m` and `n`, evaluates to the rectangular, rank-1 region consisting of the points $\{[m], \ldots, [n]\}$. If `m` is greater than `n`, the region `m..n` is empty.

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` $\leq$ `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.

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 regions `T`$_i$. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them (that is, it is *convex*). (Banded and triangular regions are not rectangular.) The operation `R.boundingBox()` gives the smallest rectangular region containing `R`.

16.2.1   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.

16.3 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 `Point`s 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 `Array`s efficiently, and to make array element accesses quick—*e.g.*, avoiding constructing `Point`s 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`.

16.3.1   Array Constructors

To construct an array whose elements all have the same value, call `new Array[T](R, init)`, where `init:T` is the initial value. 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.

16.3.2   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 §8.6.5.)

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);
```

16.4 Distributions

Distributed arrays are spread across multiple Places. 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 v2.1; 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 = R -> 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.

16.4.1 Operations returning distributions

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

Unique distribution   The distribution Dist.makeUnique(Q) is the unique distribution from the region 1..k to Q mapping each point i to pi.

Constant distributions.   The distribution R->P maps every point in region R to place P, as does Dist.makeConstant(R,P).

Block distributions.   The distribution Dist.makeBlock(R) distributes the elements of R, in order, over all the places available to the program. Let p equal |R| div N and q equal |R| mod N, where N is the size of Q, and |R| is the size of R. The first q places get successive blocks of size $(p + 1)$ and the remaining places get blocks of size p.

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.

16.5   Distributed Arrays

Distributed arrays, instances of DistArray[T], are very much like Arrays, except that they distribute information among multiple Places 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);
```


16.6   Distributed Array Construction

DistArrays 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 [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](1..1000->here, ([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 `Places`, initialized with `i*j` at point `[i,j]`.

16.7   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.

16.7.1   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.

16.7.2   Constant promotion

For a region R and a value `v` of type T, the expression `new Array[T](R, v)` produces 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 constructed 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.

16.7.3   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 subscripted by a point in region `R && da.region`, and is undefined elsewhere. '
Recall that a rich set of operators are available on distributions (§16.4) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

16.7.4   Operations on Whole Arrays

Pointwise operations   The unary `map` operation applies a function to each element 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 performing `f` on all points in `a` in some unspecified order (perhaps in parallel). 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 example,

```
val a = [1,2,3,4];
val sum = a.reduce(Int.+, 0);
assert(sum == 10); // 10 == 1+2+3+4
```

`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*	*(19.144)*
	|	*Annotations Annotation*	
Annotation	::=	@ *NamedType*	*(19.145)*

Annotations can be applied to most syntactic constructs in the language including class declarations, constructors, methods, field declarations, local variable declarations and formal parameters, statements, expressions, and types. Multiple occurrences of the same annotation (i.e., multiple annotations with the same interface type) on the same entity are permitted.

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 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`	

		`|` `transient`	
MethMods	::=	*Mods*$^?$	*(3)*
		`|` *MethMods* `property`	
		`|` *MethMods Mod*	
TypeDefDecl	::=	*Mods*$^?$ `type` *Id TypeParams*$^?$ *FormalParams*$^?$	*(4)*

`|`   `clocked` appears as the second line.

MethMods   ::=   *Mods*$^?$ *(3)*
             `|`   *MethMods* `property`
             `|`   *MethMods Mod*

TypeDefDecl   ::=   *Mods*$^?$ `type` *Id TypeParams*$^?$ *FormalParams*$^?$ *(4)*
             WhereClause$^?$ `=` *Type* `;`

Properties   ::=   `(` *PropertyList* `)`   *(5)*

PropertyList   ::=   *Property*   *(6)*
             `|`   *PropertyList* `,` *Property*

Property   ::=   *Annotations*$^?$ *Id ResultType*   *(7)*

MethodDecl   ::=   *MethMods* `def` *Id TypeParams*$^?$ *FormalParams*   *(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*

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$^?$ Properties$^?$ 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 . ˆ
			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* ˆ *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.