The initializer expression may directly or indirectly read other static fields in the program. If there is a cycle in the field initialization dependency graph for a set of static fields, then any activities accessing those fields may deadlock, which in turn may result in the program deadlocking.[3].

If an exception is raised during the evaluation of a static field's initializer expression, then the field is deemed uninitializable in that Place and any subsequent attempt to access the static field's value by another activity in the Place will also result in an exception being raised.[4]. Failure to initialize a field in one Place does not impact the initialization status of the field in other Places.

8.6.1 Compatability with Prior Versions of X10

Previous versions of X10 eagerly initialized all static fields in the program at Place 0 and serialized the resulting values to all other Places before beginning execution of the user main function. It is possible to simulate these serialization semantics for specific static fields under the lazy per-Place initialization semantics by using the idiom below:

```
// Pre X10 2.3 code
// expr evaluated once in Place 0 and resulting value
// serialized to all other places
public static val x:T = expr;

// X10 2.3 code when T haszero is false
private static val x_holder:Cell[T] =
    (here == Place.FIRST_PLACE) ? new Cell[T](expr): null;
public static val x:T = at (Place.FIRST_PLACE) x_holder();

// simpler X10 2.3 code when T haszero is true
private static val x_holder:T =
    (here == Place.FIRST_PLACE) ? expr : Zero.get[T]();
public static val x:T = at (Place.FIRST_PLACE) x_holder;
```

A slightly more complex variant of the above idiom in which the initializer expression for the public field conditionally does the `at` only when not executed at `Place.FIRST_PLACE` can be used to obtain exactly the same serialization behavior as the pre X10 v2.3 semantics. When necessary, eager initialization for specific static fields can be simulated by reading the static fields in `main` before executing the rest of the program.

[3] The current X10 runtime does not dynamically detect this situation. Future versions of X10 may be able to detect this and convert such a deadlock into the throwing of an `ExceptionInInitializer` exception.

[4] The implementation will make a best effort attempt to present stack trace information about the original cause of the exception in all subsequent raised exceptions

8.7 User-Defined Operators

MethodDecln ::= *MethMods* `def` *Id* *TypeParams*? *Formals* *Guard*? *Throws*? *(20.117)*
HasResultType? *MethodBody*
| *BinOpDecln*
| *PrefixOpDecln*
| *ApplyOpDecln*
| *SetOpDecln*
| *ConversionOpDecln*
| *KeywordOpDecln*

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

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

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

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

```
            );
        }
    }
```

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

Most X10 operators can be given definitions.[5] (However, `&&` and `||` are only short-circuiting for `Boolean` expressions; user-defined versions of these operators have no special execution behavior.)

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

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

Several of these operators have no standard meaning on any library type, and are included purely for programmer convenience.

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

Example:

```
static class Trace(n:Long){
  public static operator !(f:Trace)
      = new Trace(10 * f.n + 1);
  public operator -this = new Trace (10 * this.n + 2);
}
```

[5] Indeed, even for the standard types, these operators are defined in the library. Not even as basic an operation as integer addition is built into the language. Conversely, if you define a full-featured numeric type, it will have most of the privileges that the standard ones enjoy. The missing priveleges are (1) literals; (2) * won't track ranks, as it does for `Region`s; (3) `&&` and `||` won't short-circuit, as they do for `Boolean`s, (4) the built-in notion of equality `a==b` will only coincide with the programmible notion `a.equals(b)`, as they do for most library types, if coded that way; and (5) it is impossible to define an operation like `String.+` which converts both its left and right arguments from any type. For example, a `Polar` type might have many representations for the origin, as radius 0 and any angle; these will be `equals()`, but will not be ==; whereas for the standard `Complex` type, the two equalities are the same.

```
static class Brace extends Trace{
  def this(n:Long) { super(n); }
  public operator -this = new Brace (10 * this.n + 3);
  static def example() {
     val t = new Trace(1);
     assert (!t).n == 11;
     assert (-t).n == 12 && (-t instanceof Trace);
     val b = new Brace(1);
     assert (!b).n == 11;
     assert (-b).n == 13 && (-b instanceof Brace);
  }
}
```

8.7.1 Binary Operators

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

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

Or, it may be defined as as an instance operator by one of the forms:

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

Example:

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

```
class Poly {
  public val coeff : Rail[Long];
  public def this(coeff: Rail[Long]) {
    this.coeff = coeff;}
  public def degree() = coeff.size-1;
  public def  a(i:Long)
    = (i<0 || i>this.degree()) ? 0L : coeff(i);
  public operator this + (p:Poly) =  new Poly(
     new Rail[Long](
        Math.max(this.coeff.size, p.coeff.size),
        (i:Long) => this.a(i) + p.a(i)
     ));
  // ...
```

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

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

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

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

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

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

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

8.7.2 Unary Operators

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

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

or as instance operators by:

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

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

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

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

8.7.3 Type Conversions

Explicit type conversions, `e as A`, can be defined as operators on class `A`, or on the container type of `e`. These must be static operators.

To define an operator in `class A` (or `struct A`) converting values of type `B` into type `A`, use the syntax:

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

The `?` indicates the containing type `A`. The guard clause `{c}` may be omitted.

Example:

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

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

There is little difference between an explicit conversion `e as T` and a method call `e.asT()`. The explicit conversion does say undeniably what the result type will be. However, as described in §11.22.3, sometimes the built-in meaning of `as` as a cast overrides the user-defined explicit conversion.

Explicit casts are most suitable for cases which resemble the use of explicit casts among the arithmetic types, where, for example, `1.0 as Int` is a way to turn a floating-point number into the corresponding integer. While there is nothing in X10 which requires it, `e as T` has the connotation that it gives a good approximation of `e` in type `T`, just as `1` is a good (indeed, perfect) approximation of `1.0` in type `Int`.

8.7.4 Implicit Type Coercions

An implicit type conversion from `U` to `T` may be specified in container `T`. The syntax for it is:

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

Implicit coercions are used automatically by the compiler on method calls (§8.12) and assignments (§11.7). Implicit coercions may be used when a value of type `T` appears in a context expecting a value of type `U`. If `T <: U`, no implicit coercion is needed; *e.g.*, a method `m` expecting an `Long` argument may be called as `m(3)`, with an argument of type `Long{self==3}`, which is a subtype of `m`'s argument type `Long`. However, if it is not the case that `T <: U`, but there is an implicit coercion from `T` to `U` defined in container `U`, then this implicit coercion will be applied.

Example: *We can define an implicit coercion from* `Long` *to* `Poly`*, and avoid having to define the sum of an integer and a polynomial as many special cases. In the following example, we only define + on two polynomials. The calculation* `1+x` *coerces* `1` *to a polynomial and uses polynomial addition to add it to* `x`*.*

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

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

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

8.7.5 Assignment and Application Operators

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

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

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

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

Example:

The `Oddvec` *class of somewhat peculiar vectors illustrates this.*

`a()` *returns a string representation of the oddvec, which ordinarily would 0be done by* `toString()` *instead.* `a(i)` *sensibly picks out one of the three coordinates of* `a`*.* `a()=b` *sets all the coordinates of* `a` *to* `b`*.* `a(i)=b` *assigns to one of the coordinates.* `a(i,j)=b` *assigns different values to* `a(i)` *and* `a(j)`*.*

```
class Oddvec {
  var v : Rail[Long] = new Rail[Long](3);
```

```
public operator this () =
    "(" + v(0) + "," + v(1) + "," + v(2) + ")";
public operator this () = (newval: Long) {
  for(p in v.range) v(p) = newval;
}
public operator this(i:Long) = v(i);
public operator this(i:Long, j:Long) = [v(i),v(j)];
public operator this(i:Long) = (newval:Long)
    {v(i) = newval;}
public operator this(i:Long, j:Long) = (newval:Long)
    { v(i) = newval; v(j) = newval+1;}
public def example() {
  this(1) = 6;   assert this(1) == 6;
  this(1) += 7;  assert this(1) == 13;
}
```

8.8 User-Defined Control Structures

KeywordOpDecln	::=	*MethMods* `operator` *keywordOp TypeParams*$^?$ *Formals Guard*$^?$ *Throws*$^?$ *HasResultType*$^?$ *MethodBody*	*(20.105)*
KeywordOp	::=	`for`	*(20.104)*
	|	`if`	
	|	`try`	
	|	`throw`	
	|	`async`	
	|	`atomic`	
	|	`when`	
	|	`finish`	
	|	`at`	
	|	`continue`	
	|	`break`	
	|	`ateach`	
	|	`while`	
	|	`do`	

Similarly to user-defined operators (Section 8.7), it is possible to redefine the behavior of some control structures. For example, suppose that we want to define a `if` statement that randomly chooses which branch to execute. In a class `RandomIf`, we define a method named `if` (introduced with the keyword `operator`) that implements this behavior:

```
class RandomIf {
    val random = new Random();
    public operator if(then: ()=>void, else_: ()=>void) {
```

```
            if (random.nextBoolean()) {
                then();
            } else {
                else_();
            }
        }
    }
```

Then, we can call this method using the syntax of the `if` statement by prefixing the `if` keyword by an object that implements this method:

```
val random = new RandomIf();
random.if () {
    Console.OUT.println("true");
} else {
    Console.OUT.println("false");
}
```

The blocks that represent the `then` and the `else` branches of the `if` are automatically turned into closures and are given as argument to the `RandomIf.if` method.

To distinguish the use of a user-defined control structure from the use of a built-in one, the first keyword of the control structure must be prefixed with the object that redefines its behavior. The scoping and dispatching rules of user-defined control structures are exactly the same as the rules for methods.

User-defined control structures can also be called as standard methods using the keyword `operator` as prefix (as for user-defined operators). For example, the previous code is equivalent to:

```
val random = new RandomIf();
random.operator if (() => { Console.OUT.println("true"); },
                    () => { Console.OUT.println("false"); });
```

The correspondence between the two invocation syntaxes is formally specified in Figure 8.1 for all the control structures we support. It uses the following conventions: o is either a class path or an object; $\overline{T}$ is a list of types; $\overline{x\texttt{:}t}$ is a list of variable declaration with their types; $\overline{e}$ is a list of expressions; b is a closure body: a list of statements between curly braces that can optionally end with an expression (a return value); $(\,)^?$ is an optional group.

Let's consider the rule for `if`:

$$o\texttt{.if[}\overline{T}\texttt{]}^?\texttt{(}\overline{e}\texttt{)}\ b_1\ (\texttt{else}\ b_2)^? \quad \equiv$$
$$\quad o\texttt{.operator if[}\overline{T}\texttt{]}^?\texttt{(}\overline{e}\texttt{, ()=>}\ b_1\texttt{(, ()=>}\ b_2)^?\texttt{);}$$

Compared to the builtin `if` control structure, the user-defined one accepts type arguments and replaces one condition expression with a list of expressions, possibly empty. The branches of the user-defined `if` statement are lifted to no-arg closures and passed to the user-defined `if` method as arguments. The `else` branch is optional.

o`.if[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)` b_1 `(else` b_2`)`$^?$ $\equiv$
o`.operator if[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, ()=>` b_1`(, ()=>` b_2`)`$^?$`);`
o`.for[`$\overline{T}$`]`$^?$`((`$\overline{x\texttt{:}t}$ `in)`$^?$ $\overline{e}$`)` b $\equiv$
o`.operator for[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, (`$\overline{x\texttt{:}t}^?$`) =>` b`);`
o`.try[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)`$^?$ b_1 `catch (`$\overline{x\texttt{:}t}$`)` b_2 `(finally` b_3`)`$^?$ $\equiv$
o`.operator try[`$\overline{T}$`]`$^?$`((`$\overline{e}$`,)`$^?$ `()=>` b_1`, (`$\overline{x\texttt{:}t}$`) =>` b_2`(, ()=>` b_3`)`$^?$`);`
o`.throw[`$\overline{T}$`]`$^?$ $e^?$`;` $\equiv$ o`.operator throw[`$\overline{T}$`]`$^?$`(`$e^?$`);`
o`.async[`$\overline{T}$`]`$^?$`(`$\overline{e_1}$`)`$^?$ `(clocked (`$\overline{e_2}$`))`$^?$ b $\equiv$
o`.operator async[`$\overline{T}$`]`$^?$`((`$\overline{e_1}$`,)`$^?$ `(`$\overline{e_2}$`,)`$^?$ `()=>` b`);`
o`.atomic[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)`$^?$ b $\equiv$ o`.operator atomic[`$\overline{T}$`]`$^?$`((`$\overline{e}$`,)`$^?$ `()=>` b`);`
o`.when[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)` b $\equiv$ o`.operator when[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, ()=>` b`);`
o`.finish[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)`$^?$ b $\equiv$ o`.operator finish[`$\overline{T}$`]`$^?$`((`$\overline{e}$`,)`$^?$ `()=>` b`);`
o`.at[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)` b $\equiv$ o`.operator at[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, ()=>` b`);`
o`.continue[`$\overline{T}$`]`$^?$ $e^?$`;` $\equiv$ o`.operator continue[`$\overline{T}$`]`$^?$`(`$e^?$`);`
o`.break[`$\overline{T}$`]`$^?$ $e^?$`;` $\equiv$ o`.operator break[`$\overline{T}$`]`$^?$`(`$e^?$`);`
o`.ateach[`$\overline{T}$`]`$^?$`((`$\overline{x\texttt{:}t}$ `in)`$^?$ $\overline{e}$`)` b $\equiv$
o`.operator ateach[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, (`$\overline{x\texttt{:}t}^?$`) =>` b`);`
o`.while[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)` b $\equiv$ o`.operator while[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, ()=>` b`);`
o`.do[`$\overline{T}$`]`$^?$ b `while (`$\overline{e}$`);` $\equiv$ o`.operator do[`$\overline{T}$`]`$^?$`(()=>` b`,` $\overline{e}$`);`

Figure 8.1: Correspondence between control structure and method call notations.

The correspondence is purely syntactic. In other words, the control structure syntax is simply rewritten into the regular method invocation syntax with no consideration of types or method lookup.

8.8.1 User-Defined `for`

o`.for[`$\overline{T}$`]`$^?$`((`$\overline{x\texttt{:}t}$ `in)`$^?$ $\overline{e}$`)` b $\equiv$
o`.operator for[`$\overline{T}$`]`$^?$`(`$\overline{e}$`, (`$\overline{x\texttt{:}t}^?$`) =>` b`);`

A `for` loop over a collection may be defined in a container `A` as:

```
operator for[T](c: Iterable[T], body: (T)=>void) = ...
```

The use of such a user-defined `for` loop would have the following form:

```
A.for (x: T in c) { ... }
```

and would correspond to the following method call:

```
A.operator for (c, (x: Long) => { ... });
```

The body of the `for` is automatically translated into a closure that takes the iteration variable as parameter. Since there is no type inference for closure parameters, the type of the iteration variable must be given explicitly.

