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.

Example: *In the following variation of the preceding example,* `a`*'s original value* a_1 *is a rail with two references to the same* `Cell[Long]` 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[Long]` 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[Long](5);
val a : Rail[Cell[Long]] = [c,c as Cell[Long]];
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)
```

13.3.5 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.

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* `Long`*, 0, is used instead. Note that we could not make a transient field* `c : Long{c != 0}`*, since the type has no default value, and copying would in fact set it to zero.*

```
class Trans {
   val a : Long = 1;
   transient val b : Long = 2;
   //ERROR: transient val c : Long{c != 0} = 3;
   def example() {
     assert(a == 1 && b == 2);
     at(here) {
        assert(a == 1 && b == 0);
     }
   }
}
```

13.3.6 Copying and GlobalRef

A `GlobalRef[T]` (say `g`) contains a reference to a value `v` of type `T`, in a form which can be transmitted, and a `Place g.home` indicating where the value lives. When a `GlobalRef` is serialized an opaque, globally unique handle to `v` is created.

Example: *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 application operation `g()`. `g()` is guarded; it can only be called when `g.home == here`. If you want to do anything other than pass a global reference around or compare two of them for equality, you need to placeshift back to the home place of the reference, often with `at(g.home)`.

Example: *The following program, for reasons best known to the programmer, modifies the command-line argument array.*

```
public static def main(argv:Rail[String]) {
  val argref = GlobalRef[Rail[String]](argv);
  val world = Place.places();
  at(world.next(here))
      use(argref);
}
static def use(argref : GlobalRef[Rail[String]]) {
  at(argref) {
    val argv = argref();
    argv(0) = "Hi!";
  }
}
```

There is an implicit coercion from `GlobalRef[T]` to `Place`, so `at(argref)S` goes to `argref.home`.

13.3.7 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. A disciplined use of copy specifiers to make explicit just what gets copied can ameliorate this issue.

- As seen in the examples above, a local variable reference `x` may refer to different objects in different nested `at` scopes. The programmer must either ensure that a variable accessed across an `at` boundary has no mutable state or be prepared to reason about which copy gets modified. A disciplined use of copy specifiers to give different names to variables can ameliorate this concern.

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 progress by executing control structures. For example, `when(x==0);` blocks the current activity until some other activity sets `x` to zero. However, activities determine the loca at which they may be blocked and resumed, using `when` and similar constructs. There are no means by which one activity can arbitrarily interrupt, block, kill or resume another.

An activity may be *running*, *blocked* on some condition or *terminated*. An activity terminates when it has no more statements to execute; it terminates *normally* (*abruptly*) if the last statement it executes terminates normally (abruptly) (§14.1).

Activities can be long-running, and may access a lot of data. In particular they can call recursive methods (and therefore have runtime stacks). However, activities can also perform very few actions, such as incrementing some variables.

An activity may asynchronously and in parallel launch activities. 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 have, recursively, terminated globally. For example, consider:

```
async {s1();}
async {s2();}
```

The primary activity spawns two child activities and then terminates locally, very quickly. The child activities may take arbitrary amounts of time to terminate (and may spawn grandchildren). When `s1()`, `s2()`, and all their descendants terminate locally, then the primary activity terminates globally.

The program as a whole terminates when the root activity terminates globally. In particular, X10 does not permit the creation of daemon threads—threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§14.4).

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

14.1 The X10 rooted exception model

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

The presence of a root activity and the concept of global termination permits X10 to adopt a more powerful exception model. In any state of the computation, say that an activity A is *a root of* an activity B if A is an ancestor of B and A is blocked at a statement (such as the `finish` statement §14.3) awaiting the termination of B (and possibly other activities). For every X10 computation, the *root-of* relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the *root-of* tree when the activity terminates (based on the semantics of the statement being executed by the activity).[2] There is always a good place to put a `try-catch` block to catch exceptions thrown by an asynchronous activity.

14.2 `async`: Spawning an activity

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

An activity is created by executing the `async` statement:

AsyncStmt	::=	`async` *ClockedClause*$^?$ *Stmt*	*(20.15)*
	|	`clocked` `async` *Stmt*	
ClockedClause	::=	`clocked` *Arguments*	*(20.39)*

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

[2]In X10 v2.4 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.

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 in a place are not ordered in any way. They are added to the set of activities running in the place and will be executed based on the local scheduler's decisions. If some particular sequencing of events is needed, `when`, `atomic`, `finish`, clocks, and other X10 constructs can be used. X10 implementations are not required to have fair schedulers, though every implementation should make a best faith effort to ensure that every activity eventually gets a chance to make forward progress.

The statement in the body of an `async` is subject to the restriction that it must be acceptable as the body of a `void` method for an anonymous inner class declared at that point in the code. For example, it may reference `val` variables in lexically enclosing scopes, but not `var` variables. Similarly, it cannot `break` or `continue` surrounding loops.

14.3 Finish

The statement `finish S` converts global termination to local termination.

```
FinishStmt ::= finish Stmt                    (20.72)
             | clocked finish Stmt
```

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, of type `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` terminates normally. If `S` throws exceptions, `finish S` collects the exceptions and wraps them in a `MultipleExceptions`, whereas `finish finish S` does the same, and then puts that `MultipleExceptions` inside of a second `MultipleExceptions`.

14.4 Initial activity

An X10 computation is initiated from the command line on the presentation of a class or struct name C. The container must have a main method:

```
public static def main(a: Rail[String]):void
```

method, otherwise an exception is thrown and the computation terminates. The single statement

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

is executed where s is a one-dimensional Rail of strings created from the command line arguments. This single activity is the root activity for the entire computation. (See §13 for a discussion of places.)

14.5 Ateach statements

Deprecated: The ateach construct is deprecated.

AtEachStmt	::=	`ateach` `(` *LoopIndex* `in` *Exp* `)` *ClockedClause*$^?$ *Stmt*	*(20.18)*
	|	`ateach` `(` *Exp* `)` *Stmt*	
LoopIndexDeclr	::=	*Id* *HasResultType*$^?$	*(20.113)*
	|	`[` *IdList* `]` *HasResultType*$^?$	
	|	*Id* `[` *IdList* `]` *HasResultType*$^?$	
LoopIndex	::=	*Mods*$^?$ *LoopIndexDeclr*	*(20.112)*
	|	*Mods*$^?$ *VarKeyword* *LoopIndexDeclr*	

In ateach(p in D) S, D must be either of type Dist (see §16.4.3) or of type DistArray[T] (see §16), and p will be of type Point (see §16.3.1). 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 could write

```
for (place in D.places()) at (place) async {
    for (p in D|here) async {
        S(p);
    }
}
```

For each point p in D, statement S is executed concurrently at place D(p).

break and continue statements may not be applied to ateach.

14.6 vars and Activities

X10 restricts the use of local var variables in activities, to make programs more deterministic. Specifically, a local var variable x defined outside of async S cannot appear inside async S unless there is a finish surrounding async S with the definition of x outside of it.

Example: *The following code is fine; the definition of* result *appears outside of the* finish *block:*

```
var result : Long = 0;
finish {
  async result = 1;
}
assert result == 1;
```

This code is deterministic: the async *will finish before the* assert *starts, and the* assert*'s test will be true.*

However, without the finish*, the code would not compile in X10. If it were allowed to compile, the activity might finish or might not finish before the* println*, and the program would not be deterministic.*

14.7 Atomic blocks

X10's atomic blocks (atomic S and when (c) S) provide a high-level construct for coordinating the mutation of shared data. An atomic block is executing as if in a single step, with respect to atomic blocks executed by all other activities in the same place. That is, all atomic blocks execute in a *serializable* order. Hence no atomic block can see the intermediate state within the execution of some other atomic block.

Code executed inside of atomic S and when(E)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, Clock.advanceAll(), finish. (The use of a nested atomic is permitted.)
- S may not force() a Future.
- S may not use at expressions.

That is S must be sequential, single-place and non-blocking.

A programmer may use atomic blocks to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple activities running in the same place. An X10 program in which all accesses (both reads and writes) of shared variables appear in `atomic` or `when` blocks is guaranteed to use all shared variables atomically. Equivalently, if two accesses to some shared variable `v` could collide at runtime, and one is in an atomic block, then the other must be in an atomic block as well to guarantee atomicity of the accesses to `v`. If some accesses to shared variables are not protected by `atomic` or `when`, then race conditions or deadlocks may occur. In particular, atomic blocks at the same place are atomic with respect to each other. They may not be atomic with respect to non-atomic code, or with respect to atomic sections at different places.

AtomicStmt	::=	`atomic` *Stmt*	*(20.21)*
WhenStmt	::=	`when` `(` *Exp* `)` *Stmt*	*(20.207)*

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 is required to be true, but it is not.*

In this case, the invariant can be maintained by making the increment atomic. Doing so forbids that sequence of events; the `atomic` *block cannot be stopped partway.*

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

14.7.1 Unconditional atomic blocks

The simplest form of an atomic block is the *unconditional atomic block*: `atomic S`. it executes `S` as if in a single step with respect to all other concurrently executing atomic blocks. `S` may throw an exception; when control leaves `atomic S` the side-effects executed so far are made visible to other atomic blocks. The programmer may surround `S` with a `try/finally` block and try to undo assignments when an exception is thrown.

Atomic blocks are closely related to non-blocking synchronization constructs [7], and can be used to implement non-blocking concurrent algorithms.

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

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

Unconditional 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:Any = null;
public atomic def CAS(old1: Any, y: Any):Boolean {
   if (target.equals(old1)) {
     target = y;
     return true;
   }
   return false;
}
```

14.7.2 Conditional atomic blocks

Conditional atomic blocks are of the form `when(b)S`; `b` is called the *guard*, and `S` the *body*.

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

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

The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends.

The body `S` of `when(b)S` is subject to the same restrictions that the body of `atomic S` is. The guard is subject to the same restrictions as well. Furthermore, guards should not have side effects. Note that this implies that guarded statements are required to be *flat*, that is, they may not contain conditional atomic blocks. (The implementation of nested conditional atomic blocks may require sophisticated operational techniques such as rollbacks.)

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 falsify the condition after the test was seen to be true, but 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.

Example: *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;
      }
    }
  }
```

When `when` is Tested

Suppose that activity A is blocked waiting on `when(e)S`, because `e` is `false`. If some other activity B changes the state in an `atomic` section in a way that makes `e` become `true`, then either:

- A will eventually execute `S`, or
- Some activity $C \neq A$ will cause `e` to become `false` again, or
- Some activity at that place will execute an infinite number of steps.

In particular, if no other activity ever falsifies `e`, then A will, eventually, discover that `e` evaluates to `true` and run `S` (provided that no other activity at that place is running forever).

Two caveats are worth noting:

- X10 has no guarantees of fairness or liveness.
- X10 only makes guarantees about state changes *in atomic blocks* alerting `when`s. State changes outside of atomic blocks might not cause A to re-evaluate `e`.

Example: *The method* `good` *below will always terminate (assuming no other activities are executing an infinite number of steps). In particular, if the* `when` *statement is allowed to run first and block on* `c()`*, the* `atomic` *will alert it that* `c()` *has changed.*

The method `bad` *has no such guarantee: it might terminate if the compiler and scheduler are in a generous mood, or the* `when` *might wait forever to be told that* `c()` *is now true. Without an* `atomic`*, the* `when` *statement might not be notified about the change in* `c()`.

```
static def good() {
  val c = new Cell[Boolean](false);
  async {
    atomic {c() = true;}
  }
  when( c() );
}
static def bad() {
  val c = new Cell[Boolean](false);
  async {
    c() = true;
  }
  when( c() );
}
```

14.8 Use of Atomic Blocks

The semantics of atomicity is chosen as a compromise between programming simplicity and efficient implementation. Unlike some possible definitions of "atomic", atomic blocks do not provide absolute atomicity.

Atomic blocks are atomic with respect to *each other*.

```
var n : Long = 0;
finish {
  async atomic n = n + 1; //(a)
  async atomic n = n + 2; //(b)
}
```

This program has only two possible interleavings: either `(a)` entirely precedes `(b)` or `(b)` entirely precedes `(a)`. Both end up with `n==3`.

However, atomic blocks are not atomic with respect to non-atomic code. It we remove the `atomic`s on `(a)`, we get far messier semantics.

```
var n : Long = 0;
finish {
  // LEGAL BUT UNWISE
  async n = n + 1;          //(a)
  async atomic n = n + 2;   //(b)
}
```

If X10 had absolute ("strong") atomic semantics, this program would be guaranteed to treat the atomic increment as a single statement. This would permit three interleavings: the two possible from the fully atomic program, or a third one with the events: `(a)`'s read of `0` from `n`, the entirety of `(b)`, and then `(a)`'s write of `0+1` back to `n`. This interleaving results in `n==1`. So, with absolute atomic semantics, `n==1` or `n==3` are the possible results.

However, atomic blocks in X10 are “weak”. Atomic blocks are atomic with respect to each other — but there is no guarantee about how they interact with non-atomic statements at all. They might even break up the atomicity of an `atomic` block. In particular, the following fourth interleaving is possible: `(a)`’s read of `0` from `n`, `(b)`’s read of `0` from `n`, `(a)`’s write of `1` to `n`, and `(b)`’s write of `2` to `n`. Thus, `n==2` is permissible as a result in X10.

However, X10’s semantics do impose a certain burden on the programmer. A sufficient rule of thumb is that, if *any* access to a variable is done in an atomic section, then *all* accesses to it must be in atomic sections.

Atomic sections are a powerful and convenient general solution. Classes in the package `x10.util.concurrent` may be more efficient and more convenient in particular cases. For example, an `AtomicInteger` provides an atomic integer cell, with atomic get, set, compare-and-set, and add operations. Each `AtomicInteger` takes care of its own locking. Accesses to one `AtomicInteger` a only block activities which try to access a — not others, not even if they are using different `AtomicInteger`s or even `atomic` blocks.

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 to be dynamic (new activities can be registered with clocks, and terminated activities automatically deregister from clocks), and to support a simple syntactic discipline for deadlock-free and race-free conditions. Just like `finish`, X10's clocks can both be used within a single place and to synchronize activities across multiple places.

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

The program creates a clock `cl` to manage the phases. Each participating activity does the work of its first phase, and then executes `Clock.advanceAll();` to signal that it is finished with that work. `Clock.advanceAll();` is blocking, and causes the participant to wait until all participant have finished with the phase – as measured by the clock `cl` to which they are both registered. Then they do the second phase, and another `Clock.advanceAll();` to make sure that neither proceeds to the third phase until both are ready. This example uses `finish` to wait for both particiants to finish.

