Report on the Experimental Language X10

DRAFT v 0.404: Please do not cite.
Please send comments to Vijay Saraswat at vsaraswa@us.ibm.com
(IBM Confidential)

February 7, 2005

SUMMARY

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

X10 is based on state-of-the-art object-oriented programming languages and deviates from them only as necessary to support its design goals. The language is intended to have a simple and clear semantics and be readily accessible to mainstream object-oriented programmers. It is intended to support a wide variety of concurrent programming idioms, incuding data parallelism, task parallelism, pipelining, producer/consumer and divide and conquer.

This document provides an initial description of the language and corresponds to the first implementation of the language. We expect to revise this document in several months in the light of experience gained in implementing and using this language.

The X10 design team consists of David Bacon, Bob Blainey, Philippe Charles, Perry Cheng, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Patrick Gallop, Christian Grothoff, Allan Kielstra, Robert O'callahan, Filip Pizlo, V.T. Rajan, Vijay Saraswat (contact author), Vivek Sarkar, Christoph von Praun and Jan Vitek.

For extended discussions and support we would like to thank: Calin Cascaval, Elmootaz Elnozahy, Orren Krieger, John McCalpin, Paul McKenney, Ram Rajamony.

We also thank Jonathan Rhees and William Clinger with help in obtaining the LaTeX style file and macros used in producing the Scheme report, after which this document is based. We also acknowledge the influence of the Java Language Specification [6] document, as evidenced by the numerous citations in the text.

This document is a revision of the 0.32 version of the Report, released on 17 July 2004. It documents the language corresponding to the first version of the implementation. The implementation was done by Philippe Charles, Christopher Donawa, Christian Grothoff, Vijay Saraswat, Christoph von Praun, and Kemal Ebcioğlu.

CONTENTS

Introduction . . . 2
1 Overview of X10 . . . 3
1.1 Places and activities . . . 3
1.2 Clocks . . . 3
1.3 Interfaces and Classes . . . 4
1.4 Scalar classes . . . 4
1.5 Arrays, Regions and Distributions . . . 4
1.6 Nullable type constructor . . . 4
1.7 Statements and expressions . . . 4
1.8 Translating MPI programs to X10 . . . 4
1.9 Summary and future work . . . 5
2 Lexical structure . . . 5
3 Types . . . 6
3.1 Type constructors . . . 6
3.2 The Nullable Type Constructor . . . 7
3.3 Future type constructor . . . 8
3.4 Array Type Constructors . . . 8
3.5 Place types . . . 8
3.6 Variables . . . 10
3.7 Objects . . . 11
3.8 Built-in types . . . 11
3.9 Conversions and Promotions . . . 11
4 Names and packages . . . 11
5 Places . . . 12
6 Activities . . . 12
6.1 The X10 rooted exception model . . . 13
6.2 Spawning an activity . . . 13
6.3 Finish . . . 14
6.4 Initial activity . . . 14
6.5 Asynchronous Expression and Futures . . . 14
6.6 Atomic sections . . . 15
6.7 Remote Method Invocation . . . 17
6.8 Iteration . . . 17
7 Clocks . . . 18
7.1 Clock operations . . . 18
8 Interfaces . . . 20
9 Classes . . . 20
9.1 Reference classes . . . 20
9.2 Value classes . . . 20
9.3 Method annotations . . . 21
10 Arrays . . . 21
10.1 Regions . . . 22
10.2 Distributions . . . 22
10.3 Array initializer . . . 23
10.4 Operations on arrays . . . 24
11 Statements and Expressions . . . 25
11.1 Assignment . . . 25
11.2 Point and region construction . . . 25
11.3 Exploded variable declarations . . . 25
11.4 Expressions . . . 26
11.5 Linking with native code . . . 26
Example . . . 27
X10 syntax . . . 28
Changes from v0.32 . . . 37
References . . . 37
Alphabetic index of definitions of concepts, keywords, and procedures . . . 38

INTRODUCTION

Background

Bigger computational problems need bigger computers capable of performing a larger number of operations per second. The era of increasing performance by simply increasing clocking frequency now seems to be behind us; faster chips run hotter and current cooling technology does not scale as rapidly as the clock. Instead, computer designers are starting to look at *scale out* systems in which the system's computational capacity is increased by adding additional nodes of comparable power to existing nodes, and connecting nodes with a high-speed communication network.

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

One response to this problem has been to move to a *fragmented memory model*. Multiple processors – each sequentially consistent internally – are made to interact via a relatively language-neutral message-passing format such as MPI [9]. This model has enjoyed some success: several high-performance applications have been written in this style. Unfortunately, this model leads to a *loss of programmer productivity*: the mesage-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code (in an SPMD fashion) etc.

One response to this problem has been the advent of the *partitioned global address space* (GAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [2, 10]. These languages permit the programmer to think of a single computation running across multiple processors, sharing a common address space. All data resides at some processors, which is said to have *affinity* to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global *barriers* are used to ensure that processors remain roughly in lock-step.

X10 is a modern object-oriented programming language in the GAS family. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end computers – for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads. X10 is based on state-of-the-art object-oriented programming ideas primarily to take advantage of their proven flexibility and ease-of-use for a wide spectrum of programming problems. X10 takes advantage of several years of research (e.g. in the context of the Java^{TM} Grande forum, [8, 1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *value types* (such as `int`, `float`, `complex` etc), supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [3]) and supports IEEE-standard floating point arithmetic. Some limited operator overloading is provided for a few "built in" classes in the `x10.lang` package. Future versions of the language will support user-definable operator overloading.