The second argument of a `for` method can be a closure without argument:

```
operator for[T](c: Iterable[T], body: ()=>void) = ...
```

In this case, the method is called using the syntax of a `for` loop without iteration variable:

```
A.for (c) { ... }
```

Example: *A naive implementation of a parallel loop can be:*

```
class Parallel {

  public static operator for[T](c: Iterable[T], body: (T)=>void) {
    finish {
      for(x in c) {
        async { body(x); }
      }
    }
  }

  public static def main(Rail[String]) {
    val cpt = new Cell[Long](0);
    Parallel.for(i:Long in 1..10) {
      atomic { cpt() = cpt() + i; }
    }
    Console.OUT.println(cpt());
  }
}
```

Example: *We can also use the user-defined* `for` *loops to define iterations over a two dimensional space. Let us define a loop that creates an activity for each element of the first dimension.*

```
class Parallel2 {
  public static operator for (space: DenseIterationSpace_2,
                              body: (i:Long, j:Long)=>void) {
    finish {
      for (i in space.min0 .. space.max0) {
        async for (j in space.min1 .. space.max1) {
          body(i, j);
        }
      }
    }
  }
}
```

and it can be used as follows:

```
Parallel2.for (i:Long, j:Long in 1..10 * 1..10) { ... }
```

The list of variables before the `in` *keyword becomes the parameters of the closure whose body is the body of the loop.*

8.8.2 User-Defined `if`

$$o\texttt{.if[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}\ b_1\ (\texttt{else}\ b_2)^{?} \equiv o\texttt{.operator if[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, ()=>}\ b_1\texttt{(, ()=>}\ b_2)^{?}\texttt{);}$$

When we use a user-defined `if` statement, the condition is evaluated before calling the `if` method, but the then and else branches are implicitly lifted to closures without argument.

Note that the condition of a user-defined `if` statement can take an arbitrary number of arguments. This is why we were able to define the `Random.if` that does not take a condition.

8.8.3 User-Defined `try`

$$o\texttt{.try[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}^{?}\ b_1\ \texttt{catch (}\overline{x\texttt{:}t}\texttt{)}\ b_2\ (\texttt{finally}\ b_3)^{?} \equiv o\texttt{.operator try[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e}\texttt{,)}^{?}\ \texttt{()=>}\ b_1\texttt{, (}\overline{x\texttt{:}t}\texttt{) =>}\ b_2\texttt{(, ()=>}\ b_3)^{?}\texttt{);}$$

When we use a user-defined `try` statement, the body of the `try` is lifted to a closure without argument and handler is lifted to a closure that has the parameter of the `catch` as parameter. The `finally` block is also lifted to a closure without argument.

Example: *The user-defined* `try` *construct can be used to provide a control structure that automatically removes the nesting of* `MultipleExceptions`*:*

```
class Flatten {

  public static operator try(body:()=>void,
                             handler:(MultipleExceptions)=>void) {
    try { body(); }
    catch (me: MultipleExceptions) {
      val exns = new GrowableRail[CheckedThrowable]();
      flatten(me, exns);
      handler (new MultipleExceptions(exns));
    }
  }

  private static def flatten(me:MultipleExceptions,
                             acc:GrowableRail[CheckedThrowable]) {
    for (e in me.exceptions) {
```

```
        if (e instanceof MultipleExceptions) {
          flatten(e as MultipleExceptions, acc);
        } else {
          acc.add(e);
        }
      }
    }
  }
```

Used in the following example, the `MultipleExceptions me` *contains the exceptions* `Exception("Exn 1")`*,* `Exception("Exn 2")`*, and* `Exception("Exn 3")` *instead of the exception* `Exception("Exn 1")` *and another* `MultipleExceptions`*.*

```
  public static def main(Rail[String]) {
    Flatten.try {
      finish {
        async { throw new Exception("Exn 1"); }
        async finish {
          async { throw new Exception("Exn 2"); }
          async { throw new Exception("Exn 3"); }
        }
      }
    } catch (me: MultipleExceptions) {
      Console.OUT.println(me.exceptions);
    }
  }
```

8.8.4 User-Defined `throw`

$$o\texttt{.throw[}\overline{T}\texttt{]}^{?}\ e^{?}\texttt{;} \quad \equiv \quad o\texttt{.operator throw[}\overline{T}\texttt{]}^{?}\texttt{(}e^{?}\texttt{);}$$

The argument of a user-defined `throw` is evaluated before calling the `throw` method.

8.8.5 User-Defined `async`

$$o\texttt{.async[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e_1}\texttt{)}^{?}\ \texttt{(clocked (}\overline{e_2}\texttt{))}^{?}\ b \quad \equiv$$
$$o\texttt{.operator async[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e_1}\texttt{,)}^{?}\ \texttt{(}\overline{e_2}\texttt{,)}^{?}\ \texttt{()=>}\ b\texttt{);}$$

The body of a user-defined `async` is lifted to a closure without argument. The clock arguments are evaluated before the call to the `async` method.

Example: *An* `async` *that does not execute in the scope in which it is written. The task is created in the scope where the object that defines the* `async` *method is instantiated.*

```
class Escape {
  private var task: ()=>void = null;
  private var stop: Boolean = false;

  public def this() {
    async {
      while (!stop) {
        val t: () => void;
        when (task != null || stop) {
          t = task;
          task = null;
        }
        if (t != null) {
          async { t(); }
        }
      }
    }
  }

  public operator async (body: () => void) {
    when (task == null) {
      task = body;
    }
  }

  public def stop() {
    atomic { stop = true; }
  }
}
```

In the following example, the message `"OK"` *is printed even if the created task never terminates because the task is executed outside of the scope of the* `finish`*.*

```
public static def main(Rail[String]) {
  val toplevel = new Escape();
  finish {
    toplevel.async { when (false){} }
  }
  Console.OUT.println("OK");
}
```

8.8.6 User-Defined `atomic`

o`.atomic[`$\overline{T}$`]`$^?$`(`$\overline{e}$`)`$^?$ b $\equiv$ o`.operator atomic[`$\overline{T}$`]`$^?$`(`$(\overline{e},)^?$ `()=>` b`);`

The body of a user-defined atomic statement is lifted to a closure without argument.

8.8.7 User-Defined `when`

$$o\texttt{.when[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{) } b \;\equiv\; o\texttt{.operator when[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, ()=> }b\texttt{);}$$

The arguments of a user-defined `when` statements are evaluated before the call of the `when` method and the body is lifted to a closure without argument. It means that if the argument of a user-defined `when` is of type `Boolean`, the condition is evaluated once and cannot be changed. To be able to update the condition, it can be an object with mutable field as in the following example or a closure.

Example: *We can provide a* `when` *statement whose execution can be canceled while it is waiting:*

```
class CancelableWhen {
  private var stop : Boolean = false;

  public operator when(condition:Cell[Boolean], body:()=>void) {
    when (condition() || stop) {
      if (!stop) { body(); }
    }
  }

  public def cancel() {
    atomic { stop = true; }
  }
}
```

The following example will not print the message `"KO"` *but will terminate even if the condition* `b` *of the* `when` *remains false:*

```
public static def main(Rail[String]) {
  val c = new CancelableWhen();
  val b = new Cell[Boolean](false);
  finish {
    async {
      c.when(b) { Console.OUT.println("KO"); }
    }
    c.cancel();
  }
}
```

8.8.8 User-Defined `finish`

$$o\texttt{.finish[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}^{?}\, b \;\equiv\; o\texttt{.operator finish[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e}\texttt{,)}^{?}\texttt{ ()=> }b\texttt{);}$$

The body of a user-defined `finish` is lifted to a closure.

Example: *We define a* `finish` *that provide the ability to some parallel task to wait for its termination:*

```
class SignalingFinish {
  private var terminated : Boolean = false;
  public operator finish(body: ()=>void) {
    finish {
      body();
    }
    atomic { terminated = true; }
  }
  public def join() {
    when (terminated) {}
  }
}
```

The following example will always print the message `"before"` *before the message* `"after"`.

```
public static def main(Rail[String]) {
  val t = new SignalingFinish();
  async {
    t.join();
    Console.OUT.println("after");
  }
  t.finish {
    Console.OUT.println("before");
  }
}
```

8.8.9 User-Defined `at`

$$o.\texttt{at}[\overline{T}]^{?}(\overline{e})\ b \quad \equiv \quad o.\texttt{operator at}[\overline{T}]^{?}(\overline{e}, \texttt{()=>}\, b);$$

The arguments of the user-defined `at` statement are evaluated before the call of the `at` method and the body of the statement is lifted to a closure without argument.

Example: *We define a class* `Ring` *implementing an* `at` *statement without argument. Each call to this user-defined* `at` *statement moves the activity to the next place in the place group given when the object is instantiated.*

```
class Ring {
  val places: PlaceGroup;

  public def this (places: PlaceGroup) {
    this.places = places;
  }

  public operator at(body: ()=>void) {
```

```
    at(places.next(here)) { body(); }
  }
}

public static def main(Rail[String]) {
  val r = new Ring(Place.places());
  r.at() {
    Console.OUT.println("Hello from "+here+"!");
    r.at() {
      Console.OUT.println("Hello from "+here+"!");
    }
  }
}
```

8.8.10 User-Defined `ateach`

$$o\texttt{.ateach[}\overline{T}\texttt{]}^{?}\texttt{(}(\overline{x\texttt{:}t}\ \texttt{in})^{?}\ \overline{e}\texttt{)}\ b \equiv o\texttt{.operator ateach[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, (}\overline{x\texttt{:}t}^{?}\texttt{) => }b\texttt{);}$$

The arguments of the user-defined `ateach` statement are evaluated before the call of the `ateach` method and the body of the statement is lifted to a closure without argument.

Example: *An* `ateach` *control structure that has the same behavior as the built-in* `ateach`*, except that the activities are executed in sequence instead of being executed in parallel.*

```
class Sequential {
  public static operator ateach (d: Dist, body:(Point)=>void) {
    for (place in d.places()) {
      at(place) {
        for (p in d|here) { body(p); }
      }
    }
  }
}
```

8.8.11 User-Defined `while` and `do`

$$o\texttt{.while[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}\ b \equiv o\texttt{.operator while[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, ()=> }b\texttt{);}$$

$$o\texttt{.do[}\overline{T}\texttt{]}^{?}\ b\ \texttt{while (}\overline{e}\texttt{);} \equiv o\texttt{.operator do[}\overline{T}\texttt{]}^{?}\texttt{(()=> }b\texttt{, }\overline{e}\texttt{);}$$

The arguments of the user-defined `while` (resp. `do`) are evaluated before the call of the `while` (resp. `do`) method and the body of the loop is lifted to a closure without argument. Note that compared to usual loop, the condition is evaluated once before the call of the method that implements the behavior of the loop.

Example: *A loop that iterates during at least a given number of milliseconds:*

```
class Timeout {
  public static operator while(ms: Long, body: ()=>void) {
    val deadline = System.currentTimeMillis() + ms;
    while (System.currentTimeMillis() < deadline) {
      body();
    }
  }
}
```

Here, we increment a counter during a period of at least 10 milliseconds:

```
public static def main(Rail[String]) {
  val cpt = new Cell[Long](0);
  Timeout.while(10) {
    atomic { cpt() = cpt() + 1; }
  }
  Console.OUT.println(cpt());
}
```

8.8.12 User-Defined `continue`

o`.continue[`$\overline{T}$`]`$^?$ $e^?$`;` $\equiv$ o`.operator continue[`$\overline{T}$`]`$^?$`(`$e^?$`);`

The argument of a user-defined `continue` is evaluated before calling the corresponding method.

Example: *The following code provides a parallel* `for` *loop with a* `continue` *statement that allows skipping an iteration.*

```
class Par {
  private static class Continue extends Exception {}

  public static operator continue () {
    throw new Continue();
  }

  public static operator for[T](c: Iterable[T], body:(T)=>void) {
    finish {
      for(x in c) async {
          try {
            body(x);
          } catch (Continue) {}
        }
    }
  }
}
```

The following example skips every iteration where the loop index is even.

```
public static def main(Rail[String]) {
  val cpt = new Cell[Long](0);
  Par.for(i:Long in 1..10) {
    if (i%2 == 0) { Par.continue; }
    atomic { cpt() = cpt() + 1; }
  }
  Console.OUT.println(cpt());
}
```

8.8.13 User-Defined `break`

$$o\texttt{.break[}\overline{T}\texttt{]}^{?}\ e^{?}\texttt{;} \quad \equiv \quad o\texttt{.operator break[}\overline{T}\texttt{]}^{?}\texttt{(}e^{?}\texttt{);}$$

The argument of a user-defined `break` is evaluated before calling the corresponding method.

Example: *To break out of a user-defined loop, it is necessary to also define the* `break` *statement:*

```
class Infinite {
  private static class Break extends Exception {}

  public static operator break () {
    throw new Break();
  }

  public static operator while (body:()=>void) {
    try {
      while(true) {
        body();
      }
    } catch (Break) {}
  }

  public static def main(Rail[String]) {
    Infinite.while() {
      Infinite.break;
    }
    Console.OUT.println("OK");
  }
}
```

8.9 Class Guards and Invariants

Classes (and structs and interfaces) may specify a *class guard*, a constraint which must hold on all values of the class. In the following example, a `Line` is defined by two distinct `Pt`s[6]

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

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

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

With every container or interface `T` we associate a *type invariant* $inv(\texttt{T})$, which describes the guarantees on the properties of values of type `T`.

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

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

The type invariant associated with any interface or struct `I` that extends interfaces $\texttt{I}_1$, ..., $\texttt{I}_k$ and defines properties $\texttt{x}_1\texttt{: P}_1$, ..., $\texttt{x}_n\texttt{: P}_n$ and specifies a guard `c` is given by:

$inv(\texttt{I}_1)$ && ... && $inv(\texttt{I}_k)$ &&
$\texttt{self.x}_1$ instanceof $\texttt{P}_1$ && ... && $\texttt{self.x}_n$ instanceof $\texttt{P}_n$
&& c

Similarly the type invariant associated with any class `C` that implements interfaces $\texttt{I}_1$, ..., $\texttt{I}_k$, extends class `D` and defines properties $\texttt{x}_1\texttt{: P}_1$, ..., $\texttt{x}_n\texttt{: P}_n$ and specifies a guard `c` is given by the same thing with the invariant of the superclass `D` conjoined:

$inv(\texttt{I}_1)$ && ... && $inv(\texttt{I}_k)$
&& $\texttt{self.x}_1$ instanceof $\texttt{P}_1$ && ... && $\texttt{self.x}_n$ instanceof $\texttt{P}_n$
&& c
&& inv(D)

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

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

8.9.1 Invariants for `implements` and `extends` clauses

Consider a class definition

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

These two rules must be satisfied:

- The type invariant inv`(C)` of `C` must entail $\mathtt{c}_i$`[this/self]` for each i in $\{1, \ldots, k\}$
- The return type `c` of each constructor in a class `C` must entail the invariant inv `(C)`.

8.9.2 Timing of Invariant Checks

The invariants for a container are checked immediately after the `property` statement in the container's constructor. This is the earliest that the invariant could possibly be checked. Recall that an invariant can mention the properties of the container (which are set, forever, at that point in the code), but cannot mention the `val` or `var` fields (which might not be set at that point), or `this` (which might not have been fully initialized).

If X10 can prove that the invariant always holds given the `property` statement and other known information, it may omit the actual check.