```
class ClockEx {
  static def say(s:String) {
    Console.OUT.println(s);
  }
  public static def main(argv:Rail[String]) {
    finish async{
      val cl = Clock.make();
      async clocked(cl) {// Activity A
        say("A-1");
        Clock.advanceAll();
        say("A-2");
        Clock.advanceAll();
        say("A-3");
      }// Activity A

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

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

The key invariants associated with clocks are as follows. At any stage of the computation, a clock has zero or more *registered* activities. An activity may perform operations only on those clocks it is registered with (these clocks constitute its *clock set*). An attempt by an activity to operate on a clock it is not registered with will cause a `ClockUseException` to be thrown. An activity is registered with zero or more (existing) 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 operations on `c`.
- It may *resume* the clock, with `c.resume()`, indicating that it has finished with the current phase associated with `c` and is ready to move on to the next one.
- It may *wait* on the clock, with `c.advance()`. This first does `c.resume()`, and then blocks the current activity until the start of the next phase, *viz.*, until all other activities registered on that clock have called `c.resume()`.
- It may *block* on all the clocks it is registered with simultaneously, by the command `Clock.advanceAll();`. This, in effect, calls `c.advance()` simultaneously on all clocks `c` that the current activity is registered with.
- Other miscellaneous operations are available as well; see the `Clock` API.

15.1 Clock operations

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

15.1.1 Creating new clocks

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

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

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

15.1.2 Registering new activities on clocks

AsyncStmt	::=	`async` *ClockedClause*$^?$ *Stmt*	*(20.15)*
	|	`clocked async` *Stmt*	
ClockedClause	::=	`clocked` *Arguments*	*(20.39)*

The `async` statement with a `clocked` clause of either form, say

```
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()`. That is, a and b are in the same phase of the clock.

```
// ACTIVITY a
val c = Clock.make();
c.resume();
async clocked(c) {
  // ACTIVITY b
  c.advance();
  b_phase_two();
  // END OF ACTIVITY b
```

```
    }
    c.advance();
    a_phase_two();
    // END OF ACTIVITY a
```

In the proper execution, a and b both perform `c.advance()` and then their phase-2 actions. However, if b were not initially in the resume state for `c`, there would be a race condition; b could perform `c.advance()` and proceed to `b_phase_two` before a performed `c.advance()`.

An activity may check whether or not it is registered on a clock `c` by the method call `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 (or another process) 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 itself 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.

An activity may invoke `Clock.resumeAll()` to resume all the clocks that it is registered with and has not yet dropped. This `resume()`s all these clocks in parallel, or, equivalently, sequentially in some arbitrary order.

15.1.4 Advancing clocks

An activity may execute the following method call to signal that it is done with the current phase.

```
    Clock.advanceAll();
```

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

`Clock.advanceAll();` may be thought of as calling `c.advance()` in parallel for all clocks that the current activity is registered with. (The parallelism is conceptually important: if activities a and b are both registered on clocks `c` and `d`, and a executes `c.advance(); d.advance()` while b executes `d.advance(); c.advance()`, then the two will deadlock. However, if the two clocks are waited on in parallel, as `Clock.advanceAll();` does, a and b will not deadlock.)

Equivalently, `Clock.advanceAll();` sequentially calls `c.resume()` for each registered clock `c`, in arbitrary order, and then `c.advance()` for each clock, again in arbitrary order.

An activity blocked on `advance()` resumes execution once it is marked for progress by all the clocks it is registered with.

15.1.5 Dropping clocks

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

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

15.2 Deadlock Freedom

In general, programs using clocks can deadlock, just as programs using loops can fail to terminate. However, programs written with a particular syntactic discipline *are* guaranteed to be deadlock-free, just as programs which use only bounded loops are guaranteed to terminate. The syntactic discipline is:

- The `advance()` instance method shall not be called on any clock. (The `Clock.advanceAll();` method is allowed for this discipline.)
- Inside of `finish{S}`, all clocked `async`s shall be in the scope some unclocked `async` in `S`.

X10 does not enforce this discipline. Doing so would exclude useful programs, many of which are deadlock-free for reasons more subtle than the straightforward syntactic discipline. Still, this discipline is useful for simple cases.

The first clause of the discipline prevents a deadlock in which an activity is registered on two clocks, advances one of them, and ignores the other. The second clause prevents the following deadlock.

```
val c:Clock = Clock.make();
async clocked(c) {                // (A)
      finish async clocked(c) {   // (B) Violates clause 2
            Clock.advanceAll();   // (Bnext)
      }
      Clock.advanceAll();         // (Anext)
}
```

`(A)`, first of all, waits for the `finish` containing `(B)` to finish. `(B)` will execute its `advance` at `(Bnext)`, and then wait for all other activities registered on `c` to execute their `advance()`s. However, `(A)` is registered on `c`. So, `(B)` cannot finish until `(A)` has proceeded to `(Anext)`, and `(A)` cannot proceed until `(B)` finishes. Thus, the activities are deadlocked.

15.3 Program equivalences

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

$$\texttt{c.resume(); Clock.advanceAll();} = \texttt{Clock.advanceAll();} \quad (15.1)$$
$$\texttt{c.resume(); d.resume();} = \texttt{d.resume(); c.resume();} \quad (15.2)$$
$$\texttt{c.resume(); c.resume();} = \texttt{c.resume();} \quad (15.3)$$

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

15.4 Clocked Finish

In the most common case of a single clock coordinating a few behaviors, X10 allows coding with an implicit clock. `finish` and `async` statements may be qualified with `clocked`.

A `clocked finish` introduces a new clock. It executes its body in the usual way that a `finish` does— except that, when its body completes, the activity executing the `clocked finish` drops the clock, while it waits for asynchronous spawned `async`s to terminate.

A `clocked async` registers its async with the implicit clock of the surrounding `clocked finish`.

The bodies of the `clocked finish` and `clocked async` statements may use the `Clock.advanceAll()` method call to advance the implicit clock. Since the implicit clock is not available in a variable, it cannot be manipulated directly. (If you want to manipulate the clock directly, use an explicit clock.)

Example: *The following code starts two activities, each of which perform their first phase, wait for the other to finish phase 1, and then perform their second phase.*

```
clocked finish {
  clocked async {
     phase("A", 1);
     Clock.advanceAll();
```

```
      phase("A", 2);
   }
   clocked async {
      phase("B", 1);
      Clock.advanceAll();
      phase("B", 2);
   }
}
```

The `clocked async` and `clocked finish` constructs can be combined with `at` in the same ways as their unclocked counterparts.

Example: *The following code creates one clocked activity in every Place that synchronize to execute* `iter` *steps of a two phase computation. The clock ensures that every activity has completed the call to* `before(N)` *before any activity calls* `after(N)`. *Note that executions of* `after(N)` *and* `before(N+1)` *in different places may overlap; if this is not desired an additional call to* `Clock.advanceAll()` *could be added after the call to* `after(count)`.

```
clocked finish {
  for (p in Place.places()) {
     at (p) clocked async {
       for (count in 1..iters) {
         before(count);
         Clock.advanceAll();
         after(count);
       }
     }
  }
}
```

Clocked finishes may be nested. The inner `clocked finish` operates in a single phase of the outer one.

16 Rails and Arrays

16.1 Overview

Indexable memory is a fundamental abstraction provided by a programming language. To enable the programmer to best balance performance and flexibility, X10 provides a layered implementation of indexable memory. The foundation of all indexable storage in X10 is `x10.lang.Rail`, an intrinsic one-dimensional array analogous to the built-in arrays provided by languages such as C or Java. On top of `Rail`, more sophisticated array abstractions can be constructed completely as user-defined X10 classes. Two such families of user-defined array abstractions are included in the X10 core class libraries in the `x10.array` and `x10.regionarray` packages. Both families of arrays provide both local (single place) and distributed (multi-place) arrays.

The next section specifies `Rail`. Subsequent sections outline the `x10.array` and `x10.regionarray` packages. As discussed, in more detail below, `Rail` and the classes of `x10.array` provide significantly higher performance operations than the corresponding classes of `x10.regionarray`. Therefore we recommend that programmers only use the more general arrays of `x10.regionarray` where the increased flexibility justifies the redudced performance. We also encourage programmers to use the classes of `x10.array` as an example of how to define high-performance array abstractions in X10 and use them as templates for defining additional high-performance array abstractions (for example column-major arrays as in Fortran or 1-based arrays as in MATLAB).

16.2 Rails

The `Rail` class provides a basic abstraction of fixed-sized indexed storage. If `r` is a `Rail[T]`, then `r` contains `r.size` elements of type `T` that may be accessed using the `Long` values `0` to `r.size-1` as indices. All accesses to the elements of a `Rail` are checked: attempting to use an index that is less than `0` or greater than `r.size-1` will result in an `ArrayIndexOutOfBoundsException` being thrown.

As shown in the example below, instances of `Rail[T]` may be created using one of several constructors that initialize the data elements to the zero-value of `T`, a single initial value, or to a different initial value for each element of the `Rail`.

```
// A zero-initialized Rail of 10 doubles
val r1 = new Rail[Double](10);

// A Rail of 10 doubles, all initialized to pi
val r2 = new Rail[Double](10, Math.PI);

// A Rail of 10 doubles, r3(i) == i*pi
val r3 = new Rail[Double](10, (i:long)=>i*Math.PI);
```

As described in more detail in section 11.26, X10 also includes a shorthand form for `Rail` construction: simply put brackets around a list of expressions.

```
// A Rail[Long] containing the first 5 primes
val r1 = [2,3,5,7,11];

// A Rail[Double] such that r2(i) == i*pi
val r2 = [Math.PI, 2*Math.PI, 3*Math.PI, 4*Math.PI];
```

Accessing and updating single elements of a `Rail` is doing using application syntax. For example, `a(i)=a(i+1);` sets the `i`th element of `a` to the value of the `i+1` element of `a`. If `T` supports the `+` or `-` operation, then the usual pre/post increment/decrement operations are also available on individual array elements. For example, `a(i)++` is equivalent to `a(i) = a(i)+1`

Iteration over the elements of a `Rail` can be accomplished using several equivalent idioms. Furthermore, via some modest compiler support, each of these for loops will actually result in identical generated code. In the examples below, `r` is a `Rail[Long]` and `sum` is a local variable of type `Long`.

```
// A classic C-style for loop
for (var i:long=0; i<r.size; i++) {
    sum += r(i);
}

// Iterate over the LongRange 0..(r.size-1)
for (i in 0..(r.size-1)) {
    sum += r(i);
}

// Get the LongRange to iterate over from r
for (i in r.range()) {
    sum += r(i);
}

// Directly iterate over the values of r
for (v in r) {
    sum += v;
}
```

Basic bulk operations such as clearing (setting to zero), filling with a single value, and copying are provided by methods of the `Rail` class. Efficient copying of Rails across places is supported via the combination of the `x10.lang.GlobalRail` struct[1] and the `asyncCopy` method of `Rail`.

Additional complex bulk operations on `Rail` such as sorting, searching, map, and reduce are provided by the `x10.util.RailUtils` class.

When implementing higher-level data structures that use `Rail` as their backing storage, there may be significant performance advantage in performing unsafe operations on `Rail`. For example, when building a multi-dimensional `Array` class, the `Array` level bounds checking and initialization logic make the `Rail` level operations redundant. To support such scenarios the class `x10.lang.Unsafe` provides methods to allocate uninitialized `Rail` objects and to access the elements of a `Rail` without bounds checking. These unsafe extensions should be used judiciously, as improper use can result in memory safety violations that would not be possible in pure X10 code.

16.3 x10.array: Simple Arrays

Classes in the `x10.array` package provide high-performance implementations of both local and distributed multi-dimension arrays. The array implementations in this package are restricted to the case of rectangular, dense, zero-based index spaces. By making this restriction, the indexing calculations for both single-place and multi-place arrays can be optimized, resulting in an array implementation that should obtain equivalent performance to the corresponding array abstractions in C or Fortran. More general index spaces are supported by the classes in the `x10.regionarray` package (see Section 16.4.

The three main concepts of this package: iteration spaces, arrays, and distributed arrays are outlined below. All three concepts are implemented as simple class hierarchies with an abstract superclass and a collection of concrete final subclasses that contain the performance-critical operations. Client code using the abstractions can be written (with lower performance) using the abstract APIs provided by the superclass, but performance sensitive code should be written using the more specific type of the concrete subclasses. This enables compile time optimization and inlining of key operations, resulting in optimal code sequences.

16.3.1 Points

Both kinds of 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 `Rail[Long] by the \xcdPoint`.make‘ factory methods:

[1]a specialized version of `x10.lang.GlobalRef` that includes the `size` of the referenced `Rail` to enable bounds checking

```
val origin_1 : Point{rank==1} = Point.make(0);
val origin_2 : Point{rank==2} = Point.make(0,0);
val origin_5 : Point = Point.make(new Rail[Long](5));
```

There is an implicit conversion from `Rail[Long]` 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 function application, or, if you prefer, by subscripting; `p(i)` is the `i`th coordinate of the point `p`. `Point(`n`)` is a `type`-defined shorthand for `Point{rank==`n`}`.

16.3.2 IterationSpace

An `IterationSpace` is a generalization of `LongRange` to multiple dimensions. The `rank` property of the `IterationSpace` corresponds to its dimensionality. An `IterationSpace` represents an ordered collection of `Points` of the same `rank` as the `IterationSpace`. The primary purpose of an `IterationSpace` is to represent the indices of an `Array` or `DistArray`.

16.3.3 Array

The abstract `Array` class provides rank-generic operations for single place multi-dimensional arrays. Its concrete subclasses `Array_1`, `Array_2`, etc. provide rank-specific operations such as efficient element access. The APIs of the classes are designed to be a natural extension of the `Rail` API to multiple dimensions. In most usage scenarios, programmers should operate using the types of the concrete subclasses, not of `Array` itself.

The example below illustrates the construction and indexing operations of rank 2 arrays using a simple matrix multiply kernel where `N` is a `Long`.