The major novel contribution of X10 however is its flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the GAS model to the *globally asynchronous, locally synchronous* (GALS) model originally developed in hardware and embedded software research. X10 introduces *places* as an abstraction for a locally synchronous computational context. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more *activities*. Activities are extremely lightweight threads of execution. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. It *must* spawn activities *asynchronously* to access or update memory at other places. X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale. *Immutable* data needs no consistency management and may be freely copied between places. An attempt to read an uninitialized immutable location suspends until the location is written into, thus permitting data-flow synchronization. One or more *clocks* may be used to order activities running in multiple places. (Multi-dimensional) Arrays may be distributed across multiple places. Arrays support parallel collective operations. A novel exception flow model ensures that exceptions thrown by asynchronous activities can be caught at a suitable parent activity. The type system statically tracks which locations may reside at the same place. Linking with native code is supported.

X10 is an experimental language. Several representative concurrent idioms have already found pleasant expression in X10. We intend to develop several full-scale applications to get better experience with the language, and revisit the design in the light of this experience. Future versions of the language are expected to support user-definable operators and permit the specification of generic classes and methods.

DESCRIPTION OF THE LANGUAGE

1. Overview of X10

X10 may be thought of as (generic) Java less concurrency, arrays and built-in types, plus *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *value* types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing.

1.1. Places and activities

The central new concept in X10 is that of a *place* (§ 5). A place may be thought of conceptually as a "virtual shared-memory multi-processor": a computational unit with a finite, though perhaps dynamically varying, number of hardware threads and a bounded amount of shared memory uniformly accessible by all threads. An X10 program is intended to run on a computer capable of supporting millions of places.

An X10 computation acts on *data objects*(§ 3.7) through the execution of lightweight threads called *activities*(§ 6). Objects are of two kinds. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place and stays at that place throughout its lifetime. An *aggregate* object has many fields (the number may be known only when the object is created), uniformly accessed through an index (e.g. an integer) and may be distributed across many places. The distribution of an aggregate object remains unchanged throughout the computation. X10 assumes an underlying garbage collector will dispose of (scalar and aggregate) objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation. (There are no operations in the language to allow a programmer to explicitly release memory.)

X10 has a *unified* or *global address space*. This means that an activity can reference objects at other places. However, an activity may synchronously access data items only in the current place (the place in which the activity is running). It may atomically update one or more data items, but only in the current place. Indeed, all accesses to mutable shared data must occur from within an *atomic section*. To read a remote location, an activity must spawn another activitiy *asynchronously* (§ 6.2). This operation returns immediately, leaving the spawning activity with a *future* (§ 6.5) for the result. Similarly, remote location can be written into only by asynchronously spawning an activity to run at that location.

Throughout its lifetime an activity executes at the same place. An activity may dynamically spawn activities in the current or remote places.

Atomic sections X10 introduces statements of the form `atomic S` where `S` is a statement. The type system ensures that such a statement will dynamically access only local data. (The statement may throw a `BadPlaceException` – but only because of a failed place cast.) Such a statement is executed by the activity as if in a single step during which all other activities are frozen.

Asynch activities An asynch activity is a statement of the form `async (P) S` where `P` is a place expression and `S` is a statement. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`.

An async expression of type `future<T>` has the form `future (P) E` where `E` is an expression of type `T`. It executess the expression `E` at the place `P` as an async activity, immediately returning with a future. The future may later be forced causing the activity to be blocked until the return value has been computed by the async activity.

1.2. Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic, asynchronous network of activities in an X10 computation. Instead, it becomes necessary to allow a computation to use multiple barriers. X10 *clocks* (§ 7) are designed to offer the functionality of multiple barriers in a dynamic context while still supporting determinate, deadlock-free parallel computation.

Activities may use clocks to repeatedly detect quiescence of arbitrary programmer-specified, data-dependent set of activities. Each activity is spawned with a known set of clocks and may dynamically create new clocks. At any given time an activity is *registered* with zero or more clocks. It may register newly created activities with a clock, un-register itself with a clock, suspend on a clock or require that a statement (possibly involving execution of new async activities) be executed to completion before the clock can advance. At any given step of the execution a clock is in a given phase. It advances to the next phase only when all its registered activities have *quiesced* (by executing a `continue` operation on the clock), and all statements scheduled for execution in this phase have terminated. When a clock advances, all its activities may now resume execution.

Thus clocks act as *barriers* for a dynamically varying collection of activities. They generalize the barriers found in MPI style program in that an activity may use multiple clocks simultaneously. Yet programs using clocks are guaranteed not to suffer from deadlock. Clocks are also integrated into the X10 type system, permitting variables

to be declared so that they are `final` in each phase of a clock.

1.3. Interfaces and Classes

Programmers write X10 code by writing *generic interfaces* (§ 8) and *generic classes* (§ 9). Generic interfaces and classes may be defined over a collection of *type parameters*. Instances can be created only from *concrete* classes; such a class has all its type parameters (if any) instantiated with concrete classes and concrete interfaces.

1.4. Scalar classes

An X10 scalar class (§ 9) has fields, methods and inner types (interfaces, classes), subclasses another class, and implements one or more interfaces. Thus X10 classes live in a single-inheritance code hierarchy.

There are two kinds of scalar classes: *reference* classes (§ 9.1) and *value* classes (§ 9.2).