8.9.3 Invariants and constructor definitions

A constructor for a class `C` is guaranteed to return an object of the class on successful termination. This object must satisfy inv`(C)`, the class invariant associated with `C` (§8.9). However, often the objects returned by a constructor may satisfy *stronger* properties than the class invariant. X10's dependent type system permits these extra properties to be asserted with the constructor in the form of a constrained type (the "return type" of the constructor):

CtorDecln ::= *Mods*$^?$ `def this` *TypeParams*$^?$ *Formals Guard*$^?$ *HasResultType*$^?$ *CtorBody* (20.54)

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

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

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

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

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

Let `C` be a class with properties $\texttt{p}_1\texttt{: P}_1\texttt{, ..., p}_n\texttt{: P}_n$, and invariant `c` extending the constrained type `D{d}` (where `D` is the name of a class).

For every constructor in `C` the compiler checks that the call to super invokes a constructor for `D` whose return type is strong enough to entail `d`. Specifically, if the call to super is of the form $\texttt{super(e}_1\texttt{, ..., e}_k\texttt{)}$ and the static type of each expression $\texttt{e}_i$ is $\texttt{S}_i$, and the invocation is statically resolved to a constructor $\texttt{def this(x}_1\texttt{: T}_1\texttt{, ..., x}_k\texttt{: T}_k\texttt{)\{c\}: D\{d}_1\texttt{\}}$ then it must be the case that

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

where `a` is a constant that does not appear in $\texttt{x}_1\texttt{: S}_1 \wedge \texttt{ ... } \wedge \texttt{ x}_k\texttt{: S}_k$.

The compiler checks that every constructor for `C` ensures that the properties $\texttt{p}_1\texttt{, ..., p}_n$ are initialized with values which satisfy $inv(\texttt{T})$, and its own return type `c'` as follows. In each constructor, the compiler checks that the static types $\texttt{T}_i$ of the expressions $\texttt{e}_i$ assigned to $\texttt{p}_i$ are such that the following is true:

$\texttt{p}_1\texttt{: T}_1\texttt{, ..., p}_n\texttt{: T}_n\ \texttt{entails}\ inv(\texttt{T}) \wedge \texttt{c'}$

(Note that for the assignment of $\texttt{e}_i$ to $\texttt{p}_i$ to be type-correct it must be the case that $\texttt{p}_i\texttt{: T}_i \wedge \texttt{p}_i\texttt{: P}_i$.)

The compiler must check that every invocation $\texttt{C(e}_1\texttt{, ..., e}_n\texttt{)}$ to a constructor is type correct: each argument $\texttt{e}_i$ must have a static type that is a subtype of the declared type $\texttt{T}_i$ for the ith argument of the constructor, and the conjunction of static types of the argument must entail the constraint in the parameter list of the constructor.

8.10 Generic Classes

Classes, like other units, can be generic. They can be parameterized by types. The parameter types are used just like ordinary types inside the body of the generic class – with a few exceptions.

Example: *A* `Colorized[T]` *holds a thing of type* `T`*, and a string which is intended to represent its color. Any type can be used for* `T`*; the* `example` *method shows* `Long` *and* `Boolean`*. The* `thing()` *method retrieves the thing; note that its return type is the generic type variable* `T`*. X10 is aware that* `colLong.thing()` *is an* `Long` *and* `colTrue.thing()` *is a* `Boolean`*, and uses those typings in* `example`*.*

```
class Colorized[T] {
  private var thing:T;
  private var color:String;
  def this(thing:T, color:String) {
     this.thing = thing;
     this.color = color;
  }
  public def thing():T = thing;
  public def color():String = color;
  public static def example() {
    val colLong  : Colorized[Long]
                 = new Colorized[Long](3, "green");
    assert colLong.thing() == 3
                 && colLong.color().equals("green");
    val colTrue : Colorized[Boolean]
                 = new Colorized[Boolean](true, "blue");
    assert colTrue.thing()
                 && colTrue.color().equals("blue");
  }
}
```

8.10.1 Use of Generics

An unconstrained type variable `X` can be instantiated by any type. All the operations of `Any` are available on a variable of type `X`. Additionally, variables of type `X` may be used with `==`, `!=`, in `instanceof`, and casts.

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

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

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

8.11 Object Initialization

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

1. Use of a field before the field has been initialized.
2. A program reading two different values from a `val` field of a container.
3. `this` escaping from a constructor, which can cause problems as noted below.

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

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

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

```
class Wrong {
  val shouldBe8 : Long;
  static Cell[Wrong] wrongCell = new Cell[Wrong]();
  static def doItWrong() {
     finish {
       async { new Wrong(); } // (A)
       assert( wrongCell().shouldBe8 == 8); // (B)
     }
```

```
    }
    def this() {
       wrongCell.set(this); // (C) - ILLEGAL
       this.shouldBe8 = 8; // (D)
    }
  }
```

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

1. `doItWrong()` is called.
2. `(A)` is started. Space for a new `Wrong` object is allocated. Its `shouldBe8` field, not yet initialized, contains some garbage value.
3. `(C)` is executed, as part of the process of constructing a new `Wrong` object. The new, uninitialized object is stored in `wrongCell`.
4. Now, the initialization activity is paused, and execution of the main activity proceeds from `(B)`.
5. The value in `wrongCell` is retrieved, and is `shouldBe8` field is read. This field contains garbage, and the assertion fails.
6. Now let the initialization activity proceed with `(D)`, initializing `shouldBe8` — too late.

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

```
// This code violates this chapter's constraints
// and thus will not compile in X10.
class Example {
  val a: Long;
  def this() {
    at(here.next()) {
      // Recall that 'this' is a copy of 'this' outside 'at'.
      Console.OUT.println("this.a = " + this.a);
    }
    this.a = 1;
  }
}
```

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

8.11.1 Constructors and Non-Escaping Methods

In general, constructors must not be allowed to call methods with `this` as an argument or receiver. Such calls could leak references to `this`, either directly from a call to `cell.set(this)`, or indirectly because `toString` leaks `this`, and the concatenation ‘”Escaper = ”+this‘ calls `toString`.[7]

```
// This code violates this chapter's constraints
// and thus will not compile in X10.
class Escaper {
  static val Cell[Escaper] cell = new Cell[Escaper]();
  def toString() {
    cell.set(this);
    return "Evil!";
  }
  def this() {
    cell.set(this);
    x10.io.Console.OUT.println("Escaper = " + this);
  }
}
```

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

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

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

Non-Escaping Methods

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

A method can, however, be safe to call from constructors without being marked `@NonEscaping`. We call such methods *implicitly non-escaping*. Implicitly non-escaping methods need

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

to obey the same constraints on `this`, `super`, and variable usage as `@NonEscaping` methods. An implicitly non-escaping method *could* be marked as `@NonEscaping`; the compiler, in effect, infers the annotation. In addition, all non-escaping methods must be `private` or `final` or members of a `final` class; this corresponds to the hereditary nature of `@NonEscaping` (by forbidding inheritance of implicitly non-escaping methods).

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

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

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

```
import x10.compiler.*;

final class C2 {
  protected val a:Long; protected val b:Long; protected val c:Long;
  protected var x:Long; protected var y:Long; protected var z:Long;
  def this() {
    a = 1;
    this.aplomb();
    b = 2;
    this.boric();
    c = 3;
    this.cajoled();
  }
  @NonEscaping def aplomb() {
    x = a;
    // this.boric(); // not allowed; boric reads b.
    // z = b; // not allowed -- only 'a' can be read here
  }
  @NonEscaping final def boric() {
    y = b;
    this.aplomb(); // allowed;
       // a is definitely set before boric is called
    // z = c; // not allowed; c is not definitely written
  }
  @NonEscaping private def cajoled() {
    z = c;
```

```
  }
}
```

NoThisAccess Methods

A method may be annotated `@NoThisAccess`. `@NoThisAccess` methods may be called from constructors, and they may be overridden in subclasses. However, they may not refer to `this` in any way – in particular, they cannot refer to fields of `this`, nor to `super`.

Example:

The class `IDed` *has an* `Float`*-valued* `id` *field. The method* `count()` *is used to initialize the* `id`*. For* `IDed` *objects, the* `id` *is the count of* `IDed`*s created with the same parity of its* `kind`*. Note that* `count()` *does not refer to* `this`*, though it does refer to a* `static` *field* `counts`*.*

The subclass `SubIDed` *has* `id`*s that depend on* `kind%3` *as well as the parity of* `kind`*. It overrides the* `count()` *method. The body of* `count()` *still cannot refer to* `this`*. Nor can it refer to* `super` *(which is* `self` *under another name). This precludes the use of a* `super` *call. This is why we have separated the body of* `count` *out as the static method* `kind2count` *– without that, we would have had to duplicate its body, as we could not call* `super.count(kind)` *in a* `NoThisAccess` *method, as is shown by the* `ERROR` *line* `(A)`*.*

Note that `NoThisAccess` *is in* `x10.compiler` *and must be imported, and that the overriding method* `SubIDed.count` *must be declared* `@NoThisAccess` *as well as the overridden method. Line* `(B)` *is not allowed because* `code` *is a field of* `this`*, and field accesses are forbidden. Line* `(C)` *references* `this` *directly, which, of course, is forbidden by* `@NoThisAccess`*.*

```
import x10.compiler.*;
class UseNoThisAccess {
  static class IDed {
    protected static val counts = [0 as Long,0];
    protected var code : Long;
    val id: Float;
    public def this(kind:Long) {
      code = kind;
      this.id = this.count(kind);
    }
    protected static def kind2count(kind:Long) = ++counts(kind % 2);
    @NoThisAccess def count(kind:Long) : Float = kind2count(kind);
  }
  static class SubIDed extends IDed {
    protected static val subcounts = [0 as Long, 0, 0];
    public static val all = new x10.util.ArrayList[SubIDed]();
    public def this(kind:Long) {
```

```
        super(kind);
      }
      @NoThisAccess
      def count(kind:Long) : Float {
        val subcount <: Long = ++subcounts(kind % 3);
        val supercount <: Float = kind2count(kind);
        //ERROR: val badSuperCount = super.count(kind); //(A)
        //ERROR: code = kind;                           //(B)
        //ERROR: all.add(this);                         //(C)
        return  supercount + 1.0f / subcount;
      }
    }
  }
```

8.11.2 Fine Structure of Constructors

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

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

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

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

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

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

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

```
class Overlord(x:Long) {
  def this(x:Long) { property(x); }
}//Overlord
class Overdone(y:Long) extends Overlord  {
```

```
  val a : Long;
  val b =  y * 9000;
  def this(r:Long) {
    super(r);                       // (1)
    x10.io.Console.OUT.println(r); // (2)
    val rp1 = r+1;
    property(rp1);                  // (2)
    // field initializations here   // (3)
    a = r + 2 + b;                  // (4)
  }
  def this() {
    this(10);                       // (1), (2), (3)
    val x = a + b;                  // (4)
  }
}//Overdone
```

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

Initialization and Inner Classses

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

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

Initialization and Closures

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

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

8.11.3 Definite Initialization in Constructors

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

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

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

The fundamental principles are:

1. `val` fields must be assigned precisely once in each constructor on every possible execution path.
2. `var` fields of defaultless type must be assigned at least once on every possible execution path, but may be assigned more than once.
3. No variable may be read before it is guaranteed to have been assigned.
4. Initialization may be by field initialization expressions (`val x : Long = y+z`), or by uninitialized fields `val x : Long;` plus an initializing assignment `x = y+z`. Recall that field initialization expressions are performed after the `property` statement, in segment 3 in the terminology of §8.11.2.

8.11.4 Summary of Restrictions on Classes and Constructors

The following table tells whether a given feature is (yes), is not (no) or is with some conditions (note) allowed in a given context. For example, a property method is allowed

with the type of another property, as long as it only mentions the preceding properties.
The first column of the table gives examples, by line of the following code body.

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

Details:

- (1) Top-level `self` only.
- (2) The type of the i^{th} property may only mention properties 1 through i.
- (3) Super-interfaces follow the same rules as supertypes.
- (4) The same rules apply to types and initializers.

The example indices refer to the following code:

```
class Example (
   prop : Long,
   proq : Long{prop != proq},                    // (A)
   pror : Long
   )
   {prop != 0}                                  // (B)
   extends Supertype[Long{self != prop}]         // (C)
   implements SuperInterface[Long{self != prop}] // (D)
{
   property def propmeth() = (prop == pror);    // (E)
   static staticField
      : Cell[Long{self != 0}]                    // (F)
      = new Cell[Long{self != 0}](1);            // (G)
   var instanceField
      : Long {self != prop}                      // (H)
      = (prop + 1) as Long{self != prop};        // (I)
   def this(
      a : Long{a != 0},
```

```
        b : Long{b != a}                            // (J)
        )
        {a != b}                                    // (K)
        : Example{self.prop == a && self.proq==b} // (L)
    {
        super();                                    // (M)
        property(a,b,a);                            // (N)
        // fields initialized here
        instanceField = b as Long{self != prop};   // (O)
    }

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

8.12 Method Resolution

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

Example:

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

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

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

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

```
def example() {
   assert m(100) == 1 : "Long";
   assert m(new C()) == 0 : "C";
   // An Ace is a Surface, unambiguous best choice
   assert m(new Ace()) == 3 : "Ace";
   // ERROR: m(new Face());

   // The match must be exact.
   // ERROR: assert m(here) == 3 : "Place";

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

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

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

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

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

The match must be exact. There is an implicit coercion from `Place` *to* `Ace`*, and* `Ace` *implements* `Surface`*, so the code*

```
val ace : Ace = here;
assert m(ace) == 3;
```

works, by using the `Surface` *form of* `m`*. But doing it in one step requires a deeper search than X10 performs*[8]*, and is not allowed.*

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

The basic concept of method resolution is:

1. List all the methods that could possibly be used, inferring generic types but not performing implicit coercions.
2. If one possible method is more specific than all the others, that one is the desired method.

[8] In general this search is unbounded, so X10 can't perform it.

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

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

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

After method resolution is done, there is a validation phase that checks the legality of the call, based on the `STATIC_CHECKS` compiler flag. With `STATIC_CHECKS`, the method's constraints must be satisfied; that is, they must be entailed (§4.5.2) by the information in force at the point of the call. With `DYNAMIC_CHECKS`, if the constraint is not entailed at that point, a dynamic check is inserted to make sure that it is true at runtime.

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

8.12.1 Space of Methods

X10 allows some constructs, particularly `operator`s, to be defined in a number of ways, and invoked in a number of ways. This section specifies which forms of definition could correspond to a given definiendum.

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

- An instance method on `a`, of the form `def m(B)`.
- A static method on `a`'s class, of the form `static def m(B)`.

The meaning of an invocation of the form `m(b)`, with any number of arguments, depends slightly on its context. Inside of a constraint, it might mean `self.m(b)`. Outside of a constraint, there is no `self` defined, so it can't mean that. The first of these that applies will be chosen.

1. Invoke a method on `this`, *viz.* `this.m(b)`. Inside a constraint, it may also invoke a property method on `self`, *viz.*. `self.m(b)`. It is an error if both `this.m(b)` and `self.m(b)` are possible.

2. Invoke a function named `m` in a local or field.

3. Construct a structure named `m`.

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

- A static method on `A`, of the form `static def m(B)`.

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

- A constructor on `A`, of the form `def this(B)`.