```
val a = new Array_2[double](N, N, (i:long,j:long)=>(i*j) as double);
val b = new Array_2[double](N, N, (i:long,j:long)=>(i-j) as double);
val c = new Array_2[double](N, N, (i:long,j:long)=>(i+j) as double);

for (i in 0..(N-1))
  for (j in 0..(N-1))
    for (k in 0..(N-1))
      a(i,j) += b(i,k)*c(k,j);
```

Similarly to `Rail`, iteration over the elements of a `Array` can be accomplished using several equivalent idioms. Furthermore, via some modest compiler support, each of these for loops will actually result in identical generated code. In the examples below, `a` is a `Array_2[Long]` and `sum` is a local variable of type `Long`.

```
// A classic C-style for loop
for (var i:long=0; i<a.numElems_1; i++) {
  for (var j:long=0; j<a.numElems_2; j++) {
    sum += a(i,j);
  }
}

// Iterate over the indices of a using Point destructuring
for ([i,j] in a.indices()) {
    sum += a(i,j);
}

// Directly iterate over the values of a
for (v in a) {
    sum += v;
}
```

An additional idiom which iterates over the `Points` in the `IterationSpace` of a without destructuring the `Points` is also supported, but should be avoid in the current implementation of X10 as the compiler does not optimize away all of the overheads associated with explicit use of `Point` objects.

```
// Iterate over the indices of a using Points
for (p in a.indices()) {
    sum += a(p);
}
```

16.3.4 DistArray

The abstract class `DistArray` and its concrete subclasses represent an extension of the `Array` API to multiple places. In X10 version 2.4.0 `DistArrays` are still a work in progress. Only three concrete implementations are available: `DistArray_Unique` which has one data element per `Place`, `DistArray_Block_1` which block distributes a rank-1 array across the `Places` in its `PlaceGroup`, and `DistArray_BlockBlock_2` which distributes blocks of a rank-2 array across the `Places` in its `PlaceGroup`.

The API of `DistArray` is preliminary in this release of X10 we anticipate extending it with more operations and supporting a wider range of ranks and distributions in future release of X10.

16.4 x10.regionarray: Flexible Arrays

Classes in the `x10.regionarray` package provide the most general and flexible array abstraction that support mapping arbitrary multi-dimensional index spaces to data elements. Although they are significantly more flexible than `Rails` or the classes of the `x10.array` package, this flexibility does carry with it an expectation of lower runtime performance.

`Array`s 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.

16.4.1 Regions

A *region* is a set of points of the same rank. X10 provides a built-in class, `x10.regionarray.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.

Example:

```
val MAX_HEIGHT=20;
val Null = Region.makeUnit(); //Empty 0-dimensional region
val R1 = Region.make(1, 100); // Region 1..100
val R2 = Region.make(1..100);  // Region 1..100
val R3 = Region.make(0..99, -1..MAX_HEIGHT);
val R4 = Region.makeUpperTriangular(10);
val R5 = R4 && R3; // intersection of two regions
```

The `LongRange` *value* `1..100` *can be used to construct the one-dimensional* `Region` *consisting of the points* {`[m]`, …, `[n]`} `Region` *by using the* `Region.make` *factory method.* `LongRange`*s are useful in building up regions, especially rectangular regions.*

By a special dispensation, the compiler knows that, if `r : Region(m)` and `s : Region(n)`, then `r*s : Region(m+n)`. (The X10 type system ordinarily could not specify the sum; the best it could do would be `r*s : Region`, with the rank of the region unknown.) This feature allows more convenient use of arrays; in particular, one does not need to keep track of ranks nearly so much.

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

- `Region.makeEmpty(n)` returns an empty region of rank `n`.
- `Region.makeFull(n)` returns the region containing all points of rank `n`.
- `Region.makeUnit()` returns the region of rank 0 containing the unique point of rank 0. It is useful as the identity for Cartesian product of regions.
- `Region.makeHalfspace(normal, k)`, where `normal` is a `Point` and `k` an `Long`, returns the unbounded half-space of rank `normal.rank`, consisting of all points `p` satisfying the vector inequality `p`·`normal` $\leq$ `k`.
- `Region.makeRectangular(min, max)`, where `min` and `max` are rank-1 length-`n` integer arrays, returns a `Region(n)` equal to: `[min(0) .. max(0)`, …, `min(n-1)..max(n-1)]`.

- `Region.makeBanded(size, a, b)` constructs the banded `Region(2)` of size `size`, with `a` bands above and `b` bands below the diagonal.
- `Region.makeBanded(size)` constructs the banded `Region(2)` with just the main diagonal.
- `Region.makeUpperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `Region.makeLowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular `N x N` matrix.
- If `R` is a region, and `p` a Point of the same rank, then `R+p` is `R` translated forwards by `p` – the region whose points are `r+p` for each `r` in `R`.
- If `R` is a region, and `p` a Point of the same rank, then `R-p` is `R` translated backwards by `p` – the region whose points are `r-p` for each `r` in `R`.

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of the region `(1..2)*(1..2)` are ordered as

```
(1,1), (1,2), (2,1), (2,2)
```

Sequential iteration statements such as `for` (§12.12) iterate over the points in a region in the canonical order.

A region is said to be *rectangular* if it is of the form $(\mathtt{T}_1 * \cdots * \mathtt{T}_k)$ for some set of intervals $\mathtt{T}_i = \mathtt{l}_i \ \mathtt{..}\ \mathtt{h}_i$. In particular an `LongRange` turned into a `Region` is rectangular: `Region.make(1..10)`. 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`.

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, `Region.make(1..10) && Region.make(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, `Region.make(1..2)*Region.make(3..4)*Region.make(5..6)` is the region of rank `3` containing the eight points with coordinates `[1,3,5]`, `[1,3,6]`, `[1,4,5]`, `[1,4,6]`, `[2,3,5]`, `[2,3,6]`, `[2,4,5]`, `[2,4,6]`.

For a region `R` and point `p` of the same rank, `R+p` and `R-p` represent the translation of the region forward and backward by `p`. That is, `R+p` is the set of points `p+q` for all `q` in `R`, and `R-p` is the set of `q-p`.

More `Region` methods are described in the API documentation.

16.4.2 Arrays

Arrays are organized data, arranged so that the data can be accessed with subscripts. An `Array[T] A` has a `Region A.region`, specifying 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.

Example: *Indeed, it is possible to write code that works on* `Array`*s regardless of dimension. For example, to add one* `Array[Long] src` *into another* `dest`*,*

```
static def addInto(src: Array[Long], dest:Array[Long])
  {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`. This is the same operation as function application (§10.2); arrays implement function types, and can be used as functions.

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]` named 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`. `a.rank` is a cached copy of `a.region.rank`.

Array Constructors

To construct an array whose elements all have the same value `init`, call `new Array[T](R, init)`. For example, an array of a thousand `"oh!"`s can be made by: `new Array[String](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`.

Example: *One way to construct the array* `[11, 22, 33]` *is with an array constructor* `new Array[Long](3, (i:long)=>11*i)`. *To construct a multiplication table, call* `new Array[Long](Region.make(0..9, 0..9), (p:Point(2)) => p(0)*p(1))`.

Other constructors are available; see the API documentation and §11.26.

Array Operations

The basic operation on `Array`s is subscripting. If `a:Array[T]` and `p:Point{rank == a.rank}`, then `a(p)` is the value of type `T` appearing at position `p` in `a`. The syntax is identical to function application, and, indeed, arrays may be used as functions. `a(p)` may be assigned to, as well, by the usual assignment syntax `a(p)=t`. (This uses the application and setting syntactic sugar, as given in §8.7.5.)

Sometimes it is more convenient to subscript by longs. Arrays of rank 1-4 can, in fact, be accessed by longs:

```
val A1 = new Array[Long](10, 0);
A1(4) = A1(4) + 1;
val A4 = new Array[Long](Region.make(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 take the logarithm of every element of an `Array[Double]` by:

```
for (p in a) a(p) = Math.log(a(p));
```

The method `a.values()` can be used to enumerate all the values of an `Array[T]` array `a`.

```
static def allNonNegatives(a:Array[Double]):Boolean {
 for (v in a.values()) if (v < 0.0D) return false;
 return true;
}
```

16.4.3 Distributions

Distributed arrays are spread across multiple `Place`s. A *distribution*, a mapping from a region to a set of places, describes where each element of a distributed array is kept. Distributions are embodied by the class `x10.regionarray.Dist` and its subclasses.

The *rank* of a distribution is the rank of the underlying region, and thus the rank of every point that the distribution applies to.

Example:

```
val R  <: Region = Region.make(1..100);
val D1 <: Dist = Dist.makeBlock(R);
val D2 <: Dist = Dist.makeConstant(R, here);
```

`D1` *distributes the region* `R` *in blocks, with a set of consecutive points at each place, as evenly as possible.* `D2` *maps all the points in* `R` *to* `here`.

Let `D` be a distribution. `D.region` denotes the underlying region. Given a point `p`, the expression `D(p)` represents the application of `D` to `p`, that is, the place that `p` is mapped to by `D`. The evaluation of the expression `D(p)` throws an `ArrayIndexOutofBoundsException` if `p` does not lie in the underlying region.

`PlaceGroups`

A `PlaceGroup` represents an ordered set of `Places`. `PlaceGroups` exist for performance and scaleability: they are more efficient, in certain critical places, than general collections of `Place`. `PlaceGroup` implements `Sequence[Place]`, and thus provides familiar operations – `pg.size()` for the number of places, `pg.iterator()` to iterate over them, etc.

`PlaceGroup` is an abstract class. The concrete class `SparsePlaceGroup` is intended for a small group of places. `new SparsePlaceGroup(somePlace)` is a good `PlaceGroup` containing one place. `new SparsePlaceGroup(seqPlaces)` constructs a sparse place group from a Rail of places.

Operations returning distributions

Let `R` be a region, `Q` a `PlaceGroup`, and `P` a place.

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

Constant distributions. The distribution `Dist.makeConstant(R,P)` maps every point in region `R` to place `P`. The special case `Dist.makeConstant(R)` maps every point in `R` to `here`.

Block distributions. The distribution `Dist.makeBlock(R)` distributes the elements of `R`, in approximately-even blocks, over all the places available to the program. There are other `Dist.makeBlock` methods capable of controlling the distribution and the set of places used; see the API documentation.

Domain Restriction. If `D` is a distribution and `R` is a sub-region of `D.region`, then `D | R` represents the restriction of `D` to `R`—that is, the distribution that takes each point `p` in `R` to `D(p)`, but doesn't apply to any points but those in `R`.

Range Restriction. If `D` is a distribution and `P` a place expression, the term `D | P` denotes the sub-distribution of `D` defined over all the points in the region of `D` mapped to `P`.

Note that `D | here` does not necessarily contain adjacent points in `D.region`. For instance, if `D` is a cyclic distribution, `D | here` will typically contain points that differ by the number of places. An implementation may find a way to still represent them in contiguous memory, *e.g.*, using an arithmetic function to map from the region index to an index into the array.

16.4.4 Distributed Arrays

Distributed arrays, instances of `DistArray[T]`, are very much like `Array`s, except that they distribute information among multiple `Place`s according to a `Dist` value passed in as a constructor argument.

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 = Region.make(1..1000);
val D <: Dist = Dist.makeBlock(R);
val da <: DistArray[Float]
       = DistArray.make[Float](D, (Point(1))=>0.0f);
```

16.4.5 Distributed Array Construction

`DistArray`s are instantiated by invoking one of the `make` factory methods of the `DistArray` class. A `DistArray` creation must take either an `Long` as an argument or a `Dist`. In the first case, a distributed array is created over the distribution `Dist.makeConstant(Region.make(0, N-1),here)`; in the second over the given distribution.

Example: *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 ident = ([i]:Point(1)) => i;
val data : DistArray[Long]
    = DistArray.make[Long](Dist.makeConstant(Region.make(1, 9)), ident);
val blk = Dist.makeBlock(Region.make(1..9, 1..9));
val data2 : DistArray[Long]
    = DistArray.make[Long](blk, ([i,j]:Point(2)) => i*j);
```

The first declaration stores in `data` *a reference to a mutable distributed array with* `9` *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 9, distributed in blocks over all* `Place`*s, initialized with* `i*j` *at point* `[i,j]`.

16.4.6 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`.

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 access arrays, with `da(p)` or `da(p)=v`, at a place other than `da.dist(p)`, *viz.* at the place that the element exists.

Arrays of Single Values

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 `DistArrays` 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.

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

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.

The following produces an array of cubes:

```
val A = new Array[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
val B = A.map(cube);
assert B(3) == 27 && B(4) == 64 && B(10) == 1000;
```

A variant operation lets you specify the array `B` into which the result will be stored,

```
val A = new Array[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
val B = new Array[Long](A.region); // B = 0,0,0,0,0,0,0,0,0,0,0
A.map(B, cube);
assert B(3) == 27 && B(4) == 64 && B(10) == 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[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
A.map(A, cube);
assert A(3) == 27 && A(4) == 64 && A(10) == 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[Long], db: DistArray[Long])
    {da.dist==db.dist}
    = da.map(db, (a:Long,b:Long)=>a+b);
```

Reductions Let `f` be a function of type `(T,T)=>T`. Let `a` be an array over base type `T`. Let `unit` be a value of type `T`. Then the operation `a.reduce(f, unit)` returns a value of type `T` obtained by combining all the elements of `a` by use of `f` in some unspecified order (perhaps in parallel). The following code gives one method which meets the definition of `reduce`, having a running total `r`, and accumulating each value `a(p)` into it using `f` in turn. (This code is simply given as an example; `Array` has this operation defined already.)