A reference class typically has updatable fields. Objects of such a class may not be freely copied from place to place. Methods may be invoked on such an object only by an activity in the same place.

A value class (§ 9.2) has no updatable fields (defined directly or through inheritance), and allows no reference subclasses. (Fields may be typed at reference classes, so may contain references to objects with mutable state.) Objects of such a class may be freely copied from place to place, and may be implemented very efficiently. Methods may be invoked on such an object from any place.

X10 has no primitive classes. However, the standard library `x10.lang` supplies (final) value classes `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `complex` and `String`. The user may defined additional arithmetic value classes using the facilities of the language.

1.5. Arrays, Regions and Distributions

An X10 array is a function from a *distribution* (§ 10.2) to a base type (which may itself be an array type).

A distribution is a map from a *region* (§ 10.1) to a subset of places. A region is a collection of *points* or *indices*. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. Points are used in array index expressions to pick out a particular array element.

Operations are provided to construct regions from other regions, and to iterate over regions. Standard set operations, such as union, disjunction and set difference are available for regions.

A primitive set of distributions is provided, together with operations on distributions. A *sub-distribution* of a distribution is one which is defined on a smaller region and agrees with the distribution at all points. The standard operations on regions are extended to distributions.

In future versions of the language, a programmer may specify new distributions, and new operations on distributions.

A new array can be created by restricting an existing array to a sub-distribution, by combining multiple arrays, and by performing pointwise operations on arrays with the same distribution.

X10 allows array constructors to iterate over the underlying distribution and specify a value at each item in the underlying region. Such a constructor may spawn activities at multiple places.

1.6. Nullable type constructor

X10 has a `nullable` type constructor which can be applied uniformly to scalar (value or reference) and array types. This type constructor returns a new type which adds a special value `null` to the set of values of its argument type, unless the argument type already has this value.

1.7. Statements and expressions

X10 supports the standard set of primitive operations (assignment, classcasts) and sequential control constructs (conditionals, looping, method invocation, exception raising/catching) etc.

Place casts The programmer may use the standard classcast mechanism (§ 11.4) to cast a value to a located type. A `BadPlaceException` is thrown if the value is not of the given type. This is the only language construct that throws a `BadPlaceException`.

1.8. Translating MPI programs to X10

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

Each separate MPI process can be translated into an X10 place. Async activities may be used to read and write variables located at different processes. A single clock may be used for barrier synchronization between multiple MPI

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

1.9. Summary and future work

1.9.1. Design for scalability

X10 is designed for scalability. An activity may atomically access only multiple locations in the current place. Unconditional atomic sections are statically guaranteed to be non-blocking, and may be implemented using non-blocking techniques that avoid mutual exclusion bottlenecks. Dataflow synchronization permits point-to-point coordination between reader/writer activities, obviating the need for barrier-based or lock-based synchronization in many cases.

1.9.2. Design for productivity

X10 is designed for productivity.

Safety and correctness. Programs written in X10 are guaranteed to be statically *type safe*, *memory safe* and *pointer safe*. Static type safety guarantees that at runtime a location contains only those values who's dynamic type satisfies the constraints imposed by the location's static type and every runtime operation performed on the value in a location is permitted by the static type of the location.

Memory safety guarantees that an object may only access memory within its representation, and other objects it has a reference to. X10 supports no pointer arithmetic, and bound-checks array accesses dynamically if necessary. X10 uses dynamic garbage collection to collect objects no longer referenced by the computation. X10 guarantees that no object can retain a reference to an object whose memory has been reclaimed. Further, X10 guarantees that every location is initialized at runtime before it is read, and every value read from a location has previously been written into that location.

Pointer safety guarantees that a null pointer exception cannot be thrown by an operation on a value of a non-nullable type.

Because places are reflected in the type system, static type safety also implies *place safety*: a location may contain references to only those objects whose location satisfies the restrictions of the static place type of the location.

X10 programs that use only clocks and unconditional atomic sections are guaranteed not to deadlock. Unconditional atomic sections are non-blocking, hence cannot introduce deadlocks (assuming the implementation is correct).

Many concurrent programs can be shown to be determinate (hence race-free) statically.

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

We believe that X10, like Java, will be able to support a large number of libraries and tools. An area where we expect future versions of X10 to improve on Java like languages is *native integration* (§ 11.5). Specifically, X10 will permit permit multi-dimensional local arrays to be operated on natively by native code.

1.9.3. Conclusion

X10 is considerably higher-level than thread-based languages in that it supports dynamically spawning very lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection.

Yet it is much more concrete than languages like HPF in that it forces the programmer to explicitly deal with distribution of data objects. In this the language reflects the designers belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent.[1] At the same time we believe that the place-based type system and support for generic programming will allow the X10 programmer to be highly productive; many of the tedious details of distribution-specific code can be handled in a generic fashion.

We expect the next version of the language to be significantly informed by experience in implementing and using the language. We expect it to have constructs to support continuous program optimization, and allow the programmer to provide guidance on clustering places to (hardware) nodes. For instance, we may introduce a notion of hierarchical clustering of places.

2. Lexical structure

In general, X10 follows Java rules [6, Chapter 3] for lexical structure.

[1] In this X10 is similar to more modern languages such as ZPL [3].

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators.

Whitespace Whitespace follows Java rules [6, Chapter 3.6]. ASCII space, horizontal tab (HT), form feed (FF) and line terminators constitute white space.