A unary operator `⋆ a` may be defined as:

- An instance operator on `A`, defined as `operator ⋆ this()`.
- A static operator on `A`, defined as `operator ⋆(a:A)`.

A binary operator `a ⋆ b` may be defined as:

- An instance operator on `A`, defined as `operator this ⋆(b:B)`; or
- A right-hand operator on `B`, defined as `operator (a:A) ⋆ this`; or
- A static operator on `A`, defined as `operator (a:A) ⋆ (b:B)`, ; or
- A static operator on `B`, if `A` and `B` are different classes, defined as `operator (a:A) ⋆ (b:B)`

If none of those resolve to a method, then either operand may be implicitly coerced to the other. If one of the following two situations obtains, it will be done; if both, the expression causes a static error.

- An implicit coercion from `A` to `B`, and an operator `B ⋆ B` can be used, by coercing `a` to be of type `B`, and then using `B`'s `⋆`.
- An implicit coercion from `B` to `A`, and an operator `A ⋆ A` can be used, coercing `b` to be of type `A`, and then using `A`'s `⋆`.

An application `a(b)`, for any number of arguments, can come from a number of things.

- an application operator on `a`, defined as `operator this(b:B)`;
- If `a` is an identifier, `a(b)` can also be a method invocation equivalent to `this.a(b)`, which invokes `a` as either an instance or static method on `this`
- If `a` is a qualified identifier, `a(b)` can also be an invocation of a struct constructor.

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

- `operator this(b:B)=(c:C) {...}`

An implicit coercion, in which a value `a:A` is used in a context which requires a value of some other non-subtype `B`, can only come from implicit coercion operation defined on `B`:

- an implicit coercion in `B`: `static operator (a:A):B`;

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

1. An `as` operator in `B`: `static operator (a:A) as ?`;
2. or, failing that, an implicit coercion in `B`: `static operator (a:A):B`.

8.12.2 Possible Methods

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

Generics introduce several subtleties, especially with the inference of generic types. For the purposes of method resolution, all that matters about a method, constructor, or operator `M` — we use the word "method" to include all three choices for this section — is its signature, plus which method it is. So, a typical `M` might look like `def m[`$\mathtt{G}_1$`,..., `$\mathtt{G}_g$`](`$\mathtt{x}_1$`:`$\mathtt{T}_1$`,..., `$\mathtt{x}_f$`:`$\mathtt{T}_f$`){c} =...`. The code body `...` is irrelevant for the purpose of whether a given method call means `M` or not, so we ignore it for this section.

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

1. The method name `m`;
2. The generic type parameters of the method `m`, $\mathtt{G}_1, \ldots, \mathtt{G}_g$. If there are no generic type parameters, $g = 0$.
3. The types $\mathtt{x}_1:\mathtt{T}_1, \ldots, \mathtt{x}_f:\mathtt{T}_f$ of the formal parameters. If there are no formal parameters, $f = 0$. In the case of an instance method, the receiver will be the first formal parameter.[9]
4. A *unique identifier* `id`, sufficient to tell the compiler which method body is intended. A file name and position in that file would suffice. The details of the identifier are not relevant.

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

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

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

All that matters about an invocation `I` is:

1. The method name $\mathtt{m}'$;
2. The generic type parameters $\mathtt{G}'_1, \ldots, \mathtt{G}'_g$. If there are no generic type parameters, $g = 0$.
3. The names and types $\mathtt{x}_1\mathtt{:T}'_1, \ldots, \mathtt{x}_f\mathtt{:T}'_f$ of the actual parameters. If there are no actual parameters, $f = 0$. In the case of an instance method, the receiver is the first actual parameter.

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

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

1. Eliminate from `context` those methods which are not *acceptable*; *viz.*, those whose name, type parameters, and formal parameters do not suitably match `invo`. In more detail:
   - The method name `m` must simply equal the invocation name $\mathtt{m}'$;
   - X10 infers type parameters, by an algorithm given in §4.12.3.
   - The method's type parameters are bound to the invocation's for the remainder of the acceptability test.
   - The actual parameter types must be subtypes of the formal parameter types, or be coercible to such subtypes. Parameter i is a subtype if $\mathtt{T}'_i$ `<:` $\mathtt{T}_i$. It is implicitly coercible to a subtype if either it is a subtype, or if there is an implicit coercion operator defined from $\mathtt{T}'_i$ to some type `U`, and `U <:` $\mathtt{T}_i$. . If coercions are used to resolve the method, they will be called on the arguments before the method is invoked.
2. Eliminate from `context` those methods which are not *available*; *viz.*, those which cannot be called due to visibility constraints, such as methods from other classes marked `private`. The remaining methods are both acceptable and available; they might be the one that is intended.

3. If the method invocation is a `super` invocation appearing in class `Cl`, methods of `Cl` and its subclasses are considered unavailable as well.

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

    - If there is a unique such `ms`, then `resolve(invo,context)` returns the `id` of `ms`.
    - If there is not a unique such `ms`, then `resolve` reports an error.

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

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

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

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

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

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

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

8.12.3 Field Resolution

An identifier `p` can refer to a number of things. The rules are somewhat different inside and outside of a constraint.

Outside of a constraint, the compiler chooses the first one from the following list which applies:

1. A local variable named `p`.

2. A field of `this`, *viz.* `this.p`.

3. A nullary property method, `this.p()`

4. A member type named `p`.

5. A package named `p`.

Inside of a constraint, the rules are slightly different, because `self` is available, and packages cannot be used per se.

1. A local variable named `p`.
2. A property of `this` or of `self`, *viz.* `this.p` or `self.p`. If both are available, report an error.
3. A nullary property method, `this.p()`
4. A member type named `p`.

8.12.4 Other Disambiguations

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

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

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

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

In the case where a syntactic form `E.m(`F_1`, ..., `F_n`)` could be resolved as either a method call, or the application of a field `E.m` to some arguments, it will be treated as a method call. The application of `E.m` to some arguments can be specified by adding parentheses: `(E.m)(`F_1`, ..., `F_n`)`.

Example:

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

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

8.13 Static Nested Classes

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

Nested interfaces and static structs are permitted as well.

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

8.14 Inner Classes

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

Example: *This feature is particularly useful when an instance of the inner class makes no sense without reference to an instance of the outer, and is closely tied to it. For example, consider a range class, describing a span of integers m to n, and an*

iterator over the range. The iterator might as well have access to the range object, and there is little point to discussing iterators-over-ranges without discussing ranges as well. In the following example, the inner class `RangeIter` *iterates over the enclosing* `Range`.

It has its own private cursor field `n`*, telling where it is in the iteration; different iterations over the same* `Range` *can exist, and will each have their own cursor. It is perhaps unwise to use the name* `n` *for a field of the inner class, since it is also a field of the outer class, but it is legal. (It can happen by accident as well –* e.g.*, if a programmer were to add a field* `n` *to a superclass of the outer class, the inner class would still work.) It does not even interfere with the inner class's ability to refer to the outer class's* `n` *field: the cursor initialization refers to the* `Range`*'s lower bound through a fully qualified name* `Range.this.n`*. The initialization of its* `n` *field refers to the outer class's* `m` *field, which is not shadowed and can be referred to directly, as* `m`.

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

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

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

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

The members of inner classes must be instance members. They cannot be static members. Classes, interfaces, static methods, static fields, and typedefs are not allowed as members of inner classes. The same restriction applies to local classes (§8.15).

Consider an inner class `IC1` of some outer class `OC1`, being extended by another class `IC2`. However, since an `IC1` only exists as a dependent of an `OC1`, each `IC2` must be associated with an `OC1` — or a subtype thereof — as well. So, `IC2` must be an inner class of either `OC1` or some subclass `OC2 <: OC1`.

Example: *For example, one often extends an inner class when one extends its outer class:*

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

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

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

8.14.1 Constructors and Inner Classes

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

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

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

```
class OC {
  class IC {}
```

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

8.15 Local Classes

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

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

```
class Outer {
  val a = 1;
  def m() {
    val a = -2;
    val b = 2;
    class Local {
      val a = 3;
      def m() = 100*Outer.this.a + 10*b + a;
    }
    val l : Local = new Local();
    assert l.m() == 123;
  }//end of m()
}
```

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

The members of inner classes must be instance members. They cannot be static members. Classes, interfaces, static methods, static fields, and typedefs are not allowed as members of local classes. The same restriction applies to inner classes (§8.14).

8.16 Anonymous Classes

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

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

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

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

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

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

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

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

- Anonymous classes don't have constructors. Since they don't have names, there's no way a constructor could get called in the ordinary way. Instead, the `new C(...)` expression must match a constructor of the parent class `C`, which will be called to initialize the newly-created object of the anonymous class.
- The `public`, `private`, and `protected` modifiers don't make sense for anonymous classes: Anonymous classes, being anonymous, cannot be referenced at all, so references to them can't be public, private, or protected.
- Anonymous classes cannot be `abstract`. Since they only exist in combination with a constructor call, they must be constructable. The parent class of the anonymous class may be abstract, or may be an interface; in this case, the anonymous class must provide all the methods that the parent demands.
- Anonymous classes cannot have explicit `extends` or `implements` clauses; there's no place in the syntax for them. They have a single parent and that is that.

9 Structs

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

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

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

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

9.1 Struct declaration

StructDecln	::=	*Mods*$^?$ `struct` *Id TypeParamsI*$^?$ *Properties*$^?$ *Guard*$^?$ *Interfaces*$^?$ *ClassBody*	*(20.154)*
TypeParamsI	::=	`[` *TypeParamIList* `]`	*(20.177)*
Properties	::=	`(` *PropList* `)`	*(20.142)*
Guard	::=	*DepParams*	*(20.83)*
Interfaces	::=	`implements` *InterfaceTypeList*	*(20.103)*
ClassBody	::=	`{` *ClassMemberDeclns*$^?$ `}`	*(20.33)*

All fields of a struct must be `val`.

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

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

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

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

When a struct and a method have the same name (often in violation of the X10 capitalization convention), `new` *may be used to resolve to the struct's constructor.*

```
struct Ambig(x:Long) {
  static def Ambig(x:Long) = "ambiguity please";
  static def example() {
    val useMethod      = Ambig(1);
    val useConstructor = new Ambig(2);
  }
}
```

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

Example:

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

9.2 Boxing of structs

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

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

Example: *The rail* aa *in the following example is a* Rail[Any]. *It initially holds two objects. Then, its elements are replaced by two structs, both of which are auto-boxed. Note that no fussing is required to put an integer into a* Rail[Any]. *However, a rail of structs, such as* ah, *holds* unboxed *structs and does not incur boxing overhead.*

```
struct Horse(x:Long){
  static def example(){
    val aa : Rail[Any] = ["a String" as Any, "another one"];
    aa(0) = Horse(8);
    aa(1) = 13;
    val ah : Rail[Horse] = [Horse(7), Horse(13)];
  }
}
```

9.3 Optional Implementation of Any methods

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

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

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

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

9.4 Primitive Types

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

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

9.4.1 Signed and Unsigned Integers

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

9.5 Example structs

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

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

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

```
  static def example() {
     val qr = divmod(22un, 7un);
     assert qr.first == 3un && qr.second == 1un;
  }
}
```

9.6 Nested Structs

Static nested structs may be defined, essentially as static nested classes except for making them structs (§8.13). Inner structs may be defined, essentially as inner classes except making them structs (§8.14). **Limitation:** Nested structs must be currently be declared static.

9.7 Default Values of Structs

If all fields of a struct have default values, then the struct has a default value, *viz.*, the struct whose fields are all set to their default values. If some field does not have a default value, neither does the struct.

Example:

In the following code, the `Example` *struct has a default value whose* `i` *field is* `0`*. If an* `Example` *is ever constructed by the constructor, its* `i` *field will be* `1`*. This program does a slightly subtle dance to get ahold of a default* `Example`*, by having an instance* `var` *(which, unlike most kinds of variables, does not need to get initialized before use (though that exemption only applies if its type has a default value)). As the* `assert` *confirms, the default* `Example` *does indeed have an* `i` *field of* `0`*.*

```
class StructDefault {
  static struct Example {
    val i : Long;
    def this() { i = 1; }
  }
  var ex : Example;
  static def example() {
     val ex = (new StructDefault()).ex;
     assert ex.i == 0;
  }
```

9.8 Converting Between Classes And Structs

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

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

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

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

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

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

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

3. Objects can be set to `null`. Structs cannot.

4. The rules for default values are quite different. The default value of an object type (if it exists) is `null`, which behaves quite differently from an ordinary object of that type; *e.g.*, you cannot call methods on `null`, whereas you can on an ordinary object. The default value for a struct type (if it exists) is a struct like any other of its type, and you can call methods on it as for any other.

10 Functions

10.1 Overview

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

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

Limitation: X10 functions cannot have type arguments or constraints.

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

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

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

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

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

```
  val a1 : T1 = arg1();
  val a2 : T2 = arg2();
  result = f(a1,a2);
}
```

and

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

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

Function types may be used in `implements` clauses of class definitions. Suitable operator definitions must be supplied, with `public operator this(x1:T1, ..., xn:Tn)` declarations. Instances of such classes may be used as functions of the given type. Indeed, an object may behave like any (fixed) number of functions, since the class it is an instance of may implement any (fixed) number of function types. *e.g.* Instances of the `Funny` class behave like two functions: a constant function on Booleans, and a linear function on pairs of `Long`s.

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

10.2 Function Application

The basic operation on functions is function application. (Since, *e.g.*, array lookup has the same type as function application, these rules are used for array lookup as well, and

so on.)

A function with type `(`x_1`:`T_1`, ..., `x_n`:`T_n`){c} => T` can be applied to a sequence of expressions `e`$_1$`, ..., e`$_n$ if:

- `e`$_1$ is of type `T`$_1$`[e`$_1$`/x`$_1$`]`,
- ...,
- `e`$_n$ is of type `T`$_n$`[e`$_1$`/x`$_1$`, ..., e`$_n$`/x`$_n$`]`,
- X10 can prove that `c[e`$_1$`/x`$_1$`, ..., e`$_n$`/x`$_n$`]` holds.

In this case, if the application terminates normally, it returns a value of type `T[y1/x`$_1$`, ..., yn/x`$_n$`]` where `y1,...,yn` may be thought of as new variables defined as if by:

```
val y1=e1;
...
val yn=en;
```

Example: *Consider*

```
f : (a:Long{a!=0}, b:Long{b!=a}){b!=0} => Long{self != a}
```

Then the call `f(3,4)` *is allowed, because:*

- `3` *is of type* `Long{a!=0}` *with* `a` *replaced by* `3`*,* viz. `Long{3 != 0}`*;*
- `4` *is of type* `Long{b!=a}` *with* `a` *replaced by* `3` *and* `b` *replaced by* `4`*,* viz. `Long{3 != 4}`.
- *The guard* `b!=0`*, with* `a` *replaced by* `3` *and* `b` *replaced by* `4`*, is* `4!=0`*, which is true.*

So, `f(3,4)` *will return a value of type* `Long{self != a}` *with* `a` *replaced by* `3` *and* `b` *replaced by* `4`*, which is to say,* `Long{self!=3}`.

10.3 Function Literals

X10 provides first-class, typed functions, often called *closures*.