```
def oneWayToReduce[T](a:Array[T], f:(T,T)=>T, unit:T):T {
  var r : T = unit;
  for(p in a.region) r = f(r, a(p));
  return r;
}
```

For example, the following sums an array of longs. `f` is addition, and `unit` is zero.

```
val a = new Array[Long](4, (i:long)=>i+1);
val sum = a.reduce((a:Long,b:Long)=>a+b, 0);
assert(sum == 10); // 10 == 1+2+3+4
```

Other orders of evaluation, degrees of parallelism, and applications of `f(x,unit)` and `f(unit,x)`are also correct. In order to guarantee that the result is precisely determined, the function `f` should be associative and commutative, and the value `unit` should satisfy `f(unit,x)` == `x` == `f(x,unit)` for all `x:T`.

`DistArray`s have the same operation. This operation involves communication between the places over which the `DistArray` is distributed. The X10 implementation guarantees that only one value of type `T` is communicated from a place as part of this reduction process.

Scans Let `f:(T,T)=>T`, `unit:T`, and `a` be an `Array[T]` or `DistArray[T]`. Then `a.scan(f,unit)` is the array or distributed array of type `T` whose ith element in canonical order is the reduction by `f` with unit `unit` of the first i elements of `a`.

This operation involves communication between the places over which the distributed array is distributed. The X10 implementation will endeavour to minimize the communication between places to implement this operation.

Other operations on arrays, distributed arrays, and the related classes may be found in the `x10.regionarray` 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 constraint-free interface types that decorate the abstract syntax tree of an X10 program. The X10 type-checker ensures that an annotation is a legal interface type. In X10, interfaces may declare both methods and properties. Therefore, like any interface type, an annotation may instantiate one or more of its interface's properties.

17.1 Annotation syntax

The annotation syntax consists of an "`@`" followed by an interface type.

Annotations	::=	*Annotation*	*(20.6)*
	|	*Annotations Annotation*	
Annotation	::=	`@` *NamedTypeNoConstraints*	*(20.4)*

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: Long): T { ... }
  ```

```
// 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(1n,4n)

Rail[Rail[Double]]@Size(n * n)
```

- Expression annotations:

```
m()  @RemoteCall
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(0)
@MaxIterations(n)
for (var i: Long = 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 Interoperability with Other Languages

The ability to interoperate with other programming languages is an essential feature of the X10 implementation. Cross-language interoperability enables both the incremental adoption of X10 in existing applications and the usage of existing libraries and frameworks by newly developed X10 programs.

There are two primary interoperability scenarios that are supported by X10 v2.4: inline substitution of fragments of foreign code for X10 program fragments (expressions, statements) and external linkage to foreign code.

18.1 Embedded Native Code Fragments

The `@Native(lang,code) Construct` annotation from `x10.compiler.Native` in X10 can be used to tell the X10 compiler to substitute `code` for whatever it would have generated when compiling `Construct` with the `lang` back end.

The compiler cannot analyze native code the same way it analyzes X10 code. In particular, `@Native` fields and methods must be explicitly typed; the compiler will not infer types.

18.1.1 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;
}
```

Types are *not* inferred for methods marked as `@Native`.

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, or the `this` variable.) 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. For Java, the static variables are used with their X10 names. For C++, the names must be mangled, by use of the `FMGL` macro.

```
import x10.compiler.Native;
class Ability {
  static val A : Int = 1n;
  @Native("java", "A+2")
  @Native("c++", "Ability::FMGL(A)+2")
  static native def fromStatic():Int;
}
```

18.1.2 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 variables from the surrounding context are available inside `stmt`. For example, the method call `born.example(10n)`, 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 = 1n;
  public def example(x:Int):Int{
    @Native("java", "y=x;")
    {y = 3n;}
    return y;
  }
}
```

Note that the code being replaced is a statement – the block `{y = 3n;}` 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 external 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.

18.2 Interoperability with External Java Code

With Managed X10, we can seamlessly call existing Java code from X10, and call X10 code from Java. We call this the *Java interoperability* [13] feature.

By combining Java interoperability with X10's distributed execution features, we can even make existing Java applications, which are originally designed to run on a single Java VM, scale-out with minor modifications.

18.2.1 How Java program is seen in X10

Managed X10 does not pre-process the existing Java code to make it accessible from X10. X10 programs compiled with Managed X10 will call existing Java code as is.

Types In X10, both at compile time and run time, there is no way to distinguish Java types from X10 types. Java types can be referred to with regular `import` statement, or their qualified names. The package `java.lang` is not auto-imported into X10. In Managed x10, the resolver is enhanced to resolve types with X10 source files in the source path first, then resolve them with Java class files in the class path. Note that the resolver does not resolve types with Java source files, therefore Java source files must be compiled in advance. To refer to Java types listed in Tables 18.1, and 18.2, which include all Java primitive types, use the corresponding X10 type (e.g. use `x10.lang.Int` (or in short, `Int`) instead of `int`).

Fields Fields of Java types are seen as fields of X10 types.

Managed X10 does not change the static initialization semantics of Java types, which is per-class, at load time, and per-place (Java VM), therefore, it is subtly different than the per-field lazy initialization semantics of X10 static fields.

Methods Methods of Java types are seen as methods of X10 types.

Generic types Generic Java types are seen as their raw types (§4.8 in [6]). Raw type is a mechanism to handle generic Java types as non-generic types, where the type parameters are assumed as `java.lang.Object` or their upperbound if they have it. Java introduced generics and raw type at the same time to facilitate generic Java code interfacing with non-generic legacy Java code. Managed X10 uses this mechanism for a slightly different purpose. Java erases type parameters at compile time, whereas X10 preserves their values at run time. To manifest this semantic gap in generics, Managed X10 represents Java generic types as raw types and eliminates type parameters at source code level. For more detailed discussions, please refer to [14, 15].

```
    import x10.interop.Java;
    public class XClass {
      public static def main(args:Rail[String]):void {
        try {
          val a = Java.newArray[Int](2n);
          a(0n) = 0n;
          a(1n) = 1n;
          a(2n) = 2n;
        } catch (e:x10.lang.ArrayIndexOutOfBoundsException) {
          Console.OUT.println(e);
        }
      }
    }
> x10c -d bin src/XClass.x10
> x10 -cp bin XClass
x10.lang.ArrayIndexOutOfBoundsException: Array index out of range: 2
```

Figure 18.1: Java exceptions in X10

Arrays X10 rail and array types are generic types whose representation is different from Java array types.

Managed X10 provides a special X10 type `x10.interop.Java.array[T]` which represents Java array type `T[]`. Note that for X10 types in Table 18.1, this type means the Java array type of their primary type. For example, `array[Int]` and `array[String]` mean `int[]` and `java.lang.String[]`, respectively. Managed X10 also provides conversion methods between X10 `Rail`s and Java arrays (`Java.convert[T](a:Rail[T]):array[T]` and `Java.convert[T](a:array[T]):Rail[T]`), and creation methods for Java arrays (`Java.newArray[T](d0:Int):array[T]` etc.).

Exceptions The X10 v2.4 exception hierarchy has been designed so that there is a natural correspondence with the Java exception hierarchy. As shown in Table 18.2, many commonly used Java exception types are directly mapped to X10 exception types. Exception types that are thus aliased may be caught (and thrown) using either their Java or X10 types. In X10 code, it is stylistically preferable to use the X10 type to refer to the exception as shown in Figure 18.1.

Compiling and executing X10 programs We can compile and run X10 programs that call existing Java code with the same `x10c` and `x10` command by specifying the location of Java class files or jar files that your X10 programs refer to, with `-classpath` (or in short, `-cp`) option.

18.2.2 How X10 program is translated to Java

Managed X10 translates X10 programs to Java class files.

X10 does not provide a Java reflection-like mechanism to resolve X10 types, methods, and fields with their names at runtime, nor a code generation tool, such as `javah`, to generate stub code to access them from other languages. Java programmers, therefore, need to access X10 types, methods, and fields in the generated Java code directly as they access Java types, methods, and fields. To make it possible, Java programmers need to understand how X10 programs are translated to Java.

Some aspects of the X10 to Java translation scheme may change in future version of X10; therefore in this document only a stable subset of translation scheme will be explained. Although it is a subset, it has been extensively used by X10 core team and proved to be useful to develop Java Hadoop interop layer for a Main-memory Map Reduce (M3R) engine [11] in X10.

In the following discussions, we deliberately ignore generic X10 types because the translation of generics is an area of active development and will undergo some changes in future versions of X10. For those who are interested in the implementation of generics in Managed X10, please consult [15]. We also don't cover function types, function values, and all non-static methods. Although slightly outdated, another paper [14], which describes translation scheme in X10 2.1.2, is still useful to understand the overview of Java code generation in Managed X10.

Types X10 classes and structs are translated to Java classes with the same names. X10 interfaces are translated to Java interfaces with the same names.

Table 18.1 shows the list of special types that are mapped to Java primitives. Primitives are their primary representations that are useful for good performance. Wrapper classes are used when the reference types are needed. Each wrapper class has two static methods `$box()` and `$unbox()` to convert its value from primary representation to wrapper class, and vice versa, and Java backend inserts their calls as needed. An you notice, every unsigned type uses the same Java primitive as its corresponding signed type for its representation.

Table 18.2 shows a non-exhaustive list of another kind of special types that are mapped (not translated) to Java types. As you notice, since an interface `Any` is mapped to a class —java.lang.Object— and `Object` is hidden from the language, there is no direct way to create an instance of `Object`. As a workaround, `Java.newObject()` is provided.

As you also notice, `x10.lang.Comparable[T]` is mapped to `java.lang.Comparable`. This is needed to map `x10.lang.String`, which implements `x10.lang.Compatable[String]`, to `java.lang.String` for performance, but as a trade off, this mapping results in the lost of runtime type information for `Comparable[T]` in Managed X10. The runtime of Managed X10 has built-in knowledge for `String`, but for other Java classes that implement `java.lang.Comparable`, `instanceof Comparable[Int]` etc. may return incorrect results. In principle, it is impossible to map X10 generic type to the existing Java generic type without losing runtime type information.

X10		Java (primary)		Java (wrapper class)
`x10.lang.Byte`	`1y`	`byte`	`(byte)1`	`x10.core.Byte`
`x10.lang.UByte`	`1uy`	`byte`	`(byte)1`	`x10.core.UByte`
`x10.lang.Short`	`1s`	`short`	`(short)1`	`x10.core.Short`
`x10.lang.UShort`	`1us`	`short`	`(short)1`	`x10.core.UShort`
`x10.lang.Int`	`1n`	`int`	`1`	`x10.core.Int`
`x10.lang.UInt`	`1un`	`int`	`1`	`x10.core.UInt`
`x10.lang.Long`	`1`	`long`	`1l`	`x10.core.Long`
`x10.lang.ULong`	`1u`	`long`	`1l`	`x10.core.ULong`
`x10.lang.Float`	`1.0f`	`float`	`1.0f`	`x10.core.Float`
`x10.lang.Double`	`1.0`	`double`	`1.0`	`x10.core.Double`
`x10.lang.Char`	`'c'`	`char`	`'c'`	`x10.core.Char`
`x10.lang.Boolean`	`true`	`boolean`	`true`	`x10.core.Boolean`

Table 18.1: X10 types that are mapped to Java primitives

X10	Java
`x10.lang.Any`	`java.lang.Object`
`x10.lang.Comparable[T]`	`java.lang.Comparable`
`x10.lang.String`	`java.lang.String`
`x10.lang.CheckedThrowable`	`java.lang.Throwable`
`x10.lang.CheckedException`	`java.lang.Exception`
`x10.lang.Exception`	`java.lang.RuntimeException`
`x10.lang.ArithmeticException`	`java.lang.ArithmeticException`
`x10.lang.ClassCastException`	`java.lang.ClassCastException`
`x10.lang.IllegalArgumentException`	`java.lang.IllegalArgumentException`
`x10.util.NoSuchElementException`	`java.util.NoSuchElementException`
`x10.lang.NullPointerException`	`java.lang.NullPointerException`
`x10.lang.NumberFormatException`	`java.lang.NumberFormatException`
`x10.lang.UnsupportedOperationException`	`java.lang.UnsupportedOperationException`
`x10.lang.IndexOutOfBoundsException`	`java.lang.IndexOutOfBoundsException`
`x10.lang.ArrayIndexOutOfBoundsException`	`java.lang.ArrayIndexOutOfBoundsException`
`x10.lang.StringIndexOutOfBoundsException`	`java.lang.StringIndexOutOfBoundsException`
`x10.lang.Error`	`java.lang.Error`
`x10.lang.AssertionError`	`java.lang.AssertionError`
`x10.lang.OutOfMemoryError`	`java.lang.OutOfMemoryError`
`x10.lang.StackOverflowError`	`java.lang.StackOverflowError`
`void`	`void`

Table 18.2: X10 types that are mapped (not translated) to Java types

```
class C {
  static val a:Int = ...;
  var b:Int;
}
interface I {
  val x:Int = ...;
}

class C {
  static int get$a() { return ...; }
  int b;
}
interface I {
  abstract static class $Shadow {
    static int get$x() { return ...; }
  }
}
```

Figure 18.2: X10 fields in Java

Fields As shown in Figure 18.2, instance fields of X10 classes and structs are translated to the instance fields of the same names of the generated Java classes. Static fields of X10 classes and structs are translated to the static methods of the generated Java classes, whose name has `get$` prefix. Static fields of X10 interfaces are translated to the static methods of the special nested class named `$Shadow` of the generated Java interfaces.

Note on name resolution In X10, fields (both static and instance), local variables, and types are in the same name space, and fields and local variables have higher precedence than types in resolving names. This is same as in Java and it is programmers' responsibility to avoid name conflict. Figure 18.3 explains how name conflict occurs. Uncommenting either a or b, or replacing c with c' causes *Field C not found in type "x10.lang.Long"* at x. Even if there is no name conflict in X10 code, it may occur in the generated Java code since Java backend generates static field access to get runtime type in some situation. To avoid potential name conflict, the best practice is not to use the same name for variables and package prefix.

Methods As shown in Figure 18.4, methods of X10 classes or structs are translated to the methods of the same names of the generated Java classes. Methods of X10 interfaces are translated to the methods of the same names of the generated Java interfaces.

Every method whose return type has two representations, such as the types in Table 18.1, will have `$O` suffix with its name. For example, `def f():Int` in X10 will

Main.x10:

```
public class Main {
  //static val p:Long = 1; // a
  //val p:Long = 1;        // b
  //def func(p:Long) {     // c'
  def func() {             // c
    val f = p.C.f;         // x
  }
}
```

p/C.x10:

```
package p;
public class C {
  public static val f = true;
}
```

Figure 18.3: Name conflict in X10

be compiled as `int f$O()` in Java. For those who are interested in the reason, please look at our paper [15].

Compiling Java programs To compile Java program that calls X10 code, you should use `x10cj` command instead of javac (or whatever your Java compiler). It invokes the post Java-compiler of `x10c` with the appropriate options. You need to specify the location of X10-generated class files or jar files that your Java program refers to.