Comments Comments follows Java rules [6, Chapter 3.7]. All text included within the ASCII characters "/*" and "*/" is considered a comment and ignored. All text from the ASCII character "//" to the end of line is considered a comment and ignored.

Identifiers Identifiers are defined as in Java.

Keywords X10 reserves the following keywords from Java:

```
abstract   break  case       catch
class      const  continue   default
do         else   extends    final
finally    for    goto       if
implements import instanceof interface
native     new    package    private
protected  public return     static
super      switch this       throw
throws     try    void       while
```

(Note that the primitive types are no longer considered keywords.)

X10 introduces the following keywords:

```
activitylocal async     ateach     atomic
await         clocked   current    foreach
finish        future    here       next
nullable      or        placelocal reference
value        when
```

Literals *Note:* We have to figure out the syntax for literals, since we do not wish to build knowledge of any type into the language. For now, assume Java style literals.

Separators X10 has the following separators:

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

Operators X10 has the following operators:

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

3. Types

X10 is a *strongly typed* object language: every variable and expression has a type that is known at compile-time. Further, X10 has a *unified* type system: all data items created at runtime are *objects* (§ 3.7. Types limit the values that variables can hold, and specify the places at which these values lie.

X10 supports two kinds of objects, *reference objects* and *value objects*. Reference objects are instances of *reference classes* (§ 9.1). They may contain mutable fields and must stay resident in the place in which they were created. Value objects are instances of *value classes* (§ 9.2). They are immutable and may be freely copied from place to place. Either reference or value objects may be *scalar* (instances of a non-array class) or *aggregate* (instances of arrays).

An X10 type is either a *reference type* or a *value type*. Each type consists of a *data type*, which is a set of values, and a *place type* which specifies the place at which the value resides. Types are constructed through the application of *type constructors* (§ 3.1).

Types are used in variable declarations, casts, object creation, array creation, class literals and `instanceof` expressions.[1]

A variable is a storage location (§ 3.6). All variables are initialized with a value and cannot be observed without a value.

Variables whose value may not be changed after initialization are called *final variables* (or sometimes *constants*). The programmer indicates that a variable is final by using the annotation `final` in the variable declaration. Final variables play an important role in X10, as we shall discuss below. For this reason, X10 enforces the lexical restriction that all variables whose name starts with an upper case letter are implicitly declare final. (It is not an error to also explicitly declare such variables as final.)

3.1. Type constructors

An X10 type is a pair specifying a *datatype* and a *placetype*. Semantically, a datatype specifies a set of values and a placetype specifies the set of places at which these values may live. Thus taken together, a type specifies both the kind of value permitted and its location.

```
509   Type ::=  DataType  PlaceTypeSpecifieropt
510     | nullable  Type
511     | future <  Type >
512   DataType ::=  PrimitiveType
513   DataType ::=  ClassOrInterfaceType
514     |  ArrayType
```

For simplicity, this version of X10 does not permit the specification of generic classes or interfaces. This is expected to be remedied in future versions of the language.

Every class and interface definition in X10 defines a type with the same name. Additionally, X10 specifies three *type constructors*: `nullable`, the `future`, and array type constructors. We

[1] In order to allow this version of the language to focus on the core new ideas, X10 v0.4x does not have user-definable classloaders, though there is no technical reason why they could not have been added.

discuss these constructors and place types in detail in the secions that follow; here we briefly discss interface and class declarations.

Interface declarations. An interface declaration specifies a name, a list of extended interfaces, and constants (`public static final` fields) and method signatures associated with the interface. Each interface declaration introduces a type with the same name as the declaration. Semantically, the data type is the set of all objects which are instances of (value or reference) classes that implement the interface. A class implements an interface if it says it does and if it implements all the methods defined in the interface.

The *interface declaration* (§ 8) takes as argument one or more interfaces (the *extended* interfaces), one or more type parameters and the definition of constants and method signatures and the name of the defined interface. Each such declaration introduces a data type.

```
426   DataType ::= ClassOrInterfaceType
433   ClassOrInterfaceType ::= TypeName
13    ClassType ::= TypeName
15    TypeName ::= identifier
16     | TypeName . identifier
```

Reference class declarations. The *reference class declaration* (§ 9.1 takes as argument a reference class (the *extended class*), one or more interfaces (the *implemented interfaces*), the definition of fields, methods and inner types, and returns a class of the named type (§ 9.1). Each such declaration introduces a data type. Semantically, the data type is the set of all objects which are instances of (subclasses of) the class.

Value class declarations. The *value class declaration* (§ 9.2) is similar to the reference class declaration except that it must extend either a value class or a reference class that has no mutable fields. It may be used to construct a value type in the same way as a reference class declaration can be used to construct a reference type.

3.2. The Nullable Type Constructor

X10 supports the prefix type constructor, `nullable`. For any type `T`, the type `nullable T` contains all the values of type `T`, and a special `null` value, unless `T` already contains `null`. This value is designated by the literal `null`, which is special in that it has the type `nullable T` for all types `T`.

The visibility of the type `nullable T` is the same as the visibility of `T`. The members of the type `nullable T` are the same as those of type `T`. Note that because of this `nullable` may not be regarded as a generic class; rather it is a special type constructor.