ClosureExp	::=	*Formals Guard*$^?$ *HasResultType*$^?$ `=>` *ClosureBody*	*(20.42)*
Formals	::=	`(` *FormalList*$^?$ `)`	*(20.80)*
Guard	::=	*DepParams*	*(20.83)*
HasResultType	::=	*ResultType*	*(20.86)*
	|	`<:` *Type*	
ClosureBody	::=	*Exp*	*(20.40)*
	|	*ClosureBodyBlock*	
ClosureBodyBlock	::=	*Annotations*$^?$ `{` *BlockStmts*$^?$ *LastExp* `}`	*(20.41)*
	|	*Annotations*$^?$ *Block*	

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

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

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

Example:

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

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

As with methods, a function with return type `void` cannot have a terminating expression. If the return type is omitted, it is inferred, as described in §4.12. It is a static error if the return type cannot be inferred. *E.g.*, `(Long)=>null` is not well-specified; X10 does not know which type of `null` is intended. But `(Long):Rail[Double] => null` is legal.

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

```
def find[T](f: (T) => Boolean, xs: List[T], no:T): T {
  for (x: T in xs)
    if (f(x)) return x;
  return no;
  }
```

The method may be invoked thus, to find a positive element of `xs`*, or return* `0` *if there is no positive element.*

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

10.3.1 Outer variable access

In a function `(`x_1`: `T_1`, ..., `x_n`: `T_n`){c} => { s }` the types T_i, the guard `c` and the body `s` may access the following variables from outer scopes:

- All fields of the enclosing object(s) and class(es);
- All type parameters;
- All `val` variables;

`var` variables cannot be accessed.

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

e.g. The following is legal. Note that `a` is not a local `var` variable. It is a field of `this`. A reference to `a` is simply short for `this.a`, which is a use of a `val` variable (`this`).

```
class Lambda {
   var a : Long = 0;
   val b = 0;
   def m(var c : Long, val d : Long) {
      var e : Long = 0;
      val f : Long = 0;
      val closure = (var i: Long, val j: Long) => {
          // c and e are not usable here
          a + b + d + f + i
               + j + this.a + Lambda.this.a
      };
      return closure;
   }
}
```

10.4 Functions as objects of type `Any`

Two functions `f` and `g` are equal if both were obtained by the same evaluation of a function literal.[1] Further, it is guaranteed that if two functions are equal then they refer to the same locations in the environment and represent the same code, so their executions in an identical situation are indistinguishable. (Specifically, if `f == g`, then `f(1)` can be substituted for `g(1)` and the result will be identical. However, there is no guarantee that `f(1)==g(1)` will evaluate to true, since there is no guarantee that `f(1)==f(1)` will evaluate to true either, as `f` might be a function which returns n on its n^{th} invocation. However, `f(1)==f(1)` and `f(1)==g(1)` are interchangeable.)

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

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

11 Expressions

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

11.1 Literals

Literals denote fixed values of built-in types. The syntax for literals is given in §3.5.

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

11.2 `this`

Primary ::= `this` *(20.138)*
| *ClassName* `.this`

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

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

Example: `Outer` *is a class containing* `Inner`. *Each instance of* `Inner` *has a reference* `Outer.this` *to the* `Outer` *involved in its creation.* `Inner` *has access to the fields of* `Outer.this`. *Note that* `Inner` *has its own* `three` *field, which is different from and not even the same type as* `Outer.this.three`.

```
class Outer {
  val three = 3;
  class Inner {
```

```
        val three = "THREE";
        def example()  {
          assert Outer.this.three == 3;
          assert three.equals("THREE");
          assert this.three.equals("THREE");
        }
      }
    }
```

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

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

11.3 Local variables

Id ::= `IDENTIFIER` *(20.90)*

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

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

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

11.4 Field access

FieldAccess ::= *Primary* `.` *Id* *(20.67)*
| `super` `.` *Id*
| *ClassName* `.` `super` `.` *Id*

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

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

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

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

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

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

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

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

```
class Uncle {
  public static val f = 1;
}
class Parent {
  public val f = 2;
}
class Ego extends Parent {
  public val f = 3;
  class Child extends Ego {
     public val f = 4;
     def example() {
        assert Uncle.f == 1;
        assert Ego.super.f == 2;
        assert super.f == 3;
        assert this.f == 4;
        assert f == 4;
     }
  }
}
```

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

11.5 Function Literals

Function literals are described in §10.

11.6 Calls

MethodInvo	::=	*MethodName TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`	*(20.118)*
	|	*Primary* `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`	
	|	`super` `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`	
	|	*ClassName* `.` `super` `.` *Id TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`	
	|	*Primary TypeArgs*$^?$ `(` *ArgumentList*$^?$ `)`	
ArgumentList	::=	*Exp*	*(20.8)*
	|	*ArgumentList* `,` *Exp*	
MethodName	::=	*Id*	*(20.120)*
	|	*FullyQualifiedName* `.` *Id*	

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

The syntax for method invocations is ambiguous. `ob.m()` could either be the invocation of a method named `m` on object `ob`, or the application of a function held in a field `ob.m`. If both are defined on the same class, X10 resolves `ob.m()` to the invocation of the method. If the application of a function in a field is desired, use an alternate syntax which makes the intent clear to X10, such as `(ob.m)()`.

Example:

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

However, adding a static method [mis]named `closure` *makes* `Callsome.closure()` *refer to the method, rather than the closure*

```
  static def closure () = 3;
  static def example() {
     assert Callsome.closure() == 3;
     assert (Callsome.closure)() == 1;
  }
```

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

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

Method selection rules are given in §8.12.

Guard satisfaction depends on the STATIC_CHECKS compiler flag. With the flag on, it is a static error if a method's *Guard* is not statically satisfied by the caller. With STATIC_CHECKS off, the guard will be checked at runtime if necessary.

Example: *In this example, a* DivideBy *object provides the service of dividing numbers by* denom *— so long as* denom *is not zero. X10's strictness of checking this is under control of the* STATIC_CHECKS *compiler option (§C.1.3).*

With STATIC_CHECKS *turned on, the* example *method will not compile. The call* this.div(100) *is not allowed; there is no guarantee that* denom != 0. *Casting* this *to a type whose constraint implies* denom != 0 *permits the method call.*

With STATIC_CHECKS *turned off, the call will compile. X10 will insert a dynamic check that the denominator is non-zero, and will fail at runtime if it is zero.*

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

11.6.1 super calls

The expression super.f(e1...en) may appear in an instance method definition. This causes the method invocation to be a super invocation, as described in §8.12.

Informally, suppose the invocation appears in class Cl, which extends class Sup. An invocation this.f() will call a nullary method named f that appears in class Cl itself, if there is one. An invocation super.f() will call the nullary f method in Sup or an ancestor thereof, but not one in Cl. Note that super.f() may be used to invoke an f method in Sup which has been overridden by one appearing in Cl.

Note that there's only one choice for which f is invoked by super.f() – *viz.* the lowest one in the class hierarchy above Cl. So, super.f() performs static dispatch, like a static method call. This is generally more efficient than a dynamic dispatch, like an instance method call.

11.7 Assignment

Assignment	::=	*LeftHandSide AsstOp AsstExp*	*(20.12)*
	|	*ExpName* (*ArgumentList*$^?$) *AsstOp AsstExp*	
	|	*Primary* (*ArgumentList*$^?$) *AsstOp AsstExp*	
LeftHandSide	::=	*ExpName*	*(20.108)*
	|	*FieldAccess*	
AsstOp	::=	=	*(20.14)*
	|	*=	
	|	/=	
	|	%=	
	|	+=	
	|	-=	
	|	<<=	
	|	>>=	
	|	>>>=	
	|	&=	
	|	^=	
	|	|=	

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

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

There are three syntactic forms of assignment:

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

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

```
x ◇= e
```

behaves just like

```
x = x ◇ e.
```

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

```
{
  // The order of these evaluations is not specified
  val aa = A();  // Evaluate A() once
  val ii = I();  // Evaluate I() once
  val bb = B();  // Evaluate B() once
  // But they happen before this:
  val tmp = aa(ii) + bb; // read aa(ii)
  aa(ii) = tmp;  // write sum back to aa(ii)
}
```

11.8 Increment and decrement

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

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

These operators work for any `x` for which `1 as T` is defined, where `T` is the type of `x`.

11.9 Numeric Operations

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

Promoting a numeric value to a longer numeric type results in either sign extension or zero extension depending on whether the target type is signed or unsigned. For example, `(255 as UByte) as UInt` is 255 while `(255 as Byte) as Int` is -1.

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

11.9.1 Conversions and coercions

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

Example:

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

11.9.2 Unary plus and unary minus

The unary + operation on numbers is an identity function. The unary - operation on signed numbers is a negation function. On unsigned numbers, these are two's-complement arithmetic; the unsigned number types are closed under unary -. For example, `-(0x0F as UByte)` is `(0xF1 as UByte)`.

11.10 Bitwise complement

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

11.11 Binary arithmetic operations

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

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

11.12 Binary shift operations

When operands of the binary shift operations are of integral type, the expression performs bitwise shifts. The type of the result is the type of the left operand. The right operand, describing a number of bits, must be a Long: `x << y`.

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

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

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

11.13 Binary bitwise operations

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

11.14 String concatenation

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

The type of the result is `String`.

For example, `"one " + 2 + true` evaluates to `one 2true`.

11.15 Logical negation

The unary `!` operator applied to type `x10.lang.Boolean` performs logical negation. The type of the result is `Boolean`. If the value of the operand is `true`, the result is `false`; if if the value of the operand is `false`, the result is `true`.

11.16 Boolean logical operations

The binary operations & and | at type `Boolean` perform Boolean logical operations.

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

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

11.17 Boolean conditional operations

The binary && and || operations, on `Boolean` values, give conditional or short-circuiting Boolean operations.

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

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

11.18 Relational operations

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

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

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

11.19 Conditional expressions

ConditionalExp ::= *ConditionalOrExp* `?` *Exp* `:` *ConditionalExp* *(20.45)*

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

The type of the condition must be `Boolean`. The type of the conditional expression is some common ancestor (as constrained by §4.10) of the types of the consequent and the alternative.

Example: `a == b ? 1 : 2` *evaluates to* `1` *if* `a` *and* `b` *are the same, and* `2` *if they are different. As the type of* `1` *is* `Long{self==1}` *and of* `2` *is* `Long{self==2}`*, the type of the conditional expression has the form* `Long{c}`*, where* `self==1` *and* `self==2` *both imply* `c`*. For example, it might be* `Long{true}` *– or perhaps it might be a more accurate type, like* `Long{self != 8}`*. Note that this term has no most accurate type in the X10 type system.*

The subexpression not selected is not evaluated.

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

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

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

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

11.20 Stable equality

EqualityExp ::= *RelationalExp* *(20.59)*
| *EqualityExp* == *RelationalExp*
| *EqualityExp* != *RelationalExp*
| *Type* == *Type*

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

- If `a` and `b` are values of object type, then `a==b` holds if `a` and `b` are the same object.
- If one operand is `null`, then `a==b` holds iff the other is also `null`.
- The structs in `x10.lang` have unsurprising concepts of ==:
  - In `Boolean`, `true == true` and `false == false`.
  - In `Char`, `c == d` iff `c.ord() == d.ord()`.
  - Equality in `Double` and `Float` is IEEE floating-point equality.
  - Two `GlobalRef`s are == if they refer to the same object.

  - The integral types, `Byte`, `Short`, `Int`, `Long`, and their unsigned versions, use binary equality.

- If the operands both have struct type and are not in `x10.lang`, then they must be structurally equal; that is, they must be instances of the same struct and all their fields or components must be ==.
- The definition of equality for function types is specified in §10.4.
- No implicit coercions are performed by ==.
- It is a static error to have an expression `a == b` if the types of `a` and `b` are disjoint.

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

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

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

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

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

11.20.1 No Implicit Coercions for ==

== is a primitive operation in X10 – one of very few. Most operations, like + and <=, are defined as `operator`s. == and `!=` are not. As non-`operator`s, they need not and do not follow the general method resolution procedure of §8.12. In particular, while `operator`s perform implicit conversions on their arguments, == and `!=` do not.

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

The disadvantage is that certain straightforward-looking idioms do not work. One may not test that an `Int` variable is == to a long like `0`:

```
//ERROR: for(var i : Int = 0n; i != 100; i++)  {}
```

A `Int` like `i` can never == a `Long` like `100`. Because == does not permit implicit coercions, `i` stays a `Int`. The loop must be written with a comparison of two `Int`s:

```
for(var i : Int = 0n; i != 100n; i++) {}
```

Because the operation <= is a regular operator, and thus uses coercions in its arguments, it is legal (although not recommended) to write the loop as

```
for(var i : Int = 0n; i <= 100; i++) {}
```

In this formulation, `i` will be coerced to a `Long` on each loop iteration so it can be compared using <= against `100`.

Example: *If numbers are cast to* `Any`*, they are compared as values of type* `Any`*, not as numbers. For example,* `1 as Any == 1ul as Any` *is not a static error (because it is comparing two values of type* `Any`*), and returns* `false` *(because the two* `Any` *values refer to different values — indeed, to values of different types,* `Int` *and* `ULong`*).*

11.20.2 Non-Disjointness Requirement

It is, in many cases, a static error to have an expression `a==b` where `a` and `b` could not possibly be equal, based on their types. (In one case it is a static error even though they *could* be equal.) This is a practical codicil to §11.20.1. Consider the illegal code

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

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

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

Example: *Nonetheless, it is possible to wind up comparing values of different numeric types. Even though, say,* `0n` *and* `0L` *represent the same number, they are different values and of different types, and hence,* `0n != 0L`*. The expression* `0n == 0L` *does not compile. However, if you hide type information from X10, you can get a similar expression to compile:*

```
val a : Any = 0n;
val b : Any = 0L;
assert a != b;
```

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

- For objects, the story is different. Unconstrained object types can always be compared for equality. Given objects of unrelated classes `a:Person` and `b:Theory`, `a==b` could be true if `a==null` and `b==null`. Despite this, `a==b` is a static error, because it is generally a programming mistake. `a as Any == b as Any` can be used to express the equality, if it is necessary.
- Constraints are ignored in determining whether an equality is statically allowed. For example, the following is allowed:

```
def m(a:Long{self==1}, b:Long{self==2}) = (a==b);
```

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

11.21 Allocation

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

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

The allocation expression may have an optional class body. In this case, an anonymous subclass of the given class is allocated. An anonymous class allocation may also specify a single super-interface rather than a superclass; such an anonymous class does not have a superclass.

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

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

§8.14.1 describes allocation expressions for inner classes.

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

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

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

11.22 Casts and Conversions

CastExp ::= *Primary* *(20.30)*
| *ExpName*
| *CastExp* `as` *Type*

The cast and conversion operation `e as T` may be used to force an expression into a given type `T`, if is permissible at run time, and either a compile-time error or a runtime exception (`x10.lang.ClassCastException`) if it is not.

The `e as T` operation comes in two forms. Which form applies depends on both the source type (the type of `e`) and the target type `T`.