```
x10cj -cp MyX10Lib.jar MyJavaProg.java
```

Executing Java programs Before executing any X10-generated Java code, the runtime of Managed X10 needs to be set up properly at each place. To set up the runtime, a special launcher named `runjava` is used to run Java programs. All Java programs that call X10 code should be launched with it, and no other mechanisms, including direct execution with `java` command, are supported.

```
Usage: runjava <Java-main-class> [arg0 arg1 ...]
```

18.3 Interoperability with External C and C++ Code

C and C++ code can be linked to X10 code, either by writing auxiliary C++ files and adding them with suitable annotations, or by linking libraries.

```
interface I {
  def f():Int;
  def g():Any;
}
class C implements I {
  static def s():Int = 0n;
  static def t():Any = null;
  public def f():Int = 1n;
  public def g():Any = null;
}

interface I {
  int f$O();
  java.lang.Object g();
}
class C implements I {
  static int s$O() { return 0; }
  static java.lang.Object t() { return null; }
  public int f$O() { return 1; }
  public java.lang.Object g() { return null; }
}
```

Figure 18.4: X10 methods in Java

18.3.1 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:Rail[String]) {
        { @Native("c++","foo();") {} }
    }
}
```

18.3.2 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 primary mechanism used for this is the `-cxx-prearg` and `-cxx-postarg` command line arguments to `x10c++`. The values of any `-cxx-prearg` commands are placed in the post compiler command before the list of .cc files to compile. The values of any `-cxx-postarg` commands are placed in the post compiler command after the list of .cc files to compile. Typically pre-args are arguments intended for the C++ compiler itself, while post-args are arguments intended for the linker.

The following example shows how to compile `blas` into the executable via these commands. The command must be issued on one line.

```
x10c++ Test.x10 -cxx-prearg -I/usr/local/blas
  -cxx-postarg -L/usr/local/blas -cxx-postarg -lblas'
```

19 Definite Assignment

X10 requires that every variable be set before it is read. Sometimes this is easy, as when a variable is declared and assigned together:

```
  var x : Long = 0;
  assert x == 0;
```

However, it is convenient to allow programs to make decisions before initializing variables.

```
def example(a:Long, b:Long) {
  val max:Long;
  //ERROR: assert max==max; // can't read 'max'
  if (a > b) max = a;
  else max = b;
  assert max >= a && max >= b;
}
```

This is particularly useful for `val` variables. `var`s could be initialized to a default value and then reassigned with the right value. `val`s must be initialized once and cannot be changed, so they must be initialized with the correct value.

However, one must be careful – and the X10 compiler enforces this care. Without the `else` clause, the preceding code might not give `max` a value by the time `assert` is invoked.

This leads to the concept of *definite assignment* [5]. A variable is *definitely assigned* at a point in code if, no matter how that point in code is reached, the variable has been assigned to. In X10, variables must be definitely assigned before they can be read.

As X10 requires that `val` variables *not* be initialized twice, we need the dual concept as well. A variable is *definitely unassigned* at a point in code if it cannot have been assigned no matter how that point in code is reached. For example, immediately after `val x:Long`, `x` is definitely unassigned.

Finally, we need the concept of *singly* and *multiply assigned*. A variable is singly assigned in a block if it is assigned precisely once; it is multiply assigned if it could possibly be assigned more than once. `var`s can multiply assigned as desired. `val`s must be singly assigned. For example, the code `x = 1; x = 2;` is perfectly fine if `x` is a `var`, but incorrect (even in a constructor) if `x` is a `val`.

At some points in code, a variable might be neither definitely assigned nor definitely unassigned. Such states are not always useful.

```
def example(flag : Boolean) {
  var x : Long;
  if (flag) x = 1;
  // x is neither def. assigned nor unassigned.
  x = 2;
  // x is def. assigned.
```

This shows that we cannot simply define "definitely unassigned" as "not definitely assigned". If `x` had been a `val` rather than a `var`, the previous example would not be allowed.

Unfortunately, a completely accurate definition of "definitely assigned" or "definitely unassigned" is undecidable – impossible for the compiler to determine. So, X10 takes a *conservative approximation* of these concepts. If X10's definition says that `x` is definitely assigned (or definitely unassigned), then it will be assigned (or not assigned) in every execution of the program.

However, there are programs which X10's algorithm says are incorrect, but which actually would behave properly if they were executed. In the following example, `flag` is either `true` or `false`, and in either case `x` will be initialized. However, X10's analysis does not understand this — thought it *would* understand if the example were coded with an `if-else` rather than a pair of `if`s. So, after the two `if` statements, `x` is not definitely assigned, and thus the `assert` statement, which reads it, is forbidden.

```
def example(flag:Boolean) {
  var x : Long;
  if (flag) x = 1;
  if (!flag) x = 2;
  // ERROR: assert x < 3;
}
```

19.1 Asynchronous Definite Assignment

Local variables and instance fields allow *asynchronous assignment*. A local variable can be assigned in an `async` statement, and, when the `async` is `finish`ed, the variable is definitely assigned.

Example:

```
val a : Long;
finish {
  async {
    a = 1;
  }
  // a is not definitely assigned here
}
// a is definitely assigned after 'finish'
assert a==1;
```

This concept supports a core X10 programming idiom. A `val` variable may be initialized asynchronously, thereby providing a means for returning a value from an `async` to be used after the enclosing `finish`.

19.2 Characteristics of Definite Assignment

The properties "definitely assigned", "singly assigned", and "definitely unassigned" are computed by a conservative approximation of X10's evaluation rules.

The precise details are up to the implementation. Many basic cases must be handled accurately; *e.g.*, `x=1;` definitely and singly assigns `x`.

However, in more complicated cases, a conforming X10 may mark as invalid some code which, when executed, would actually be correct. For example, the following program fragment will always result in `x` being definitely and singly assigned:

```
val x : Long;
var b : Boolean = mysterious();
if (b) x = cryptic();
if (!b) x = unknown();
```

However, most conservative approximations of program execution won't mark `x` as properly initialized, though it is. For `x` to be properly initialized, precisely one of the two assignments to `x` must be executed. If `b` were true initially, it would still be true after the call to `cryptic()` — since methods cannot modify their caller's local variables – and so the first but not the second assignment would happen. If `b` were false initially, it would still be false when `!b` is tested, and so the second but not the first assignment would happen. Either way, `x` is definitely and singly assigned.

However, for a slightly different program, this analysis would be wrong. *E.g.*, if `b` were a field of `this` rather than a local variable, `cryptic()` could change `b`; if `b` were true initially, both assignments might happen, which is incorrect for a `val`.

This sort of reasoning is beyond most conservative approximation algorithms. (Indeed, many do not bother checking that `!b` late in the program is the opposite of `b` earlier.) Algorithms that pay attention to such details and subtleties tend to be fairly expensive, which would lead to very slow compilation for X10 – for the sake of obscure cases.

X10's analysis provides at least the following guarantees. We describe them in terms of a statement `S` performing some collection of possible numbers of assignments to variables — on a scale of "0", "1", and "many". For example, `if (b) x=1; else {x=1;x=2;y=2;}` might assign to `x` one or many times, and might assign to `y` zero or one time. Hence, after it, `x` is definitely assigned and may be multiply assigned, and `y` is neither definitely assigned nor definitely unassigned.

These descriptions are combined in natural ways. For example, if `R` says that `x` will be assigned 0 or 1 times, and `S` says it will be assigned precisely once, then `R;S` will assign it one or many times. If only one or `R` or `S` will occur, as from `if (b) R; else S;`, then `x` may be assigned 0 or 1 times.

This information is sufficient for the tests X10 makes. If `x` can is assigned one or many times in `S`, it is definitely assigned. It is an error if `x` is ever read at a point where it have been assigned zero times. It is an error if a `val` may be assigned many times.

We do not guarantee that any particular X10 compiler uses this algorithm; indeed, as of the time of writing, the X10 compiler uses a somewhat more precise one. However, any conformant X10 compiler must provide results which are at least as accurate as this analysis.

Assignment: `x = e`

`x = e` assigns to `x`, in addition to whatever assignments `e` makes. For example, if `this.setX(y)` sets a field `x` to `y` and returns `y`, then `x = this.setX(y)` definitely and multiply assigns `x`.

`async` **and** `finish`

By itself, `async S` provides few guarantees. After an activity executes `async{x=1;}` we know that there is a separate activity which (on being scheduled) will set `x` to `1`. We do not know that this has happened yet.

However, if there is a `finish` around the `async`, the situation is clearer. After `finish async x=1;`, `x` has definitely been assigned.

In general, if an `async S` appears in the body of a `finish` in a way that guarantees that it will be executed, then, after the `finish`, the assignments made by `S` will have occurred. For example, if `S` definitely assigns to `x`, and the body of the `finish` guarantees that `async S` will be executed, then `finish{...async S...}` definitely assigns `x`.

`if` **and** `switch`

When `if(E) S else T` finishes, it will have performed the assignments of `E`, together with those of either `S` or `T` but not both. For example, `if (b) x=1; else x=2;` definitely assigns `x`, but `if (b) x=1;` does not.

`switch` is more complex, but follows the same principles as `if`. For example, `switch(E){case 1: A; break; case 2: B; default: C;}` performs the assignments of `E`, and those of precisely one of `A`, or `B;C`, or `C`. Note that case `2` falls through to the default case, so it performs the same assignments as `B;C`.

Sequencing

When `R;S` finishes, it will have performed the assignments of `R` and those of `S`, if `R` and `S` terminate normally. If `R` terminates abruptly, then only the assignments of `R` executed till the point of termination will have been executed. if `R` terminates normally, but `S` terminates abruptly then the assignments of `R` will have been executed and those of `S` executed till the point of termination.

For example, `x=1;y=2;` definitely assigns `x` and `y`, and `x=1;x=2;` multiply assigns `x`.

Loops

`while(E)S` performs the assignments of `E` one or more times, and those of `S` zero or more times. For example, if `while(b()){x=1;}` might assign to `x` zero, one, or many times. `do S while(E)` performs the assignments of `E` one or more times, and those of `S` one or more times.

`for(A;B;C)D` performs the assignments of `A` once, those of `B` one or more times, and those of `C` and `D` one or more times. `for(x in E)S` performs the assignments of `E` once and those of `S` zero or more times.

Loops are of very little value for providing definite assignments, since X10 does not in general know how many times they will be executed.

`continue` and `break` inside of a loop are hard to describe in simple terms. They may be conservatively assumed to cause the loop to give no information about the variables assigned inside of it. For example, the analysis may conservatively conclude that `do{ x = 1; if (true) break; } while(true)` may assign to `x` zero, one, or many times, overlooking the more precise fact that it is assigned once.

Method Calls

A method call `E.m(A,B)` performs the assignments of `E`, `A`, and `B` once each, and also those of `m`. This implies that X10 must be aware of the possible assignments performed by each method.

If X10 has complete information about `m` (as when `m` is a `private` or `final` method), this is straightforward. When such information is fundamentally impossible to acquire, as when `m` is a non-final method invocation, X10 has no choice but to assume that `m` might do anything that a method can do. (For this reason, the only methods that can be called from within a constructor on a raw – incompletely-constructed – object) are the `private` and `final` ones.)

- `m` cannot assign to local variables of the caller; methods have no such power.

- Let `m` be an instance method. `m` can assign to `var` fields of `this` freely,
- Let `m` be an instance method. `m` cannot initialize `val` fields of `this`. (But see §8.5.2; when one constructor calls another as the first statement of its body, the other constructor can initialize `vval` fields. This is a constructor call, not a method call.)