This type constructor can be used in any type expression used to declare variables (e.g. local variables, method parameters, class fields, iterator parameters, try/catch parameters etc). It may be applied to value types, reference types or aggregate types. It may not be used in an `extends` clause or an `implements` clause in a class or interface declaration. It may not be used in a new expression – a new expression is used to construct

If `T` is a value (respectively, reference) type, then `nullable T` is defined to be a value (respectively, reference) type.

An immediate consequence of the definition of `nullable` is that for any type `T`, the type `nullable nullable T` is equal to the type `nullable T`.

Any attempt to access a field or invoke a method on the value `null` results in a `NullPointerException`.

An expression `e` of type `nullable T` may be checked for nullity using the expression `e==null`. (It is a compile time error for the static type of `e` to not be `nullable T`, for some `T`.)

Conversions `null` can be passed as an argument to a method call whose corresponding formal parameter is of type `nullable T` for some type `T`. (This is a widening reference conversion, per [6, Sec 5.1.4].) Similarly it may be returned from a method call of return type `nullable T` for some type `T`.

For any value `v` of type `T`, the class cast expression `(nullable T) v` succeeds and specifies a value of type `nullable T` that may be seen as the "boxed" version of `v`.

X10 permits the widening reference conversion from any type `T` to the type `nullable T1` if `T` can be widened to the type `T1`. Thus, the type `T` is a subtype of the type `nullable T`.

Correspondingly, a value `e` of type `nullable T` can be cast to the type `T`, resulting in a `NullPointerException` if `e` is `null` and `nullable T` is not equal to `T`, and in the corresponding value of type `T` otherwise. If `T` is a value type this may be seen as the "unboxing" operator.

The expression `(T) null` throws a `ClassCastException` if `T` is not equal to `nullable T`; otherwise it returns `null` at type `T`. Thus it may be used to check whether `T=nullable T`.

Arrays of nullary type The nullary type constructor may also be used in (aggregate) instance creation expressions (e.g. `new (nullable T)[R]`). In such a case `T` must designate a class. Each member of the array is initialized to `null`, unless an explicit array initializer is specified.

Implementation notes A value of type `nullable T` may be implemented by boxing a value of type `T` unless the value is already boxed. The literal `null` may be represented as the unique null reference.

Java **compatibility** Java provides a somewhat different treatment of `null`. A class definition extends a nullable type to produce a nullable type, whereas primitive types such as `int` are not nullable — the programmer has to explicitly use a boxed version of `int`, `Integer`, to get the effect of `nullable int`. Wherever Java uses a variable at reference type `S`, and at runtime the variable may carry the value `null`, the X10 programmer should declare the variable at type `nullable S`. However, there are many situations in Java in which a variable at reference type `S` can be statically determined to not carry null as a value. Such variables should be declared at type `S` in X10

Design rationale The need for `nullable` arose because X10 has value types and reference types, and arguably the ability to add a `null` value to a type is orthogonal to whether the type is a value type or a reference type. This argues for the notion of nullability as a type constructor.

The key question that remains is whether it should be possible to define "towers", that is, define the type constructor in such a way that `nullable nullable T` is distinct from `nullable T`. Here one would think of nullable as a disjoint sum type constructor that adds a value `null` to the interpretation of its argument type even if it already has that value. Thus `nullable nullable T` is distinct from `nullable T` because it has one more `null` value. Explicit injection/projection functions (of signature `T -> nullable T` to `nullable T ->T`) would need to be provided.

The designers of X10 felt that while such a definition might be mathematically tenable, and programmatically interesting, it was likely to be too confusing for programmers. More importantly, it would be a deviation from current practice that is not forced by the core focus of X10 (concurrency and distribution). Hence the decision to collapse the tower. As discussed below, this results in no loss of expressiveness because towers can be obtained through explicit programming.

Examples Consider the following class:

```
final value Box {
  public nullable Object datum;
  public Box(nullable Object v) { this.datum = v; }
}
```

Now one may use a variable `x` at type `nullable Box` to distinguish between the `null` at type `nullable Box` and at type `nullable Object`. In the first case the value of `x` will be `null`, in the second case the value of `x.datum` will be `null`.

Such a type may be used to define efficient code for memoization:

```
abstract class Memo  {
  (nullable Box)[]  values;
  Memo(int n) {
    // initialized to all nulls
    values = new (nullable Box)[n];
  }
  nullable Object compute(int key);
  nullable Object lookup(int key) {
   if (values[key] != null)
     return values[key].datum;
   V val = compute(key);
   values[key] = new Box(val);
   return val;
  }
}
```

3.3. Future type constructor

3.4. Array Type Constructors

X10 v0.4x does not have array class declarations (§ 10). This means that user cannot define new array class types. Instead arrays are created as instances of array types constructed through the application of *array type constructors* (§ 10).

The array type constructor takes as argument a type (the *base type*), an optional distribution (§ 10.2), and optionally either the keyword `reference` or `value` (the default is reference):

```
18    ArrayType ::= Type [ ]
438   ArrayType ::= X10ArrayType
439   X10ArrayType ::= Type [ . ]
440     | Type reference [ . ]
441     | Type value [ . ]
442     | Type [ DepParameterExpr ]
443     | Type reference [ DepParameterExpr ]
444     | Type value [ DepParameterExpr ]
```

The array type `Type[ ]` is the type of all one-dimensional locally distributed arrays of basetype `Type` whose underlying region is of the form `0:N`, for some `N`.

The qualifier `value` (`reference`) specifies that the array is a `value`(`reference`) array. The array elements of a `value` array are all `final`.[2]If the qualifier is not specified, the array is a `reference` array.