- **Cast:** A cast makes a value have a different type, without changing the value's identity. For example, `"a String" as Any` simply reconsiders the `String` object as an `Any`. This cast does not need to do any run-time computation, since every `String` is an `Any`; a cast in the reverse direction, from `Any` to `String`, would need a run-time check that the `Any` was in fact a `String`. Casts are all system-defined, following from the X10 type system.
- **Conversions:** A conversion takes a value of one type and produces one of a different type which, conceptually, means the same thing. For example, `1 as Float` is a conversion. It performs some computation on `1` to come up with a `Float` value. Conversions are all library- or user-defined.

11.22.1 Casts

A cast `v as T2` re-imagines a value `v` of one type `T1` as being a value of another type `T2`. The value itself does not change, nor is a new value computed. The only run-time computation that happens is to check that `v` is indeed a value of type `T2` (which, in many cases, is unnecessary), and auto-boxing (§9.2).

Casts to generic types can be unsound. The instantiations of the generic types have constraints, but the runtime does not preserve the representation of these types. See §4.5.5 for more details.

There are two forms of casts. *Upcasts* happen when `T1 <: T2`, that is, when a value is being cast to a more general type. Upcasts often don't require any runtime computation at all, since, if `T1 <: T2` and `T2 isref`, every value of type `T1` is automatically one of type `T2`. For example, `"A String" as Any` is a trivial upcast: every `String` can simply be used as a value of type `Any` because it is already represented as a heap-allocated object. Upcasts from structs to interface types however do require auto-boxing, such as `1 as Any`.

Downcasts are casts which are not upcasts. Often they are recasting something from a more general to a more specific type, though casts that cross the type hierarchy laterally are also called downcasts.

```
val ob : Any = "a String" as Any; // upcast
val st : String = ob as String;   // downcast
assert st == ob;
```

Example:

In the following example, `Snack` *and* `Crunchy` *are unrelated interfaces: neither inherits from the other. Some objects are both; some are one but not the other. Casting from a* `Crunchy` *to a* `Snack` *requires confirming that the value being cast is indeed a* `Snack`.

```
interface Snack {}
interface Crunchy {}
class Pretzel implements Snack, Crunchy{}
class Apricot implements Snack{}
class Gravel  implements Crunchy{}
class Example{
  def example(crunchy : Crunchy) {
    if (crunchy instanceof Snack) {
       val snack = crunchy as Snack;
    } } }
```

An upcast `v as T2` requires no computation. A downcast `v as T2` requires testing that `v` really is a value of type `T2`. In either case, the cast returns the value `v`; casts do not change value identity.

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

- If `T` is a class, then the first half of the cast succeeds if the run-time value of `E` is an instance of class `T`, or of a subclass.
- If `T` is an interface, then the first half of the cast succeeds if the run-time value of `E` is an instance of a class or struct implementing `T`.
- If `T` is a struct type, then the first half of the cast succeeds if the run-time value of `E` is an instance of `T`.
- If `T` is a function type, then the first half of the cast succeeds if the run-time value of `X` is a function of that type, or an object or struct which implements it.

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

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

because the compiler knows that 1, which has type Long{self==1}, cannot possibly be of type Long{self==2}.

If, for some reason, you need to write one of these forbidden casts, cast to Any first. (1 as Any) as Long{self==2} always returns false, but compiles.

11.22.2 Explicit Conversions

Explicit conversions are written with the same syntax as casts: v as T2. Explicit conversions transform a value of one type T1 to an unrelated type T2. Unlike casts, conversions *do* execute code, and *may* (and generally do) return new values.

Explicit conversions do not arise spontaneously, as casts do. They may be programmed directly, using the operator syntax of §8.7.3. Implicit coercions can also be called explicitly as conversions. (The reverse is not true – explicit conversions cannot be used as implicit conversions.)

The numeric types in x10.lang have explicit conversions, as described in §11.23.1. These conversions enable 1 as Float and the like.

Example: *The following class has an explicit conversion from* Long *to* Knot*, and an implicit one from* String *to* Knot. a *uses the explicit conversion,* b *uses the implicit coercion, and* c *uses the implicit coercion explicitly.*

```
class Knot(s:String){
  public def is(t:String):Boolean = s.equals(t);
  // explicit conversion
  public static operator (n:Long) as Knot = new Knot("knot-" + n);
  // implicit coercion
  public static operator (s:String):Knot = new Knot(s);
  // using them
  public static def example() {
     val a : Knot = 1 as Knot;
     val b : Knot = "frayed";
     val c : Knot = "three" as Knot;
     assert a.is("knot-1") && b.is("frayed") && c.is("three");
  }
}
```

11.22.3 Resolving Ambiguity

If v as T could either be a cast or an explicit coercion, X10 treats its as a cast. With the VERBOSE compiler flag, this is flagged as a warning.

Example: *The* Person *class provides an explicit conversion from its subclass* Fop *to itself. However, since* Fop *is a subclass of* Person*, using the* as *operator invokes the upcast, rather than the explicit conversion. This is visible in the example because the*

user-defined operator `f as Person` *returns* `new Person()` *(just like the* `asPerson` *method), while the upcast returns* `f` *itself.*

```
class Person {
  static operator (f:Fop) as Person = new Person();
  static def asPerson(f:Fop) = new Person();
  public static def example() {
     val f = new Fop();
     val cast = f as Person; // WARNING on this line
     assert cast == f;
     val meth = asPerson(f);
     assert meth != f;
  }
}
class Fop extends Person {}
```

The definition of an explicit conversion in this case is of little value, since any use of it in the `f as Person` *syntax will invoke the upcast.*

11.23 Coercions and conversions

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast. A *conversion* may change object identity if the type being converted to is not the same as the type converted from. X10 permits both user-defined coercions and conversions (§11.23.2).

11.23.1 Coercions

CastExp ::= *Primary* *(20.30)*
| *ExpName*
| *CastExp* `as` *Type*

Subsumption coercion. A value of a subtype may be implicitly coerced to any supertype.

Example: *If* `Child <: Person` *and* `val rhys:Child`*, then* `rhys` *may be used in any context that expects a* `Person`*. For example,*

```
class Example {
  def greet(Person) = "Hi!";
  def example(rhys: Child) {
     greet(rhys);
  }
}
```

Similarly, `2` *(whose innate type is* `Long{self==2}`*) is usable in a context requiring a non-zero integer (*`Long{self != 0}`*).*

Explicit Coercion (Casting with `as`) All classes and interfaces allow the use of the `as` operator for explicit type coercion. Any class or interface may be cast to any interface. Any interface may be cast to any class. Also, any interface can be cast to a struct that implements (directly or indirectly) that interface.

Example: *In the following code, a* `Person` *is cast to* `Childlike`*. There is nothing in the class definition of* `Person` *that suggests that a* `Person` *can be* `Childlike`*. However, the* `Person` *in question,* `p`*, is actually a* `HappyChild` *— a subclass of* `Person` *— and is, in fact,* `Childlike`*.*

Similarly, the `Childlike` *value* `cl` *is cast to* `Happy`*. Though these two interfaces are unrelated, the value of* `cl` *is, in fact,* `Happy`*. And the* `Happy` *value* `hc` *is cast to the class* `Child`*, though there is no relationship between the two, but the actual value is a* `HappyChild`*, and thus the cast is correct at runtime.*

`Cyborg` *is a struct rather than a class. So, it cannot have substructs, and all the interfaces of all* `Cyborg`*s are known: a* `Cyborg` *is* `Personable`*, but not* `Childlike` *or* `Happy`*. So, it is correct and meaningful to cast* `r` *to* `Personable`*. There is no way that a cast to* `Childlike` *could succeed, so* `r as Childlike` *is a static error.*

```
interface Personable {}
class Person implements Personable {}
interface Childlike extends Personable {}
class Child extends Person implements Childlike {}
struct Cyborg implements Personable {}
interface Happy {}
class HappyChild extends Child implements Happy {}
class Example {
  static def example() {
    var p : Person = new HappyChild();
    // class -> interface
    val cl : Childlike = p as Childlike;
    // interface -> interface
    val hc : Happy = cl as Happy;
    // interface -> class
    val ch : Child = hc as Child;
    var r : Cyborg = Cyborg();
    val rl : Personable = r as Personable;
    // ERROR: val no = r as Childlike;
  }
}
```

If the value coerced is not an instance of the target type, and no coercion operators that can convert it to that type are defined, a `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

It is a static error, rather than a `ClassCastException`, when the cast is statically determinable to be impossible.

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

- Casting a number to a *wider* numeric type is safe and effective, and can be done by an implicit conversion as well as an explicit coercion. For example, `4 as Long` produces the `Long` value of 4.
- Casting a floating-point value to an integer value truncates the digits after the decimal point, thereby rounding the number towards zero. `54.321 as Int` is `54n`, and `-54.321 as Int` is `-54n`. If the floating-point value is too large to represent as that kind of integer, the coercion returns the largest or smallest value of that type instead: `1e110 as Int` is `Int.MAX_VALUE`, *viz.* `2147483647`.
- Casting a `Double` to a `Float` normally truncates binary digits: `0.12345678901234567890 as Float` is approximately `0.12345679f`. This can turn a nonzero `Double` into `0.0f`, the zero of type `Float`: `1e-100 as Float` is `0.0f`. Since `Double`s can be as large as about `1.79E308` and `Float`s can only be as large as about `3.4E38f`, a large `Double` will be converted to the special `Float` value of `Infinity`: `1e100 as Float` is `Infinity`.
- Integers are coerced to smaller integer types by truncating the high-order bits. If the value of the large integer fits into the smaller integer's range, this gives the same number in the smaller type: `12 as Byte` is the `Byte`-sized 12, `-12 as Byte` is -12. However, if the larger integer *doesn't* fit in the smaller type, the numeric value and even the sign can change: `254 as Byte` is the `Byte`sized `-2y`.
- Casting an unsigned integer type to a signed integer type of the same size (*e.g.*, `UInt` to `Int`) preserves 2's-complement bit pattern (*e.g.*, `UInt.MAX_VALUE as Int == -1n`. Casting an unsigned integer type to a signed integer type of a different size is equivalent to first casting to an unsigned integer type of the target size, and then casting to a signed integer type.
- Casting a signed integer type to an unsigned one is similar.

User-defined Coercions

Users may define coercions from arbitrary types into the container type `B`, and coercions from `B` to arbitrary types, by providing `static operator` definitions for the `as` operator in the definition of `B`.

Example:

```
class Bee {
  public static operator (x:Bee) as Long = 1;
  public static operator (x:Long) as Bee = new Bee();
  def example() {
    val b:Bee = 2 as Bee;
    assert (b as Long) == 1;
  }
}
```

11.23.2 Conversions

Widening numeric conversion. A numeric type may be implicitly converted to a wider numeric type. In particular, an implicit conversion may be performed between a numeric type and a type to its right, below:

```
Byte < Short < Int < Long < Float < Double
UByte < UShort < UInt < ULong
```

Furthermore, an unsigned integer value may be implicitly coerced to a signed type large enough to hold any value of the type: `UByte` to `Short`, `UShort` to `Int`, `UInt` to `Long`. There are no implicit conversions from signed to unsigned numbers, since they cannot treat negatives properly.

There are no implicit conversions in cases when overflow is possible. For example, there is no implicit conversion between `Int` and `UInt`. If it is necessary to convert between these types, use `n as Int` or `n as UInt`, generally with a test to ensure that the value will fit and code to handle the case in which it does not.

String conversion. Any value that is an operand of the binary + operator may be converted to `String` if the other operand is a `String`. A conversion to `String` is performed by invoking the `toString()` method.

User defined conversions. The user may define implicit conversion operators from type `A` *to* a container type `B` by specifying an operator in `B`'s definition of the form:

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

The return type `T` should be a subtype of `B`. The return type need not be specified explicitly; it will be computed in the usual fashion if it is not. However, it is good practice for the programmer to specify the return type for such operators explicitly. The return type can be more specific than simply `B`, for cases when there is more information available.

Example: *The code for* `x10.lang.Point` *contains a conversion from a* `Rail`*s of longs to* `Point`*s of the same length:*

```
public operator (r: Rail[Long]): Point(r.size)
       = make(r);
```

This conversion is used whenever a `Rail` *of integers appears in a context that requires a* `Point`*, such as subscripting. Note that* `a` *requires a* `Point` *of rank 2 as a subscript, and that a two-element* `Rail` *(like* `[2,4]`*) is converted to a* `Point(2)`.

```
val a = new Array[String](Region.make(2..3, 4..5), "hi!");
a([2,4]) = "converted!";
```

11.24 instanceof

X10 permits types to be used in an in instanceof expression to determine whether an object is an instance of the given type:

RelationalExp ::= *ShiftExp* *(20.144)*
| *HasZeroConstraint*
| *SubtypeConstraint*
| *RelationalExp* < *ShiftExp*
| *RelationalExp* > *ShiftExp*
| *RelationalExp* <= *ShiftExp*
| *RelationalExp* >= *ShiftExp*
| *RelationalExp* `instanceof` *Type*

In the above expression, *Type* is any type. At run time, the result of `e instanceof T` is `true` if the value of `e` is an instance of type `T`. Otherwise the result is `false`. This determination may involve checking that the constraint, if any, associated with the type is true for the given expression.

For example, `3 instanceof Long{self==x}` is an overly-complicated way of saying `3==x`.

However, it is a static error if `e` cannot possibly be an instance of `C{c}`; the compiler will reject `1 instanceof Long{self == 2}` because 1 can never satisfy `Long{self == 2}`. Similarly, `1 instanceof String` is a static error, rather than an expression always returning false.

If `x instanceof T` returns `true` for some value `x` and type `T`, then `x as T` will evaluate normally.

Limitation: X10 does not currently handle `instanceof` of generics in the way you might expect. For example, `r instanceof Array[Long{self != 0}]` does not test that every element of `r` is non-zero; instead, the compiler gives an unsound cast warning.

11.24.1 Nulls in Constraints in `as` and `instanceof`

Both `as` and `instanceof` expressions can throw `NullPointerException`s, if the constraints involve selecting fields or properties of variables which are bound to `null`.

These operations give some guarantees for any type `T`, constraint `c`, and class `SomeObj` with an `a` field:

1. `null instanceof T` always returns `false`. It never throws an exception. It never returns `true`, not even in cases where `null` could be assigned to a variable of type `T`.
2. `null` can be assigned to a variable of type `SomeObj{self.a==b}`, or, more broadly, to a variable of a constrained object type whose constraint does not explicitly exclude `null`. This is the case even though `null.a==b` would throw a `NullPointerException` rather than evaluate to either `true` ‘ or `false`.
3. If `x instanceof T` returns `true`, then `x as T` is a cast rather than an explicit conversion, and will succeed and have static type `T`.
4. If the static type of `x` is `T`, then `x instanceof T` and `x as T` will do one of these:
   - Succeed, with `x instanceof T` returning `true`, and `x as T` being a cast and returning value of type `T`; **or**
   - Throw a `NullPointerException`.
   - If `x==null`, then `x instanceof T` will always return `false`, and `x as T` will either return a `null` of type `T`, or, if `T` has a constraint which tries to extract a field of `x`, will throw a `NullPointerException`.
5. If `x instanceof SomeObj{self.a==b}` is `true`, then `x.a==b` evaluates to `true` (rather than a null pointer exception). Indeed, in general, if `x instanceof T{c}` succeeds, then `cc` evaluates to true, where `cc` is `c` with suitable occurrences of `self` replaced by `x`.