Recall that every container must be fully initialized upon exit from its constructor. X10 places certain restrictions on which methods can be called from a constructor; see §8.11.1. One of these restrictions is that methods called before object initialization is complete must be `final` or `private` — and hence, available for static analysis. So, when checking field initialization, X10 will ensure:

1. Each `val` field is initialized before it is read. A method that does not read a `val` field `f` *may* be called before `f` is initialized; a method that reads `f` must not be called until `f` is initialized. For example, a constructor may have the form:

   ```
   class C {
     val f : Long;
     val g : String;
     def this() {
        f = fless();
        g = useF();
     }
     private def fless() = "f not used here".length();
     private def useF() = "f=" + this.f;
   }
   ```

2. `var` fields require a deeper analysis. Consider a `var` field `var x:T` without initializer. If `T` has a default value, `x` may be read inside of a constructor before it is otherwise written, and it will have its default value.

   If `T` has no default value, an analysis like that used for `val`s must be performed to determine that `x` is initialized before it is used. The situation is more complex than for `val`s, however, because a method can assign to `x` as well read from it. The X10 compiler computes a conservative approximation of which methods read and write which `var` fields. (Doing this carefully requires finding a solution of a set of equations over sets of variables, with each callable method having equations describing what it reads and writes.)

`at`

`at(p)S` performs precisely the assignments of `p` and those of `S`. Note that `S` is executed at the place named by `p` in an environment in which all variables used in `S` but defined outside `S` are bound to copies (made at `p`) of the values they had at the `at(p)S` statement (§13.3).

`atomic`

`atomic S` performs the assignments of `S`, and `when(E)S` performs those of `E` and `S`. Note that `E` or `S` may terminate abruptly.

`try`

`try S catch(x:T1) E1 ... catch(x:Tn) En finally F` performs some or all of the assignments of `S`, plus all the assignments of zero or one of the `E`'s, plus those of `F`. For example,

```
try {
  x = boomy();
  x = 0;
}
catch(e:Boom) { y = 1; }
finally { z = 1; }
```

assigns `x` zero, one, or many times[1], assigns `y` zero or one time, and assigns `z` exactly once.

Expression Statements

Expression statements `E;`, and other statements that execute an expression and do something innocuous with it (local variable declaration and `assert`) have the same effects as `E`.

`return`, `throw`

Statements that do not finish normally, such as `return` and `throw`, do not initialize anything (though the computation of the return or thrown value may). They also terminate a line of computation. For example, `if(b) {x=1; return;}  x=2;` definitely and singly assigns `x`.

[1] A more precise analysis could discover that `x` cannot be initialized only once.

20 Grammar

In this grammar, $X^?$ denotes an optional X element.

(0)	*AdditiveExp*	::=	*MultiplicativeExp*
		|	*AdditiveExp* + *MultiplicativeExp*
		|	*AdditiveExp* - *MultiplicativeExp*
(1)	*AndExp*	::=	*EqualityExp*
		|	*AndExp* `&` *EqualityExp*
(2)	*AnnotatedType*	::=	*Type Annotations*
(3)	*Annotation*	::=	`@` *NamedTypeNoConstraints*
(4)	*AnnotationStmt*	::=	*Annotations*$^?$ *NonExpStmt*
(5)	*Annotations*	::=	*Annotation*
		|	*Annotations Annotation*
(6)	*ApplyOpDecln*	::=	*MethMods* `operator` `this` *TypeParams*$^?$ *Formals* *Guard*$^?$ *HasResultType*$^?$ *MethodBody*
(7)	*ArgumentList*	::=	*Exp*
		|	*ArgumentList* `,` *Exp*
(8)	*Arguments*	::=	`(` *ArgumentList* `)`
(9)	*AssertStmt*	::=	`assert` *Exp* `;`
		|	`assert` *Exp* `:` *Exp* `;`
(10)	*AssignPropCall*	::=	`property` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` `;`

(11) *Assignment* ::= *LeftHandSide AsstOp AsstExp*
| *ExpName* (*ArgumentList*$^?$) *AsstOp AsstExp*
| *Primary* (*ArgumentList*$^?$) *AsstOp AsstExp*

(12) *AsstExp* ::= *Assignment*
| *ConditionalExp*

(13) *AsstOp* ::= `=`
| `*=`
| `/=`
| `%=`
| `+=`
| `-=`
| `<<=`
| `>>=`
| `>>>=`
| `&=`
| `^=`
| `|=`

(14) *AsyncStmt* ::= `async` *ClockedClause*$^?$ *Stmt*
| `clocked` `async` *Stmt*

(15) *AtCaptureDeclr* ::= *Mods*$^?$ *VarKeyword*$^?$ *VariableDeclr*
| *Id*
| `this`

(16) *AtCaptureDeclrs* ::= *AtCaptureDeclr*
| *AtCaptureDeclrs* , *AtCaptureDeclr*

(17) *AtEachStmt* ::= `ateach` (*LoopIndex* `in` *Exp*) *ClockedClause*$^?$ *Stmt*
| `ateach` (*Exp*) *Stmt*

(18) *AtExp* ::= `at` (*Exp*) *ClosureBody*

(19) *AtStmt* ::= `at` (*Exp*) *Stmt*

(20) *AtomicStmt* ::= `atomic` *Stmt*

(21) *BasicForStmt* ::= `for` (*ForInit*$^?$; *Exp*$^?$; *ForUpdate*$^?$) *Stmt*

```
(22)  BinOp   ::=  +
               |   -
               |   *
               |   /
               |   %
               |   &
               |   |
               |   ^
               |   &&
               |   ||
               |   <<
               |   >>
               |   >>>
               |   >=
               |   <=
               |   >
               |   <
               |   ==
               |   !=
               |   ..
               |   ->
               |   <-
               |   -<
               |   >-
               |   **
               |   ~
               |   !~
               |   !
```

```
(23)  BinOpDecln   ::=  MethMods operator TypeParams? ( Formal ) BinOp ( Formal )
                        Guard? HasResultType? MethodBody
                    |   MethMods operator TypeParams? this BinOp ( Formal ) Guard?
                        HasResultType? MethodBody
                    |   MethMods operator TypeParams? ( Formal ) BinOp this Guard?
                        HasResultType? MethodBody
```

```
(24)  Block   ::=  { BlockStmts? }
```

```
(25)  BlockInteriorStmt   ::=  LocVarDeclnStmt
                           |   ClassDecln
                           |   StructDecln
                           |   TypeDefDecln
                           |   Stmt
```

(26)	*BlockStmts*	::=	*BlockInteriorStmt*
		|	*BlockStmts BlockInteriorStmt*
(27)	*BooleanLiteral*	::=	`true`
		|	`false`
(28)	*BreakStmt*	::=	`break` *Id*$^?$ `;`
(29)	*CastExp*	::=	*Primary*
		|	*ExpName*
		|	*CastExp* `as` *Type*
(30)	*CatchClause*	::=	`catch` `(` *Formal* `)` *Block*
(31)	*Catches*	::=	*CatchClause*
		|	*Catches CatchClause*
(32)	*ClassBody*	::=	`{` *ClassMemberDeclns*$^?$ `}`
(33)	*ClassDecln*	::=	*Mods*$^?$ `class` *Id TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *Super*$^?$ *Interfaces*$^?$ *ClassBody*
(34)	*ClassMemberDecln*	::=	*InterfaceMemberDecln*
		|	*CtorDecln*
(35)	*ClassMemberDeclns*	::=	*ClassMemberDecln*
		|	*ClassMemberDeclns ClassMemberDecln*
(36)	*ClassName*	::=	*TypeName*
(37)	*ClassType*	::=	*NamedType*
(38)	*ClockedClause*	::=	`clocked` *Arguments*
(39)	*ClosureBody*	::=	*Exp*
		|	*ClosureBodyBlock*
(40)	*ClosureBodyBlock*	::=	*Annotations*$^?$ `{` *BlockStmts*$^?$ *LastExp* `}`
		|	*Annotations*$^?$ *Block*

(41) *ClosureExp* ::= *Formals Guard*$^?$ *HasResultType*$^?$ => *ClosureBody*

(42) *CompilationUnit* ::= *PackageDecln*$^?$ *TypeDeclns*$^?$
| *PackageDecln*$^?$ *ImportDeclns TypeDeclns*$^?$
| *ImportDeclns PackageDecln ImportDeclns*$^?$ *TypeDeclns*$^?$
| *PackageDecln ImportDeclns PackageDecln ImportDeclns*$^?$ *TypeDeclns*$^?$

(43) *ConditionalAndExp* ::= *InclusiveOrExp*
| *ConditionalAndExp* && *InclusiveOrExp*

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

(45) *ConditionalOrExp* ::= *ConditionalAndExp*
| *ConditionalOrExp* || *ConditionalAndExp*

(46) *ConstantExp* ::= *Exp*

(47) *ConstrainedType* ::= *NamedType*
| *AnnotatedType*

(48) *ConstraintConjunction* ::= *Exp*
| *ConstraintConjunction* , *Exp*

(49) *ContinueStmt* ::= `continue` *Id*$^?$;

(50) *ConversionOpDecln* ::= *ExplConvOpDecln*
| *ImplConvOpDecln*

(51) *CtorBlock* ::= { *ExplicitCtorInvo*$^?$ *BlockStmts*$^?$ }

(52) *CtorBody* ::= *CtorBlock*
| = *ExplicitCtorInvo*
| = *AssignPropCall*
| ;

(53) *CtorDecln* ::= *Mods*$^?$ `def` `this` *TypeParams*$^?$ *Formals Guard*$^?$ *HasResultType*$^?$ *CtorBody*

(54) *DepNamedType* ::= *SimpleNamedType DepParams*
| *ParamizedNamedType DepParams*

(55) *DoStmt* ::= `do` *Stmt* `while` `(` *Exp* `)` `;`

(56) *EmptyStmt* ::= `;`

(57) *EnhancedForStmt* ::= `for` `(` *LoopIndex* `in` *Exp* `)` *Stmt*
| `for` `(` *Exp* `)` *Stmt*

(58) *EqualityExp* ::= *RelationalExp*
| *EqualityExp* == *RelationalExp*
| *EqualityExp* != *RelationalExp*
| *Type* == *Type*
| *EqualityExp* ~ *RelationalExp*
| *EqualityExp* !~ *RelationalExp*

(59) *ExclusiveOrExp* ::= *AndExp*
| *ExclusiveOrExp* ^ *AndExp*

(60) *Exp* ::= *AsstExp*

(61) *ExpName* ::= *Id*
| *FullyQualifiedName* `.` *Id*

(62) *ExpStmt* ::= *StmtExp* `;`

(63) *ExplConvOpDecln* ::= *MethMods* `operator` *TypeParams*$^?$ `(` *Formal* `)` `as` *Type Guard*$^?$ *MethodBody*
| *MethMods* `operator` *TypeParams*$^?$ `(` *Formal* `)` `as` `?` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*

(64) *ExplicitCtorInvo* ::= `this` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` `;`
| `super` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` `;`
| *Primary* `.` `this` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` `;`
| *Primary* `.` `super` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` `;`

(65) *ExtendsInterfaces* ::= `extends` *Type*
| *ExtendsInterfaces* `,` *Type*

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

(67) *FieldDecln* ::= *Mods*$^?$ *VarKeyword FieldDeclrs* `;`
| *Mods*$^?$ *FieldDeclrs* `;`

(68) *FieldDeclr* ::= *Id HasResultType*
| *Id HasResultType*$^?$ = *VariableInitializer*

(69) *FieldDeclrs* ::= *FieldDeclr*
| *FieldDeclrs* `,` *FieldDeclr*

(70) *Finally* ::= `finally` *Block*

(71) *FinishStmt* ::= `finish` *Stmt*
| `clocked` `finish` *Stmt*

(72) *ForInit* ::= *StmtExpList*
| *LocVarDecln*

(73) *ForStmt* ::= *BasicForStmt*
| *EnhancedForStmt*

(74) *ForUpdate* ::= *StmtExpList*

(75) *Formal* ::= *Mods*$^?$ *FormalDeclr*
| *Mods*$^?$ *VarKeyword FormalDeclr*
| *Type*

(76) *FormalDeclr* ::= *Id ResultType*
| `[` *IdList* `]` *ResultType*
| *Id* `[` *IdList* `]` *ResultType*

(77) *FormalDeclrs* ::= *FormalDeclr*
| *FormalDeclrs* `,` *FormalDeclr*

(78) *FormalList* ::= *Formal*
| *FormalList* `,` *Formal*

(79) *Formals* ::= `(` *FormalList*$^?$ `)`

(80) *FullyQualifiedName* ::= *Id*
| *FullyQualifiedName* `.` *Id*

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

(82) *Guard* ::= *DepParams*

(83) *Throws* ::= `throws` *ThrowsList*

(84) *ThrowsList* ::= *Type*
| *ThrowsList* `,` *Type*

(85) *HasResultType* ::= *ResultType*
| `<:` *Type*

(86) *HasZeroConstraint* ::= *Type* `haszero`

(87) *HomeVariable* ::= *Id*
| `this`

(88) *HomeVariableList* ::= *HomeVariable*
| *HomeVariableList* `,` *HomeVariable*

(89) *Id* ::= `IDENTIFIER`

(90) *IdList* ::= *Id*
| *IdList* `,` *Id*

(91) *IfThenElseStmt* ::= `if` `(` *Exp* `)` *Stmt* `else` *Stmt*

(92) *IfThenStmt* ::= `if` `(` *Exp* `)` *Stmt*

(93) *ImplConvOpDecln* ::= *MethMods* `operator` *TypeParams*$^?$ `(` *Formal* `)` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*

(94) *ImportDecln* ::= *SingleTypeImportDecln*
| *TypeImportOnDemandDecln*

(95) *ImportDeclns* ::= *ImportDecln*
| *ImportDeclns ImportDecln*

(96) *InclusiveOrExp* ::= *ExclusiveOrExp*
| *InclusiveOrExp* `|` *ExclusiveOrExp*

(97) *InterfaceBody* ::= `{` *InterfaceMemberDeclns*$^?$ `}`

(98) *InterfaceDecln* ::= *Mods*$^?$ `interface` *Id* *TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *ExtendsInterfaces*$^?$ *InterfaceBody*