The array type `Type reference [.]` is the type of all (reference) arrays of basetype `Type`. Such an array can take on any distribution, over any region. Similarly, `Type value [.]` is the type of all value arrays of basetype `Type`.

X10 v0.4x also allows a distribution to be specified between `[` and `]`. The distribution must be an expression of type `distribution` (e.g. a `final` variable) whose value does not depend on the value of any mutable variable.

Future extensions to X10 will support a more general syntax for arrays which allows for the specification of dependent types, e.g. `double[:rank 3]`, the type of all arrays of `double` of rank `3`.

3.5. Place types

Recall that an X10 computation spans multiple places (§ 5). Each place constains data and activities that operate on that data. X10 v0.4x does not permit the dynamic creation of a place. Each X10 computation is initiated with a fixed number of places, as determined by a configuration parameter. In this section we discuss how the programmer may supply place type information, thereby allowing the compiler to check data locality, i.e. that data items being accessed in an atomic section are local.

```
422   Type ::= DataType PlaceTypeSpecifieropt
428   PlaceTypeSpecifier ::= AT PlaceType
429   PlaceType ::= ?
431     | current
432     | PlaceExpression
```

[2]Note that the base type of a `value` array can be a value class or a reference class, just as the type of a `final` variable can be a value class or a reference class.

The anywhere place type The simplest piece of place information that a programmer can specify is no information at all. The type `F@?` expresses this: it is satisfied by an object of datatype `F` located at any place. This type is particularly useful in specifying collections of objects at heterogenous places. For intance the type `F@?[.]` is the type of all distributed arrays whose elements can contain references to objects of datatype `F` located at any place. Thus this type specifies no static relationship between the places of two different array elements.

A datatype `F` occurring in a place where a type is expected is always taken as shorthand for `F@?`. This is compatible with the Principle of Least Disclosure: the programmer may quickly get his code running without specifying where objects are located. The compiler will insert dynamic runtime checks to ensure that locality conditions are satisfied within atomic sections. As the programmer discovers more information about the locality of data elements s/he may supply more refined place type information.

Place expression in place types. The programmer may use the indexical place expression `here` as a place type. Recall that in any runtime context, `here` stands for the place of the current activity (§ 5.0.2). `here` cannot be used to specify the placetype of any object field because the type of a field cannot vary with the activity accessing the field.

Consider the example:

```
public class F {
  int datum;
  int m(F@here a, F@here b) {
    return a.datum + b.datum;
 }
}
```

This code satisfies X10's typing rules. Recall that an object's fields can be accessed only if the object is located in the same place as the activity. Hence in the body of the method `m`, the compiler must check that `a` and `b` are statically known to be local. But this follows from the type declarations of the method parameters.

The programmer may also specify the place information `P` for any final variable `P` of type place. This may allow the compiler to check locality conditions for remote activities at place `P`. For example:

```
public class F {
  int datum;
  place P;
  public F(place P) {
    this.P = P;
  }
  future<int> m(final F@P a, final F@P b) {
    return future(P) a.datum + b.datum;
  }
}
```

In this class the constructor for `F` takes a place argument and stores it in a final field. This field may be used in types in the body of the class, e.g. to specify the types of method parameters, as stated above. The body of the method `m` typechecks because the compiler can verify that at the place `P` the values contained in the variables `a` and `b` are local, hence it is legal to access their fields.

An example:

In many circumstances it becomes necessary to use final variables of type `place` as X10 distinguishes two kinds of places: *shared places*, which correspond to the memory of individual processors which is shared across multiple activities, and *scoped places* such as threadlocal memory and method memory which is available only in limited scope. X10 v0.4x supports only `threadlocal` scoped memory, i.e. memory accessible only to the current activity. Future versions may support `methodlocal` and `blocklocal` memory.

Future work In future versions, X10 will support value and type parameters for classes, interfaces and methods. This will permit code of the form:

```
public class F {
  int datum;
  <P instance place> future<int> m(F@P a, F@P b) {
    return future(P) a.datum + b.datum;
  }
}
```

The class `F` specifies a generic method `m` which requires that its two arguments must be of data type `F` and must both reside at the same unknown place `P`. Now the body of the `future` expression can be type-checked by the compiler, without knowing the identity of the place `P`.

Similarly in the body of a generic class with a place parameter `P` one may construct the type `F@P[.]` to represent the type of distributed arrays each of whose elements must contain a reference to an object of datatype `F` located at the place `P`.

An object can be cast to a particular place. If the object is not at the right place, a `BadPlaceException` is thrown:

```
m(Object obj) { // accept an object at any place
     // Cast it to here. This may throw
     // an exception if the object is not local.
     Object@here h = (Object@here)obj;
     // The object is local, invoke a
     // method on it synchronously.
     h.m1();
 }
```

Places can be checked for equality. One can view this as the analog of the `instanceof` operator for places.

```
m(Object obj) {
    if (here == obj.location) {
       // will never throw an exception
      Object@here h = (Object@here)obj;
      h.m();
   }
}
```

`current` Similarly the object reference type `current` can be used in (i) constructing a distribution for a reference array or (ii) specifying the location of the base type of a reference array. In the first case points mapped to `current` by the distribution will reside in the same place as the array object itself, and in the second case the value of the array at a particular point is an object in the same place as that array component. Example:

```
Object@current[] objects;
```

X10 permits user-definable place constants (= final variables = variables that must be assigned before use, and can be assigned only once). Place constants may be used in type expressions after the `@` sign. For instance, consider the following class definition:

```
public class Cell {
 Object@P value;

 public Cell( Object@P value ) {
    this.value = value;
 }

 public Object@P getValue() {
    return this.value;
 }

 public void setValue( Object@P value ) {
    this.value = value;
 }
}
```

This class may be used thus:

```
Cell cell =
   new Cell(new Point@Q());
```

3.6. Variables

A variable of a reference data type `reference R` where `R` is the name of an interface (possibly with type arguments) always holds a reference to an instance of a class implementing the interface `R`.