11.25 Subtyping expressions

SubtypeConstraint ::= *Type* `<:` *Type* | *Type* `:>` *Type* *(20.155)*

The subtyping expression $\texttt{T}_1$ `<:` $\texttt{T}_2$ evaluates to `true` if $\texttt{T}_1$ is a subtype of $\texttt{T}_2$.

The expression $\texttt{T}_1$ `:>` $\texttt{T}_2$ evaluates to `true` if $\texttt{T}_2$ is a subtype of $\texttt{T}_1$.

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

Example: *Subtyping expressions are particularly useful in giving constraints on generic types.* `x10.util.Ordered[T]` *is an interface whose values can be compared with values of type* `T`*. In particular,* `T <: x10.util.Ordered[T]` *is true if values of type* `T` *can be compared to other values of type* `T`*. So, if we wish to define a generic class* `OrderedList[T]`*, of lists whose elements are kept in the right order, we need the elements to be ordered. This is phrased as a constraint on* `T`*:*

```
class OrderedList[T]{T <: x10.util.Ordered[T]} {
  // ...
}
```

11.26 Rail Constructors

Primary ::= `[` *ArgumentList*$^?$ `]`

X10 includes short syntactic forms for constructing Rails. Enclose some expressions in brackets to put them in a Rail:

```
val ints <: Rail[Long] = [1,3,7,21];
```

The expression `[`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`]` produces an `n`-element `Rail[T]`, where `T` is the computed common supertype (§4.10) of the types of of the expressions $\mathtt{e}_i$.

Example: *The type of* `[0,1,2]` *is* `Rail[Long]`*. The type of* `[0]` *is* `Rail[Long{self==0}]`.

To make a `Rail[Long]` *containing just a* `0`*, use* `[0 as Long]`*. The* `as Long` *masks more detailed type information, such as the fact that* `0` *is zero.*

Example: *Occasionally one does actually need* `Rail[Long{self==0}]`*, or, say,* `Rail[Eel{self != null}]`*, a rail of non-null* `Eel`*s. For these cases, cast one or more of the elements of the rail to the desired type, and the rail constructor will do the right thing.*

```
val zero <: Rail[Long{self == 0}]
          = [0];
val non1 <: Rail[Long{self != 1}]
          = [0 as Long{self != 1}];
val eels <: Rail[Eel{self != null}]
          = [new Eel() as Eel{self != null},
             new Eel(), new Eel()];
```

11.27 Parenthesized Expressions

If `E` is any expression, `(E)` is an expression which, when evaluated, produces the same result as `E`.

Example: *The main use of parentheses is to write complex expressions for which the standard precedence order of operations is not appropriate:* `1+2*3` *is 7, but* `(1+2)*3` *is 9.*

Similarly, but perhaps less familiarly, parentheses can disambiguate other expressions. In the following code, `funny.f` *is a field-selection expression, and so* `(funny.f)()` *means "select the* `f` *field from* `funny`*, and evaluate it". However,* `funny.f()` *means "evaluate the* `f` *method on object* `funny`*."*

```
class Funny {
  def f () = 1;
  val f = () => 2;
  static def example() {
    val funny = new Funny();
    assert funny.f() == 1;
    assert (funny.f)() == 2;
  }
}
```

Note that this does *not* mean that `E` and `(E)` are identical in all respects; for example, if `i` is an `Long` variable, `i++` increments `i`, but `(i)++` is not allowed. `++` is an assignment; it operates on variables, not merely values, and `(i)` is simply an expression whose *value* is the same as that of `i`.

12 Statements

This chapter describes the statements in the sequential core of X10. Statements involving concurrency and distribution are described in §14.

12.1 Empty statement

The empty statement `;` does nothing.

Example: *Sometimes, the syntax of X10 requires a statement in some position, but you do not actually want to do any computation there. The following code searches the rail* `a` *for the value* `v`*, assumed to appear somewhere in* `a`*, and returns the index at which it was found. There is no computation to do in the loop body, so we use an empty statement there.*

```
static def search[T](a: Rail[T], v: T):Long {
  var i : Long;
  for(i = 0L; a(i) != v; i++)
     ;
  return i;
}
```

12.2 Local variable declaration

LocVarDecln	::=	*Mods*$^?$ *VarKeyword VariableDeclrs*	*(20.110)*
	|	*Mods*$^?$ *VarDeclsWType*	
	|	*Mods*$^?$ *VarKeyword FormalDeclrs*	
LocVarDeclnStmt	::=	*LocVarDecln* ;	*(20.111)*
VarDeclsWType	::=	*VarDeclWType*	*(20.201)*
	|	*VarDeclsWType* , *VarDeclWType*	
VariableDeclrs	::=	*VariableDeclr*	*(20.204)*
	|	*VariableDeclrs* , *VariableDeclr*	
VariableInitializer	::=	*Exp*	*(20.205)*
FormalDeclrs	::=	*FormalDeclr*	*(20.78)*
	|	*FormalDeclrs* , *FormalDeclr*	

Short-lived variables are introduced by local variables declarations, as described in §12.2. Local variables may be declared only within a block statement (§12.3). The scope of a local variable declaration is the subsequent statements in the block.

```
if (a > 1) {
   val b = a/2;
   var c : Long = 0;
   // b and c are defined here
}
// b and c are not defined here.
```

Variables declared in such statements shadow variables of the same name declared elsewhere. A local variable of a given name, say `x`, cannot shadow another local variable or parameter named `x` unless there is an intervening method, constructor, initializer, or closure declaration.

Example: *The following code illustrates both legal and illegal uses of shadowing. Note that a shadowed* field *name* `x` *can still be accessed as* `this.x`.

```
class Shadow{
  var x : Long;
  def this(x:Long) {
     // Parameter can shadow field
     this.x = x;
  }
  def example(y:Long) {
     val x = "shadows a field";
     // ERROR: val y = "shadows a param";
     val z = "local";
     for (a in [1,2,3]) {
        // ERROR: val x = "can't shadow local var";
     }
     async {
```

```
        // ERROR: val x = "can't shadow through async";
      }
      val f = () => {
        val x = "can shadow through closure";
        x
      };
      class Local {
         val f = at(here) { val x = "can here"; x };
         def this() { val x = "can here, too"; }
      }
    }
  }
```

Example: *Note that recursive definitions of local variables is not allowed. There are few useful recursive declarations of objects and structs;* `x`*, in the following example, has no meaningful definition. Recursive declarations of local functions is forbidden, even though (like* `f` *below) there are meaningful uses of it.*

```
val x : Long = x + 1; // ERROR: recursive local declaration
val f : (Long)=>Long
      = (n:Long) => (n <= 2) ? 1 : f(n-1) + f(n-2);
      // ERROR: recursive local declaration
```

12.3 Block statement

Block	::=	{ *BlockStmts*$^?$ }	*(20.25)*
BlockStmts	::=	*BlockInteriorStmt*	*(20.27)*
	|	*BlockStmts BlockInteriorStmt*	
BlockInteriorStmt	::=	*LocVarDeclnStmt*	*(20.26)*
	|	*ClassDecln*	
	|	*StructDecln*	
	|	*TypeDefDecln*	
	|	*Stmt*	

A block statement consists of a sequence of statements delimited by "{" and "}". When a block is evaluated, the statements inside of it are evaluated in order. Blocks are useful for putting several statements in a place where X10 asks for a single one, such as the consequent of an `if`, and for limiting the scope of local variables.

```
if (b) {
  // This is a block
  val v = 1;
  S1(v);
  S2(v);
}
```

12.4 Expression statement

Any expression may be used as a statement.

ExpStmt	::=	*StmtExp* ;	*(20.63)*
StmtExp	::=	*Assignment*	*(20.152)*
	|	*PreIncrementExp*	
	|	*PreDecrementExp*	
	|	*PostIncrementExp*	
	|	*PostDecrementExp*	
	|	*MethodInvo*	
	|	*ObCreationExp*	

The expression statement evaluates an expression. The value of the expression is not used. Side effects of the expression occur, and may produce results used by following statements. Indeed, statement expressions which terminate without side effects cannot have any visible effect on the results of the computation.

Example:

```
class StmtEx {
  def this() {
     x10.io.Console.OUT.println("New StmtEx made");  }
  static def call() {
     x10.io.Console.OUT.println("call!");}
  def example() {
     var a : Long = 0;
     a = 1; // assignment
     new StmtEx(); // allocation
     call(); // call
  }
}
```

12.5 Labeled statement

LabeledStatement ::= *Id* : *Statement*

Statements may be labeled. The label may be used to describe the target of a `break` statement appearing within a substatement (which, when executed, ends the labeled statement), or, in the case of a loop, a `continue` as well (which, when executed, proceeds to the next iteration of the loop). The scope of a label is the statement labeled.

Example: *The label on the outer* `for` *statement allows* `continue` *and* `break` *statements to continue or break it. Without the label,* `continue` *or* `break` *would only continue or break the inner* `for` *loop.*

```
lbl : for (i in 1..10) {
   for (j in i..10) {
      if (a(i,j) == 0) break lbl;
      if (a(i,j) == 1) continue lbl;
      if (a(i,j) == a(j,i)) break lbl;
   }
}
```

In particular, a block statement may be labeled: `L:{S}`. This allows the use of `break L` within `S` to leave `S`, which can, if carefully used, avoid deeply-nested `if`s.

Example:

```
multiphase: {
  if (!exists(filename)) break multiphase;
  phase1(filename);
  if (!suitable_for_phase_2(filename)) break multiphase;
  phase2(filename);
  if (!suitable_for_phase_3(filename)) break multiphase;
  phase3(filename);
}
// Now the file has been phased as much as possible
```

Limitation: Blocks cannot currently be labeled.

12.6 Break statement

BreakStmt ::= `break` *Id*? `;` *(20.29)*

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

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

```
var found: Boolean = false;
outer: for (i in a.range)
    for (j in a(i).range)
        if (a(i)(j) == v) {
            found = true;
            break outer;
        }
```

12.7 Continue statement

ContinueStmt ::= `continue` *Id*$^{?}$ `;` *(20.50)*

An unlabeled `continue` skips the rest of the current iteration of the innermost enclosing loop, and proceeds on to the next. A labeled `continue` does the same to the enclosing loop with that label. It is illegal to continue a loop not defined in the current method, constructor, initializer, or closure. `continue` is only allowed in sequential code.

12.8 If statement

IfThenStmt ::= `if (` *Exp* `)` *Stmt* *(20.93)*
IfThenElseStmt ::= `if (` *Exp* `)` *Stmt* `else` *Stmt* *(20.92)*

An if statement comes in two forms: with and without an else clause.

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

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

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

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

could be intended to mean either

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

or

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

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

12.9 Switch statement

SwitchStmt	::=	`switch (` *Exp* `)` *SwitchBlock*	*(20.162)*
SwitchBlock	::=	`{` *SwitchBlockGroups*$^?$ *SwitchLabels*$^?$ `}`	*(20.157)*
SwitchBlockGroups	::=	*SwitchBlockGroup*	*(20.159)*
	|	*SwitchBlockGroups SwitchBlockGroup*	
SwitchBlockGroup	::=	*SwitchLabels BlockStmts*	*(20.158)*
SwitchLabels	::=	*SwitchLabel*	*(20.161)*
	|	*SwitchLabels SwitchLabel*	
SwitchLabel	::=	`case` *ConstantExp* `:`	*(20.160)*
	|	`default :`	

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

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

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

Example: *In this* `switch`*, case* `1` *falls through to case* `2`*. The other cases are separated by* `break`*s.*

```
switch (i) {
  case 1n: println("one, and ");
  case 2n: println("two");
           break;
  case 3n: println("three");
           break;
  default: println("Something else");
           break;
}
```

12.10 While statement

WhileStmt	::=	`while (` *Exp* `)` *Stmt*	*(20.208)*

A while statement evaluates a `Boolean`-valued condition and executes a loop body if `true`. If the loop body completes normally (either by reaching the end or via a `continue` statement with the loop header as target), the condition is reevaluated and the loop repeats if `true`. If the condition is `false`, the loop exits.

Example: *A loop to execute the process in the Collatz conjecture (a.k.a. 3n+1 problem, Ulam conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, and Syracuse problem) can be written as follows:*

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

12.11 Do–while statement

DoStmt ::= `do` *Stmt* `while (` *Exp* `) ;` *(20.56)*

A `do-while` statement executes the loop body, and then evaluates a `Boolean`-valued condition expression. If `true`, the loop repeats. Otherwise, the loop exits.

12.12 For statement

ForStmt	::=	*BasicForStmt*	*(20.74)*
	|	*EnhancedForStmt*	
BasicForStmt	::=	`for (` *ForInit*$^?$ `;` *Exp*$^?$ `;` *ForUpdate*$^?$ `)` *Stmt*	*(20.22)*
ForInit	::=	*StmtExpList*	*(20.73)*
	|	*LocVarDecln*	
ForUpdate	::=	*StmtExpList*	*(20.75)*
StmtExpList	::=	*StmtExp*	*(20.153)*
	|	*StmtExpList* `,` *StmtExp*	
EnhancedForStmt	::=	`for (` *LoopIndex* `in` *Exp* `)` *Stmt*	*(20.58)*
	|	`for (` *Exp* `)` *Stmt*	

`for` statements provide bounded iteration, such as looping over a list. It has two forms: a basic form allowing near-arbitrary iteration, *a la* C, and an enhanced form designed to iterate over a collection.

A basic `for` statement provides for arbitrary iteration in a somewhat more organized fashion than a `while`. The loop `for(init; test; step)body` is similar to:

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

except that `continue` statements which continue the `for` loop will perform the `step`, which, in the `while` loop, they will not do.

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

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

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

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

An enhanced for statement is used to iterate over a collection, or other structure designed to support iteration by implementing the interface `Iterable[T]`. The loop variable must be of type `T`, or destructurable from a value of type `T` (§5). Each iteration of the loop binds the iteration variable to another element of the collection. The loop `for(x in c)S` behaves like:

```
val iterator: Iterator[T] = c.iterator();
while (iterator.hasNext()) {
  val x : T = iterator.next();
  S();
}
```

A number of library classes implement `Iterable`, and thus can be iterated over. For example, iterating over a `Rail` iterates the elements stored in the rail.

The type of the loop variable may be supplied as `x <: T`. In this case the iterable `c` must have type `Iterable[U}` for some `U <: T`, and `x` will be given the type `U`.

Example: *This loop adds up the elements of a* `List[Long]`. *Note that iterating over a list yields the elements of the list, as specified in the* `List` *API.*

```
static def sum(a:x10.util.List[Long]):Long {
  var s : Long = 0;
  for(x in a) s += x;
  return s;
}
```

The following code sums the elements of an integer rail.

```
static def sum(a: Rail[Long]): Long {
  var s : Long = 0;
  for(v in a) s += v;
  return s;
}
```

Iteration over a `LongRange` *is quite common. This allows looping while varying a long index:*

```
var sum : Long = 0;
for(i in 1..10) sum += i;
assert sum == 55;
```

Iteration variables have the `for` statement as scope. They shadow other variables of the same names.

12.13 Return statement

ReturnStmt ::= `return` *Exp*$^{?}$ `;` *(20.146)*

Methods and closures may return values using a `return` statement. `void` methods must return without a value; other methods must return a value of the return type.