(99) *InterfaceMemberDecln* ::= *MethodDecln*
| *PropMethodDecln*
| *FieldDecln*
| *TypeDecln*

(100) *InterfaceMemberDeclns* ::= *InterfaceMemberDecln*
| *InterfaceMemberDeclns InterfaceMemberDecln*

(101) *InterfaceTypeList* ::= *Type*
| *InterfaceTypeList* `,` *Type*

(102) *Interfaces* ::= `implements` *InterfaceTypeList*

(103) *KeywordOp* ::= `for`
| `if`
| `try`
| `throw`
| `async`
| `atomic`
| `when`
| `finish`
| `at`
| `continue`
| `break`
| `ateach`
| `while`
| `do`

(104) *KeywordOpDecln* ::= *MethMods* `operator` *keywordOp* *TypeParams*$^?$ *Formals* *Guard*$^?$ *Throws*$^?$ *HasResultType*$^?$ *MethodBody*

(105) *LabeledStmt* ::= *Id* `:` *LoopStmt*

(106) *LastExp* ::= *Exp*

(107) *LeftHandSide* ::= *ExpName*
| *FieldAccess*

(108) *Literal* ::= `IntegerLiteral`
| `LongLiteral`
| `ByteLiteral`
| `UnsignedByteLiteral`
| `ShortLiteral`
| `UnsignedShortLiteral`
| `UnsignedIntegerLiteral`
| `UnsignedLongLiteral`
| `FloatingPointLiteral`
| `DoubleLiteral`
| *BooleanLiteral*
| `CharacterLiteral`
| `StringLiteral`
| `null`

(109) *LocVarDecln* ::= *Mods*$^?$ *VarKeyword VariableDeclrs*
| *Mods*$^?$ *VarDeclsWType*
| *Mods*$^?$ *VarKeyword FormalDeclrs*

(110) *LocVarDeclnStmt* ::= *LocVarDecln* `;`

(111) *LoopIndex* ::= *Mods*$^?$ *LoopIndexDeclr*
| *Mods*$^?$ *VarKeyword LoopIndexDeclr*

(112) *LoopIndexDeclr* ::= *Id HasResultType*$^?$
| `[` *IdList* `]` *HasResultType*$^?$
| *Id* `[` *IdList* `]` *HasResultType*$^?$

(113) *LoopStmt* ::= *ForStmt*
| *WhileStmt*
| *DoStmt*
| *AtEachStmt*

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

(115) *MethodBody* ::= = *LastExp* `;`
| *Annotations*$^?$ *Block*
| `;`

(116) *MethodDecln* ::= *MethMods* `def` *Id TypeParams*$^?$ *Formals Guard*$^?$ *Throws*$^?$ *HasResultType*$^?$ *MethodBody*
| *BinOpDecln*
| *PrefixOpDecln*
| *ApplyOpDecln*
| *SetOpDecln*
| *ConversionOpDecln*
| *KeywordOpDecln*

(117) *MethodInvo* ::= *MethodName TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`
| *Primary* `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`
| `super` `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`
| *ClassName* `.` `super` `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`
| *Primary TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`

(118) *MethodInvoStmt* ::= *MethodName TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*
| *Primary* `.` *Id TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*
| `super` `.` *Id TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*
| *ClassName* `.` `super` `.` *Id TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*
| *Primary TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*

(119) *MethodName* ::= *Id*
| *FullyQualifiedName* `.` *Id*

(120) *Mod* ::= `abstract`
| *Annotation*
| `atomic`
| `final`
| `native`
| `private`
| `protected`
| `public`
| `static`
| `transient`
| `clocked`

(121) MultiplicativeExp ::= RangeExp
| MultiplicativeExp * RangeExp
| MultiplicativeExp / RangeExp
| MultiplicativeExp `%` RangeExp
| MultiplicativeExp `**` RangeExp

(122) NamedType ::= NamedTypeNoConstraints
| DepNamedType

(123) NamedTypeNoConstraints ::= SimpleNamedType
| ParamizedNamedType

(124) NonExpStmt ::= Block
| EmptyStmt
| AssertStmt
| SwitchStmt
| DoStmt
| BreakStmt
| ContinueStmt
| ReturnStmt
| ThrowStmt
| TryStmt
| LabeledStmt
| IfThenStmt
| IfThenElseStmt
| WhileStmt
| ForStmt
| AsyncStmt
| AtStmt
| AtomicStmt
| WhenStmt
| AtEachStmt
| FinishStmt
| AssignPropCall
| userStmtPrefix userStmt
| MethodInvoStmt

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

(126) PackageDecln ::= Annotations$^?$ `package` PackageName `;`

```
(127)  PackageName  ::=  Id
                      |  PackageName . Id

(128)  PackageOrTypeName  ::=  Id
                            |  PackageOrTypeName . Id

(129)  ParamizedNamedType  ::=  SimpleNamedType Arguments
                             |  SimpleNamedType TypeArgs
                             |  SimpleNamedType TypeArgs Arguments

(130)  PostDecrementExp  ::=  PostfixExp --

(131)  PostIncrementExp  ::=  PostfixExp ++

(132)  PostfixExp  ::=  CastExp
                    |  PostIncrementExp
                    |  PostDecrementExp

(133)  PreDecrementExp  ::=  -- UnaryExpNotPlusMinus

(134)  PreIncrementExp  ::=  ++ UnaryExpNotPlusMinus

(135)  PrefixOp  ::=  +
                  |  -
                  |  !
                  |  ~
                  |  ^
                  |  |
                  |  &
                  |  *
                  |  /
                  |  %

(136)  PrefixOpDecln  ::=  MethMods operator TypeParams? PrefixOp ( Formal ) Guard?
                           HasResultType? MethodBody
                       |  MethMods operator TypeParams? PrefixOp this Guard?
                           HasResultType? MethodBody
```

(137) *Primary* ::= `here`
| `[` *ArgumentList*$^?$ `]`
| *Literal*
| `self`
| `this`
| *ClassName* `.` `this`
| `(` *Exp* `)`
| *ObCreationExp*
| *FieldAccess*
| *MethodInvo*

(138) *Prop* ::= *Annotations*$^?$ *Id ResultType*

(139) *PropList* ::= *Prop*
| *PropList* `,` *Prop*

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

(141) *Properties* ::= `(` *PropList* `)`

(142) *RangeExp* ::= *UnaryExp*
| *RangeExp* `..` *UnaryExp*

(143) *RelationalExp* ::= *ShiftExp*
| *HasZeroConstraint*
| *SubtypeConstraint*
| *RelationalExp* < *ShiftExp*
| *RelationalExp* > *ShiftExp*
| *RelationalExp* <= *ShiftExp*
| *RelationalExp* >= *ShiftExp*
| *RelationalExp* `instanceof` *Type*

(144) *ResultType* ::= `:` *Type*

(145) *ReturnStmt* ::= `return` *Exp*$^?$ `;`

(146) *SetOpDecln* ::= *MethMods* `operator` `this` *TypeParams*$^?$ *Formals* `=` `(` *Formal* `)` *Guard*$^?$ *HasResultType*$^?$ *MethodBody*

(147)	*ShiftExp*	::=	*AdditiveExp*
		|	*ShiftExp* `<<` *AdditiveExp*
		|	*ShiftExp* `>>` *AdditiveExp*
		|	*ShiftExp* `>>>` *AdditiveExp*
		|	*ShiftExp* `->` *AdditiveExp*
		|	*ShiftExp* `<-` *AdditiveExp*
		|	*ShiftExp* `-<` *AdditiveExp*
		|	*ShiftExp* `>-` *AdditiveExp*
		|	*ShiftExp* `!` *AdditiveExp*
(148)	*SimpleNamedType*	::=	*TypeName*
		|	*Primary* `.` *Id*
		|	*ParamizedNamedType* `.` *Id*
		|	*DepNamedType* `.` *Id*
(149)	*SingleTypeImportDecln*	::=	`import` *TypeName* `;`
(150)	*Stmt*	::=	*AnnotationStmt*
		|	*ExpStmt*
(151)	*StmtExp*	::=	*Assignment*
		|	*PreIncrementExp*
		|	*PreDecrementExp*
		|	*PostIncrementExp*
		|	*PostDecrementExp*
		|	*MethodInvo*
		|	*ObCreationExp*
(152)	*StmtExpList*	::=	*StmtExp*
		|	*StmtExpList* `,` *StmtExp*
(153)	*StructDecln*	::=	*Mods*$^?$ `struct` *Id* *TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *Interfaces*$^?$ *ClassBody*
(154)	*SubtypeConstraint*	::=	*Type* `<:` *Type*
		|	*Type* `:>` *Type*
(155)	*Super*	::=	`extends` *ClassType*
(156)	*SwitchBlock*	::=	`{` *SwitchBlockGroups*$^?$ *SwitchLabels*$^?$ `}`

(157) *SwitchBlockGroup* ::= *SwitchLabels BlockStmts*

(158) *SwitchBlockGroups* ::= *SwitchBlockGroup*
| *SwitchBlockGroups SwitchBlockGroup*

(159) *SwitchLabel* ::= `case` *ConstantExp* `:`
| `default` `:`

(160) *SwitchLabels* ::= *SwitchLabel*
| *SwitchLabels SwitchLabel*

(161) *SwitchStmt* ::= `switch` `(` *Exp* `)` *SwitchBlock*

(162) *ThrowStmt* ::= `throw` *Exp* `;`

(163) *TryStmt* ::= `try` *Block Catches*
| `try` *Block Catches*$^?$ *Finally*

(164) *Type* ::= *FunctionType*
| *ConstrainedType*
| *Void*

(165) *TypeArgs* ::= `[` *TypeArgumentList* `]`

(166) *TypeArgumentList* ::= *Type*
| *TypeArgumentList* `,` *Type*

(167) *TypeDecln* ::= *ClassDecln*
| *StructDecln*
| *InterfaceDecln*
| *TypeDefDecln*
| `;`

(168) *TypeDeclns* ::= *TypeDecln*
| *TypeDeclns TypeDecln*

(169) *TypeDefDecln* ::= *Mods*$^?$ `type` *Id TypeParams*$^?$ *Guard*$^?$ = *Type* `;`
| *Mods*$^?$ `type` *Id TypeParams*$^?$ `(` *FormalList* `)` *Guard*$^?$ = *Type* `;`

(170) *TypeImportOnDemandDecln* ::= `import` *PackageOrTypeName* `.` `*` `;`

(171) *TypeName* ::= *Id*
| *TypeName* . *Id*

(172) *TypeParam* ::= *Id*

(173) *TypeParamIList* ::= *TypeParam*
| *TypeParamIList* , *TypeParam*
| *TypeParamIList* ,

(174) *TypeParamList* ::= *TypeParam*
| *TypeParamList* , *TypeParam*

(175) *TypeParams* ::= [*TypeParamList*]

(176) *TypeParamsI* ::= [*TypeParamIList*]

(177) *UnannotatedUnaryExp* ::= *PreIncrementExp*
| *PreDecrementExp*
| + *UnaryExpNotPlusMinus*
| - *UnaryExpNotPlusMinus*
| *UnaryExpNotPlusMinus*

(178) *UnaryExp* ::= *UnannotatedUnaryExp*
| *Annotations UnannotatedUnaryExp*

(179) *UnaryExpNotPlusMinus* ::= *PostfixExp*
| ~ *UnaryExp*
| ! *UnaryExp*
| ^ *UnaryExp*
| | *UnaryExp*
| & *UnaryExp*
| * *UnaryExp*
| / *UnaryExp*
| % *UnaryExp*

(180) *UserAsyncStmt* ::= `async` *TypeArgs*$^?$ *Arguments*$^?$ *ClockedClause*$^?$ *ClosureBodyBlock*

(181) *UserAtEachStmt* ::= `ateach` *TypeArgs*$^?$ (*FormalList* `in` *ArgumentList*$^?$) *ClosureBodyBlock*
| `ateach` *TypeArgs*$^?$ (*ArgumentList*$^?$) *ClosureBodyBlock*

(182) *UserAtomicStmt* ::= `atomic` *TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*

(183) *UserAtStmt* ::= `at` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock*

(184) *UserBreakStmt* ::= `break` *TypeArgs*$^?$ *Exp*$^?$ `;`

(185) *UserCatchClause* ::= `catch` `(` *FormalList*$^?$ `)` *ClosureBodyBlock*

(186) *UserCatches* ::= *UserCatchClause*
| *UserCatches UserCatchClause*

(187) *UserContinueStmt* ::= `continue` *TypeArgs*$^?$ *Exp*$^?$ `;`

(188) *UserDoStmt* ::= `do` *TypeArgs*$^?$ *ClosureBodyBlock* `while` `(` *ArgumentList*$^?$ `)` `;`

(189) *UserEnhancedForStmt* ::= `for` *TypeArgs*$^?$ `(` *FormalList* `in` *ArgumentList*$^?$ `)` *ClosureBodyBlock*
| `for` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock*

(190) *UserFinallyBlock* ::= `finally` *ClosureBodyBlock*

(191) *UserFinishStmt* ::= `finish` *TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock*

(192) *UserIfThenStmt* ::= `if` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock*
| `if` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock* `else` *ClosureBodyBlock*

(193) *UserStmt* ::= *UserEnhancedForStmt*
| *UserIfThenStmt*
| *UserTryStmt*
| *UserThrowStmt*
| *UserAsyncStmt*
| *UserAtomicStmt*
| *UserWhenStmt*
| *UserFinishStmt*
| *UserAtStmt*
| *UserContinueStmt*
| *UserBreakStmt*
| *UserAtEachStmt*
| *UserWhileStmt*
| *UserDoStmt*

(194)	*UserStmtPrefix*	::=	*FullyQualifiedName* `.`
		|	*Primary* `.`
		|	`super .`
		|	*ClassName* `. super .`
(195)	*UserThrowStmt*	::=	`throw` *TypeArgs*$^?$ *Exp*$^?$ `;`
(196)	*UserTryStmt*	::=	`try` *TypeArgs*$^?$ *Arguments*$^?$ *ClosureBodyBlock* *UserCatches*$^?$ *UserFinallyBlock*$^?$
(197)	*UserWhenStmt*	::=	`when` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock*
(198)	*UserWhileStmt*	::=	`while` *TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)` *ClosureBodyBlock*
(199)	*VarDeclWType*	::=	*Id HasResultType* = *VariableInitializer*
		|	`[` *IdList* `]` *HasResultType* = *VariableInitializer*
		|	*Id* `[` *IdList* `]` *HasResultType* = *VariableInitializer*
(200)	*VarDeclsWType*	::=	*VarDeclWType*
		|	*VarDeclsWType* `,` *VarDeclWType*
(201)	*VarKeyword*	::=	`val`
		|	`var`
(202)	*VariableDeclr*	::=	*Id HasResultType*$^?$ = *VariableInitializer*
		|	`[` *IdList* `]` *HasResultType*$^?$ = *VariableInitializer*
		|	*Id* `[` *IdList* `]` *HasResultType*$^?$ = *VariableInitializer*
(203)	*VariableDeclrs*	::=	*VariableDeclr*
		|	*VariableDeclrs* `,` *VariableDeclr*
(204)	*VariableInitializer*	::=	*Exp*
(205)	*Void*	::=	`void`
(206)	*WhenStmt*	::=	`when (` *Exp* `)` *Stmt*
(207)	*WhileStmt*	::=	`while (` *Exp* `)` *Stmt*

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] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[7] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[8] Kiyokuni Kawachiya, Mikio Takeuchi, Salikh Zakirov, and Tamiya Onodera. Distributed garbage collection for managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 5:1–5:11, New York, NY, USA, 2012. ACM.