A variable of a reference data type `R` where `R` is the name of a reference class (possibly with type arguments) always holds a reference to an instance of the class `R` or a class that is a subclass of `R`.

A variable of a reference array data type `R [D]` is always an array which has as many variables as the size of the region underlying the distribution `D`. These variables are distributed across places as specified by `D` and have the type `R`.

A variable of a nullary (reference or value) data type `?T` always holds either the value (named by) `null` or a value of type `T` (these cases are not mutually exclusive).

A variable of a value data type `value R` where `R` is the name of an interface (possibly with type arguments) always holds either a reference to an instance of a class implementing `R` or an instance of a class implementing `R`. No program can distinguish between the two cases.

A variable of a value data type `R` where `R` is the name of a value class (possibly with type arguments) always holds a reference to an instance of `R` (or a class that is a subclass of `R`) or an instance of `R` (or a class that is a subclass of `R`). No program can distinguish between the two cases.

A variable of a value array data type `V value [R]` is always an array which has as many variables as the size of the region `R`. Each of these variables is immutable and has the type `V`.

X10 supports seven kinds of variables: final *class variables* (static variables), *instance variables* (the instance fields of a class), *array components*, *method parameters*, *constructor parameters*, *exception-handler parameters* and *local variables*.

3.6.1. Final variables

A final variable satisfies two conditions:

- it can be assigned to at most once,
- it must be assigned to before use.

X10 follows Java language rules in this respect [6, §4.5.4,8.3.1.2,16]. Briefly, the compiler must undertake a specific analysis to statically guarantee the two properties above.

3.6.2. Initial values of variables

Every variable declared at a type must always contain a value of that type.

Every class variable, instance variable or array component variable is initialized with a default value when it is created. A variable declared at a nullary type is always initialized with `null`. For a variable declared at a scalar class type it must be the case that a nullary constructor for that class is visible at the site of the declaration; the variable is initialized with the value returned by invoking this constructor. For a variable declared at an array type it must be the case that the base type is either nullable or a class type with a nullary constructor visible at the site of the declaration. The variable is then initialized with an array defined over the smallest region and default distribution consistent with its declaration and with each component of the array initialized to `null` or the result of invoking the nullary constructor.

Each method and constructor parameter is initialized to the corresponding argument value provided by the invoker of the method. An exception-handling parameter is initialized to the object thrown by the exception. A local variable must be explicitly given a value by initialization or assignment, in a way that the compiler can verify using the rules for definite assignment [6, § 16].

Each class `C` has an explicitly or implicitly defined nullary constructor. If `C` does not have an explicit nullary constructor, it is a compile-time error if the class has a field at (a) a scalar type that is a `class` whose nullary constructor is not visible in `C` or is an `interface`, or (b) at an array type whose base type is a `class` whose nullary constructor is not visible in `C` or is an `interface`.

Otherwise a `public` nullary constructor is created by default. This constructor initializes each field of the class (if any) as if it were a variable of the declared type of the field, as described above.

3.7. Objects

An object is an instance of a scalar class or an array type. It is created by using a class instance creation expression (§ 11.4) or an array creation (§ 10.3) expression, such as an array initializer. An object that is an instance of a reference (value) type is called a *reference* (*value*) *object*.

All value and reference classes subclass from `x10.lang.Object`. This class has one `const` field `location` of type `x10.lang.place`. Thus all objects in X10 are located (have a place). However, X10 permits value objects to be freely copied from place to place because they contain no mutable state. It is permissible for a read of the `location` field of such a value to always return `here` (§ 5.0.2); therefore no space needs to be allocated in the object representation for such a field.

In X10 v0.4x a reference object stays resident at the place at which it was created for its entire lifetime.

X10 has no operation to dispose of a reference. Instead the collection of all objects across all places is globally garbage collected.

X10 objects do not have any synchronization information (e.g. a lock) associated with them. Thus the methods on `java.lang.Object` for waiting/synchronizing/notification etc are not available in X10. Instead the programmer should use atomic sections (S 6.6) for mutual exclusion and clocks (S 7) for sequencing multiple parallel operations.

A reference object may have many references, stored in fields of objects or components of arrays. A change to an object made through one reference is visible through another reference. X10 mandates that all accesses to mutable objects shared between multiple activities must occur in an atomic section (§6.6).

Note that the creation of a remote async activity (§ 6.2) `A` at `P` may cause the automatic creation of references to remote objects at `P`. (A reference to a remote object is called a *remote object reference*, to a local object a *local object reference*.) For instance `A` may be created with a reference to an object at `P` held in a variable referenced by the statement in `A`. Similarly the return of a value by a `future` may cause the automatic creation of a remote object reference, incurring some communication cost. An X10 implementation should try to ensure that the creation of a second or subsequent reference to the same remote object at a given place does not incur any (additional) communication cost.