Example: *The following code illustrates returning values from a closure and a method. The* `return` *inside of* `closure` *returns from* `closure`*, not from* `method`*.*

```
def method(x:Long) {
  val closure = (y:Long) => {return x+y;};
  val res = closure(0);
  assert res == x;
  return res == x;
}
```

12.14 Assert statement

AssertStmt ::= `assert` *Exp* `;` *(20.10)*
| `assert` *Exp* `:` *Exp* `;`

The statement `assert E` checks that the Boolean expression `E` evaluates to true, and, if not, throws an `x10.lang.Error` exception. The annotated assertion statement `assert E : F;` checks `E`, and, if it is false, throws an `x10.lang.Error` exception with `F`'s value attached to it.

Example: *The following code compiles properly.*

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

However, when run, it prints a stack trace starting with

```
x10.lang.Error: Changed my mind about a.
```

12.15 Exceptions in X10

X10 programs can throw *exceptions* to indicate unusual or problematic situations; this is *abrupt termination*. Exceptions, as data values, are instances of `x10.lang.CheckedThrowable` or its subclasses. Note that for ease of implementation X10 does not permit subclasses of `x10.lang.CheckedThrowable` to be generic, that is, take type parameters.

Exceptions may be thrown intentionally with the `throw` statement. Many primitives and library functions throw exceptions if they encounter problems; *e.g.*, dividing by zero throws an instance of `x10.lang.ArithmeticException`.

When an exception is thrown, dynamically enclosing `try-catch` blocks in the same activity can attempt to handle it. If the throwing statement in inside some `try` clause, and some matching `catch` clause catches that type of exception, the corresponding `catch` body will be executed, and the process of throwing is finished. If no statically-enclosing `try-catch` block can handle the exception, the current method call returns (abnormally), throwing the same exception from the point at which the method was called.

This process continues until the exception is handled or there are no more calling methods in the activity. In the latter case, the activity will terminate abnormally, and the exception will propagate to the activity's root; see §14.1 for details.

X10 supports both *checked* and *unchecked* exceptions. Methods are obligated to declare via a `throws` clause any checked exceptions that they might throw. However, in X10, the class library design favors unchecked exceptions: virtually all exceptions in the standard library are unchecked. Checked exceptions are defined to be any subclass of `x10.lang.CheckedThrowable` that are not also subclasses of either `x10.lang.Exception` or `x10.lang.Error`. All of the concrete exception classes in the X10 standard library are subclasses of either `Exception` or `Error`.

12.16 Throw statement

ThrowStmt ::= `throw` *Exp* `;` *(20.163)*

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

Example: *The following code checks if an index is in range and throws an exception if not.*

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

12.17 Try–catch statement

TryStmt	::=	`try` *Block Catches*	*(20.164)*
	|	`try` *Block* $Catches^{?}$ *Finally*	
Catches	::=	*CatchClause*	*(20.32)*
	|	*Catches CatchClause*	
CatchClause	::=	`catch (` *Formal* `)` *Block*	*(20.31)*
Finally	::=	`finally` *Block*	*(20.71)*

Exceptions are handled with a `try` statement. A `try` statement consists of a `try` block, zero or more `catch` blocks, and an optional `finally` block.

First, the `try` block is evaluated. If the block throws an exception, control transfers to the first matching `catch` block, if any. A `catch` matches if the value of the exception thrown is a subclass of the `catch` block's formal parameter type.

The `finally` block, if present, is evaluated on all normal and exceptional control-flow paths from the `try` block. If the `try` block completes normally or via a `return`, a `break`, or a `continue` statement, the `finally` block is evaluated, and then control resumes at the statement following the `try` statement, at the branch target, or at the caller as appropriate. If the `try` block completes exceptionally, the `finally` block is evaluated after the matching `catch` block, if any, and when and if the `finally` block finishs normally, the exception is rethrown.

The parameter of a `catch` block has the block as scope. It shadows other variables of the same name.

Example: *The* `example()` *method below executes without any assertion errors*

```
class Example {
  class ThisExn extends Exception {}
  class ThatExn extends Exception {}
  var didFinally : Boolean = false;
  def example(b:Boolean) {
    try {
       throw b ? new ThatExn() : new ThisExn();
    }
    catch(ThatExn) {return true;}
```

```
    catch(ThisExn) {return false;}
    finally {
       this.didFinally = true;
    }
  }
  static def doExample() {
    val e = new Example();
    assert e.example(true);
    assert e.didFinally == true;
  }
}
```

Limitation: Constraints on exception types in `catch` blocks are not currently supported.

12.18 Assert

The `assert` statement `assert B;` checks that the Boolean expression `B` evaluates to true. If so, computation proceeds. If not, it throws `x10.lang.AssertionError`.

The extended form `assert B:A;` is similar, but provides more debugging information. The value of the expression `A` is available as part of the `AssertionError`, *e.g.*, to be printed on the console.

Example: `assert` *is useful for confirming properties that you believe to be true and wish to rely on. In particular, well-chosen* `assert`*s make a program robust in the face of code changes and unexpected uses of methods. For example, the following method compute percent differences, but asserts that it is not dividing by zero. If the mean is zero, it throws an exception, including the values of the numbers as potentially useful debugging information.*

```
static def percentDiff(x:Double, y:Double) {
  val diff = x-y;
  val mean = (x+y)/2;
  assert mean != 0.0  : [x,y];
  return Math.abs(100 * (diff / mean));
}
```

At times it may be considered important not to check `assert` statements; *e.g.*, if the test is expensive and the code is sufficiently well-tested. The `-noassert` command line option causes the compiler to ignore all `assert` statements.

13 Places

An X10 place is a repository for data and activities, corresponding loosely to a process or a processor. Places induce a concept of "local". The activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote places may take orders of magnitude longer. X10's system of places is designed to make this obvious. Programmers are aware of the places of their data, and know when they are incurring communication costs, but the actual operation to do so is easy. It's not hard to use non-local data; it's simply hard to to do so accidentally.

The set of places available to a computation is determined at the time that the program is started, and remains fixed through the run of the program. See the `README` documentation on how to set command line and configuration options to set the number of places.

Places are first-class values in X10, as instances `x10.lang.Place`. `Place` provides a number of useful ways to query places, such as `Place.places()`, which returns a `PlaceGroup` of the places available to the current run of the program.

Objects and structs are created in a single place – the place that the constructor call was running in. They cannot change places. They can be *copied* to other places, and the special library struct `GlobalRef` allows values at one place to point to values at another.

13.1 The Structure of Places

Places are numbered starting at 0; the number is stored in the field `pl.id`. The method `Place.numPlaces()` returns the number of Places in the current execution of the program. The program starts by executing a `main` method at `Place.FIRST_PLACE`, which is `Place.places()(0)`; see §14.4.

13.2 `here`

The variable `here` is always bound to the place at which the current computation is running, in the same way that `this` is always bound to the instance of the current

class (for non-static code), or `self` is bound to the instance of the type currently being constrained. `here` may denote different places in the same method body or even the same expression, due to place-shifting operations.

This is not unusual for automatic variables: `self` denotes two different values (one `List`, one `Long`) when one describes a non-null list of non-zero numbers as `List[Long{self!=0}]{self!=null}`. In the following code, `here` has one value at `h0`, and a different one at `h1` (unless there is only one place).

```
val h0 = here;
val world = Place.places();
at (world.next(here)) {
  val h1 = here;
  assert (h0 != h1);
}
```

(Similar examples show that `self` and `this` have the same behavior: `self` can be shadowed by constrained types appearing inside of type constraints, and `this` by inner classes.)

The following example looks through a list of references to `Thing`s. It finds those references to things that are `here`, and deals with them.

```
public static def deal(things: List[GlobalRef[Thing]]) {
   for(gr in things) {
      if (gr.home == here) {
         val grHere =
             gr as GlobalRef[Thing]{gr.home == here};
         val thing <: Thing = grHere();
         dealWith(thing);
      }
   }
}
```

13.3 `at`: Place Changing

An activity may change place synchronously using the `at` statement or `at` expression. Like any distributed operation, it is potentially expensive, as it requires, at a minimum, two messages and the copying of all data used in the operation, and must be used with care – but it provides the basis for distributed programming in X10.

AtStmt ::= `at (` *Exp* `)` *Stmt* *(20.20)*
AtExp ::= `at (` *Exp* `)` *ClosureBody* *(20.19)*

The *PlaceExp* must be an expression of type `Place` or some subtype. For programming convenience, if *PlaceExp* is of type `GlobalRef[T]` then the `home` property of `GlobalRef` is used as the value of *PlaceExp*.

An activity may also spawn an asynchronous remote child activity. For optimal performance, it is desirable for the spawning activity to continue executing locally without waiting for the message creating the remote child activity to arrive at the destination place. X10 supports this "fire-and-forget" style of remote activity creation by special handling of the combination of `at (P) async S`. In particular, any exceptions raised during deserialization (§13.3.2) at the remote place will be reported asynchronously (as if they occured after the remote activity `async S` was spawned).

Example: *The following example creates a rail* `a` *located* `here`*, and copies it to another place.* `a` *in the second place refers to the copy. The copy is modified and examined. After the* `at` *finishes, the original is also examined, and (since only the copy, not the original, was modified) is observed to be unchanged.*

```
val a = [1,2,3];
val world = Place.places();
at(world.next(here)) {
  a(1) = 4;
  assert a(0)==1 && a(1)==4 && a(2)==3;
}
assert  a(0)==1 && a(1)==2 && a(2)==3;
```

13.3.1 Copying Values

An activity executing `at(q)S` at a place `p` evaluates `q` at place `p`, which should be a `Place`. It then moves to place `q` to execute `S`. The values variables that `S` refers to are copied (§13.3.2) to `q`, and bound to the variables of the same name. If the `at` is inside of an instance method and `S` uses `this`, `this` is copied as well. Note that a field reference `this.fld` or a method call `this.meth()` will cause `this` to be copied — as will their abbreviated forms `fld` and `meth()`, despite the lack of a visible `this`.

Note that the value obtained by evaluating `q` is not necessarily distinct from `p` (*e.g.*, `q` may be `here`). This does not alter the behavior of `at`. `at(here)S` will copy all the values mentioned in `S`, even though there is no actual change of place, and even though the original values already exist there.

On normal termination of `S` control returns to `p` and execution is continued with the statement following `at (q) S`. If `S` terminates abruptly with exception `E`, `E` is serialized into a buffer, the buffer is communicated to `p` where it is deserialized into an exception `E1` and `at (p) S` throws `E1`.

Since `at(p) S` is a synchronous construct, usual control-flow constructs such as `break`, `continue`, `return` and `throw` are permitted in `S`. All concurrency related constructs – `async`, `finish`, `atomic`, `when` are also permitted.

The `at`-expression `at(p)E` is similar, except that, in the case of normal termination of `E`, the value that `E` produces is serialized into a buffer, transported to the starting place, and deserialized, and the value of the `at`-expression is the result of deserialization.

Limitation: X10 does not currently allow `break`, `continue`, or `return` to exit from an `at`.

13.3.2 How at Copies Values

The values mentioned in S are copied to place p by at(p)S as follows.

First, the original-expressions are evaluated to give a vector of X10 values. Consider the graph of all values reachable from these values (except for transient fields (§13.3.5, GlobalRefs (§13.3.6); also custom serialization (§13.3.2 may alter this behavior)).

Second this graph is *serialized* into a buffer and transmitted to place q. Third, the vector of X10 values is re-created at q by deserializing the buffer at q. Fourth, S is executed at q, in an environment in which each variable v declared in F refers to the corresponding deserialized value.

Note that since values accessed across an at boundary are copied, the programmer may wish to adopt the discipline that either variables accessed across an at boundary contain only structs or stateless objects, or the methods invoked on them do not access any mutable state on the objects. Otherwise the programmer has to ensure that side effects are made to the correct copy of the object. For this the struct x10.lang.GlobalRef[T] is often useful.

Serialization and deserialization.

The X10 runtime provides a default mechanism for serializing/deserializing an object graph with a given set of roots. This mechanism may be overridden by the programmer on a per class or struct basis as described in the API documentation for x10.io.CustomSerialization. The default mechanism performs a deep copy of the object graph (that is, it copies the object or struct and, recursively, the values contained in its fields), but does not traverse or copy transient fields. transient fields are omitted from the serialized data. On deserialization, transient fields are initialized with their default values (§4.7). The types of transient fields must therefore have default values.

The default serialization/deserialization mechanism will not (modulo error conditions like OutOfMemoryError) throw any exceptions. However, user code running during serialization/deserialization via CustomSerialization may raise exceptions. These exceptions are handled like any other exception raised during the execution of an X10 activity. However, due to the special treatment of at (p) async S (§13.3) any exception raised during deserialization will be handled as if it was raised by async S, not by the at statement itself.

A struct s of type x10.lang.GlobalRef[T] 13.3.6 is serialized as a unique global reference to its contained object o (of type T). Please see the documentation of x10.lang.GlobalRef[T] for more details.

13.3.3 at and Activities

at(p)S does *not* start a new activity. It should be thought of as transporting the current activity to p, running S there, and then transporting it back. async is the only construct

in the language that starts a new activity. In different contexts, each one of the following combination of `async` and `at` makes sense: (1) `at(p) async S` and, (2) `async at(p) S`. In the first case, the expression `p` is evaluated synchronously by the current activity and then a single remote async is spawned. In the second case, `p` is semantically required to be evaluated asynchronously with the parent async as it is contained in the body of an async. Then the evaluation of `S` is transported to the new place. In most cases, the first form (`at(p) async S`) is preferred to second one (`async at(p) S`), since it enables a more efficient runtime implementation (it avoids the spawning a local async solely to evaluate `p`).

Since `at(p) S` does not start a new activity, `S` may contain constructs which only make sense within a single activity. For example,

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

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

Limitation: X10 does not currently allow `break`, `continue`, or `return` to exit from an `at`.

13.3.4 Copying from `at`

`at(p)S` copies data required in `S`, and sends it to place `p`, before executing `S` there. The only things that are not copied are values only reachable through `GlobalRef`s and `transient` fields, and data omitted by custom serialization.

Example:

```
val c = new Cell[Long](9); // (1)
at (here) {                // (2)
   assert(c() == 9);       // (3)
   c.set(8);               // (4)
   assert(c() == 8);       // (5)
}
assert(c() == 9);          // (6)
```

The `at` *statement copies the* `Cell` *and its contents. After* `(1)`*,* `c` *is a* `Cell` *containing 9; call that cell* c_1 *At* `(2)`*, that cell is copied, resulting in another cell* c_2 *whose contents are also 9, as tested at* `(3)`*. (Note that the copying behavior of* `at` *happens* even when the destination place is the same as the starting place— *even with* `at(here)`*.) At* `(4)`*, the contents of* c_2 *are changed to 8, as confirmed at* `(5)`*; the contents of* c_1 *are of course untouched. Finally, at* `(c)`*, outside the scope of the* `at` *started at line* `(2)`*,* `c` *refers to its original value* c_1 *rather than the copy* c_2*.*

The `at` statement induces a *deep copy*. Not only does it copy the values of variables, it copies values that they refer to through zero or more levels of reference. Structures are