[9] 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.

[10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2 edition, January 2011.

[11] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. In *Proceedings of VLDB Conference*, VLDB '12, 2012.

[12] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[13] Mikio Takeuchi, David Cunningham, David Grove, and Vijay Saraswat. Java interoperability in managed X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 39–46, New York, NY, USA, 2013. ACM.

[14] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. Compiling X10 to Java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 3:1–3:10, New York, NY, USA, 2011. ACM.

[15] Mikio Takeuchi, Salikh Zakirov, Kiyokuni Kawachiya, and Tamiya Onodera. Fast method dispatch and effective use of primitives for reified generics in managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 4:1–4:7, New York, NY, USA, 2012. ACM.

[16] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

A Deprecations

X10 version 2.4 has a few relics of previous versions, code that is being used by libraries but is not intended for general programming. They should be ignored.

These are:

- `acc` variables.
- The `offers` clause for use with collecting finish.
- The grammar allows covariant and contravariant type parameters, marked by `+` and `-`:

  ```
  class Variant[X, +Y, -Z] {}
  ```

  X10 does not support these in any other way.
- The syntax allows for a few Java-isms, such as `c.class` and `super.class`, which are not used.

B Change Log

B.1 Changes from X10 v2.5

To resolve parsing ambiguities, the property keyword is now mandatory to introduce property methods.

The X10 language now supports trailing closures. A trailing closure is a closure block that is written and after the parentheses of a function call. This closure block is passed as a closure without argument to the function.

B.2 Changes from X10 v2.4

Although there were no backwards incompatible language changes between X10 v2.4 and X10 v2.5, a few core class library APIs did have backwards incompatible changes. These changes were driven by our experience with Resilient and Elastic X10 and are designed to better support X10 computations over a dynamically varying number of Places. In summary,

1. Static constants such as `PlaceGroup.WORLD`, and `Place.MAX\_PLACES` were removed. They are replaced by `Place.places()` and `Place.numPlaces()` which return values that represent the current view on the dynamically changeable set of Places available to the computation.

2. The removal of iteration functionality (`next` and `prev`) from `Place`. This functionality is now provided only through `PlaceGroup`.

3. The addition of `PlaceTopology` to provide a more flexible set of APIs describing the topological relationships of Places.

In addition, the `put` and `get` methods of `x10.util.Map` were changed to no longer wrap their return value in an instance of `x10.util.Box`. This improves the common case efficiency of map usage, but does require that the Values stored in Map satisfy the haszero constraint.

B.3 Changes from X10 v2.3

X10 v2.4 is not backwards compatible with X10 v2.3. The motivation for making backwards incompatible language changes with this release of X10 is to significantly improve the ability of the X10 programmer to exploit the expanded memory capabilities of modern computer systems. In particular, X10 v2.4 includes an extensive redesign of arrays and a change of the default type of unqualified integral literals (*e.g.*`2`) from `Int` to `Long`. Taken together these two changes enable natural exploitation of large memories via 64-bit addressing and `Long`-based indexing of arrays and similar data structures.

B.3.1 Integral Literals

The default type of unqualified integral literals was changed from `Int` to `Long`.

The qualifying suffix `n` and `un` are used to indicate `Int` and `UInt` literals respectively. The suffix `u` is now interpreted as indicating a `ULong` literal.

B.3.2 Arrays

An extensive redesign of the X10 abstractions for arrays is the major new feature of the X10 v2.4 release. Although this redesign only involved very minor changes to the actual X10 language specification, the core class libraries did change significantly. As mentioned above, the driving motivation for the change was a long-contemplated strategic decision to shift to from `Int`-based (32-bit) to `Long`-based (64-bit) indexing for all X10 arrays. This change enables X10 to better utilize the rapidly expanding memory capacity and 64-bit address space found on modern machines. For consistency, the `id` field of `x10.lang.Place` and the `size` and indexing-related APIs of the `x10.util` collection hierarchy were also changed from `Int` to `Long`.

Once this inherently backwards-incompatible decision was made, the X10 team decided to do a larger rethinking of all of X10's array implementations to introduce a new time and space optimal implementation of zero-based, dense, rectangular multi-dimensional arrays. This new implementation, in the `x10.array` package, is intended to provide the best possible performance for the common-case it supports. The previous, more general array implementation is still available, but has been relocated to a new package `x10.array.regionarray`. In addition, the `x10.lang.Rail` class was re-introduced as a separate class in its own right and provides the intrinsic indexed storage abstraction on which both array packages are built. The intent is that the combination of `Rail`, `x10.array` and `x10.regionarray` provide a spectrum of array abstractions that capture common usage patterns and enable appropriate trade-offs between performance and flexibility.

In more detail the major array-related changes made in the X10 v2.4 release are

1. The class `x10.lang.Rail` was introduced. It provides an efficient one-dimensional, zero-based, densely indexed array implementation. `Rail` will provide the best performance and is the preferred implementation of this basic abstraction.
2. The array literal syntax `[1,2,3]` is now defined to create a `Rail` instead of an `Array`.
3. The main method signature is changed from `Array[String]` to `Rail[String]`.
4. `x10.util.IndexedMemoryChunk` has removed from the X10 standard library.
5. To enable usage of classes from both `x10.array` and `x10.regionarray`, the package `x10.array` is no longer auto-imported by the X10 compiler.
6. Most classes in the `x10.array` package in the X10 v2.3 release were relocated to the `x10.regionarray` package in v2.4. A few classes like `Point` and `PlaceGroup` were moved to the `x10.lang` package instead.
7. `Point`, `Region`, `Dist`, etc. were all updated to support long-based indexing by consistently changing indexing related fields and methods from `Int` to `Long`.

B.3.3 Other Changes from X10 v2.3

1. The custom serialization protocol was changed to operate in terms of new user-level classes `x10.io.Serializer` and `x10.io.Deserializer`. The `serialize` method of the xcdx10.io.CustomSerialization interface now takes a `Serializer` as an argument. The custom deserialization constructor for a class takes a `Deserializer`. The `x10.io.SerialData` class used by the X10 v2.3 custom serialization protocol has been removed from the class library.
2. A constraint was added to `PlaceLocalHandle` that types used to instantiate a `PlaceLocalHandle` must satisfy both the `isref` and `haszero` constraints.
3. The `x10.util.Team` API was revised by (a) removing the endpoint argument from all API calls and (b) to operate on `Rail` and xcd‘Long‘ where appropriate.

B.4 Changes from X10 v2.2

1. In previous versions of X10 static fields were eagerly initialized in `Place 0` and the resulting values were serialized to all other places before execution of the user main function was started. Starting with X10 v2.2.3, static fields are lazily initialized on a per-Place basis when the field is first read by an activity executing in a given Place.
2. The new syntax `T isref` for some type `T` will hold if `T` is represented by a pointer at runtime. This is similar to the type constraint `T haszero`. `T isref` is true for `T` that are function types, classes, and all values that have been cast to

interfaces (including boxed structs). `T isref` is used in the standard library, e.g. for the `GlobalRef[T]` and `PlaceLocalHandle[T]` APIs.

3. `x10.lang.Object is gone`, there is now no single class that is the root of the X10 class hierarchy.
   - If, for some reason, you were explicitly extending `Object`, don't do that anymore.
   - If you were doing `new Object()` to get a fresh value, use `new Empty()` instead.
   - If you were using `Object` as a supertype, use `Any` (the one true supertype).
   - If you were using the type constraint `T <: Object` to disallow structs, use `T isref` instead.
4. The exception hierarchy has changed, and checked exceptions have been reintroduced. The 'throws' annotation is required on methods, as in Java. It is not supported on closures, so checked exceptions cannot be thrown from a closure. The exception hierarchy has been chosen to exist in a 1:1 relationship with Java's. However, unlike Java, we prefer using unchecked exceptions wherever possible, and this is reflected in the naming of the X10 classes. The following classes are all in the `x10.lang` package.
   - `CheckedThrowable` (mapped to `java.lang.Throwable`)
   - `CheckedException extends CheckedThrowable` (mapped to `java.lang.Exception`)
   - `Exception extends CheckedException` (mapped to `java.lang.RuntimeException`)
   - `Error extends CheckedThrowable` (mapped to `java.lang.Error`)

   Anything under `CheckedThrowable` can be thrown using the `throw` statement. But anything that is not under `Exception` or `Error` can only be thrown if it is caught by an enclosing `try/catch`, or it is thrown from a method with an appropriate throws annotation, as in Java.

   `RuntimeException` is gone from X10. Use `Exception` instead.

   All the exceptions in the standard library are under `Exception`, except `AssertionError` and `OutOfMemoryException`, which are under `Error` (as in Java). This means all exceptions in the standard library remain unchecked.

B.5 Changes from X10 v2.1

1. Covariance and contravariance are gone.
2. Operator definitions are regularized. A number of new operator symbols are available.
3. The operator `in` is gone. `in` is now only a keyword.

4. Method functions and operator functions are gone.
5. `m..n` is now a type of struct called `IntRange`.
6. `for(i in m..n)` now works. The old forms, `for((i) in m..n)` and `for([i] in m..n)`, are no longer needed.
7. `(e as T)` now has type `T`. (It used to have an identity constraint conjoined in.)
8. `vars` can no longer be assigned in their place of origin. Use a `GlobalRef[Cell[T]]` instead. We'll have a new idiom for this in 2.3.
9. The `-STATIC_CALLS` command-line flag is now `-STATIC_CHECKS`.
10. Any string may be written in backquotes to make an identifier: `` `while` ``.
11. The `next` and `resume` keywords are gone; they have been replaced by static methods on `Clock`.
12. The typed array construction syntax `new Array[T][t1,t2]` is gone. Use `[t1 as T, t2]` (if just plain `[t1,t2]` doesn't work).

B.6 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.

B.6.1 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`.

B.6.2 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.

B.6.3 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.

B.6.4 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);
  }
  // WRONG: print("a=" + a);
}
print("a=" + a);
```

B.6.5 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)) {..}
```

B.6.6 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)`.

B.6.7 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

B.6.8 Removed Topics

The following are gone:

1. `foreach` is gone.
2. All `vars` 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.

B.6.9 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.

B.7 Changes from X10 v2.0

Some of these changes have been made obsolete in X10 2.2.

- `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.

B.8 Changes from X10 v1.7

The language has changed in the following ways. Some of these changes have been made obsolete in X10 2.2.

- **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.

C Options

C.1 Compiler Options: Common

The X10 compilers have many useful options.

C.1.1 Optimization: `-O` or `-optimize`

This flag causes the compiler to generate optimized code.

C.1.2 Debugging: `-DEBUG=boolean`

This flag, if true, causes the compiler to generate debugging information. It is false by default.

C.1.3 Call Style: `-STATIC_CHECKS, -VERBOSE_CHECKS`

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 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_CHECKS`.

You may also turn on strict static checking, with the `-STATIC_CHECKS` flag. With static checking, calls that cannot be proved correct statically will be marked as errors.

C.1.4 Help: `-help` and `-- -help`

These options cause the compiler to print a list of all command-line options.

C.1.5 Source Path: `-sourcepath` *`path`*

This option tells the compiler where to look for X10 source code.

C.1.6 Output Directory: `-d` *`directory`*

This option tells the compiler to produce its output files in the specified directory.

C.1.7 Executable File: `-o` *`path`*

This option tells the compiler what path to use for the executable file.

C.2 Compiler Option: C++

The C++ compilation command `x10c++` has the following option as well.

C.2.1 Runtime: `-x10rt` *`impl`*

This option tells which runtime implementation to use. The choices are `sockets`, `standalone`, `pami`, `mpi`, and `bgas_bgp`.

C.3 Compiler Option: Java

The Java compilation command `x10c` has the following option as well.

C.3.1 Class Path: `-classpath` *`path`*

This option is used in conjunction with the Java interoperability feature to tell the compiler where to look for Java .class files that may be used by the X10 code being compiled.