A reference to an object may carry with it the values of final fields of the object. The programmer is guaranteed that the implementation will not incur the cost of communicating the values of final fields of an object from the place where it is hosted to any other place more than once for each target place, even for reference objects.

X10 does not have an operation (such as Pascal's "dereference" operation) which returns an object given a reference to the object. Rather, most operations on object references are transparently performed on the bound object, as indicated below. The operations on objects and object references include:

- Field access (§ 11.4). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely from place to place). In this case the cost of cost of copying the field from the place where the object was created to the referencing place will be incurred at most once per referencing place, according to the rule for final fields discussed above.
- Method invocation (§ 11.4). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely). The X10 implementation must guarantee that the cost of copying enough relevant state of the value object to enable this method invocation to succeed is incurred at most once for each value object per place.
- Casting (§ 11.4). An activity can perform this operation on local or remote objects, and does not incur any communication cost.
- `instanceof` operator (§ 11.4). An activity can perform this operation on local or remote objects, and does not incur any communication cost.
- The stable equality operator `==` and `!=` (§ 11.4). An activity can perform these operations on local or remote objects, and does not incur any communication cost.

3.8. Built-in types

The package `x10.lang` provides a number of built-in class and interface declarations that can be used to construct types.

For instance several value types are provided that encapsulate abstractions (such as fixed point and floating point arithmetic) commonly implemented in hardware by modern computers.

Please consult [5] for more details.

3.9. Conversions and Promotions

X10 v0.4x supports Java's conversions and promotions (identity, widening, narrowing, value set, assignment, method invocation, string, casting conversions and numeric promotions) appropriately modified to support X10's built-in numeric classes rather than Java's primitive numeric types.

This decision may be revisited in future version of the language in favor of a streamlined proposal for allowing user-defined specification of conversions and promotions for value types, as part of the syntax for user-defined operators.

Built-in interfaces:

```
Field            FixedPoint          FloatingPoint
SignedFixedPoint UnsignedFixedPoint
```

Built-in reference types:

```
Object Reference
```

Built-in value types:

```
boolean byte          char  complex<Field>
double  doubledouble  float int
long    longlong      short string
ubyte   ushort        value

place   distribution  region clock
```

Figure 3.1: The contents of the `x10.lang` package.

4. Names and packages

X10 supports Java's mechanisms for names and packages [6, §6,§7], including `public`, `protected`, `private` and package-specific access control.

X10 supports the following naming conventions. Names of `value` classes should start with a lower-case leter, and those of `reference` classes with an upper-case letter. X10 also supports the convention that fields, local variable names and method parameter names that start with an uppercase letter are automatically considered to be annotated with `final`.

5. Places

An X10 place is a repository for data and activities. Each place is to be thought of as a locality boundary: 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.

In X10 v0.4x, the set of places available to a computation is determined at the time that the program is run and remains fixed through the run of the program. The number of places available to a computation may be determined by querying a run-time int constant (`place.MAX_PLACES`).

All scalar objects created during program execution are located in one place, though they may be accessed from other places. Aggregate objects (arrays) may be distributed across multiple places using distributions.

The set of all places in a running instance of an X10 program may be obtained through the `const` field `place.places`. (This set may be used to define distributions, for instance, § 10.2.)

The set of all places is totally ordered. Places may be used as keys in hash-tables.

5.0.1. Place expressions

Any expression of type `place` is called a place expression. Examples of place expressions are `this.location` (the place at which the current object lives), `place.FIRST_PLACE` (the first place in the system in canonical order).

Place expressions are used in the following contexts:

- As a place type in a type (§ 3.5).
- As a target for an `async` activity or a future (§ 6.2).
- In a class cast expression (§ 11.4).
- In an `instanceof` expression (§ 11.4).
- In stable equality comparisons, at type `place`.

Like values of any other type, places may be passed as arguments to methods, returned from methods, stored in fields etc.

5.0.2. here

X10 supports a special indexical constant `here`:

```
22 ExpressionName ::= here
```

The constant evaluates to the place at which the current activity is running. Unlike other place expressions, this constant cannot be used as the placetype of fields, since the type of a field should be independent of the activity accessing it.

Example. The code:

```
public class F {
   public void m( F a ) {
     place OldHere = here;
     async ( a ) {
      System.out.println("OldHere == here:"
                         + (OldHere == here));
     }
   }
  public static void main(String[] s) {
     new F().m(future( place.FIRST_PLACE.next())
                { new F()}.force());
  }
}
```

will print out `true` iff the computation was configured to start with the number of places set to `1`.

6. Activities

An X10 computation may have many concurrent *activities* "in flight" at any give time. We use the term activity to denote a dynamic execution instance of a piece of code (with references to data). An activity is intended to execute in parallel with other activities. An activity may be thought of as a very light-weight thread. In X10 v0.4x, activities may not be interrupted, suspended or resumed from outside.

An activity may asynchronously and in parallel launch activities at other places. An activity is spawned in a given place and stays in that place for its lifetime. An activity may be *running*, *blocked* on some condition or *terminated*. When the statement associated with an activity terminates normally, the activity terminates normally; when it terminates abruptly with some reason R, the activity terminates with the same reason.

An activity may be long-running and may invoke recursive methods (thus may have a stack associated with it). On the other hand, an activity may be short-running, involving a fine-grained operation such as a single read or write.

An activity may have an *activitylocal* heap accessible only to the activity.

X10 distinguishes between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate locally when the activity has finished all its computation related to that statement. (For instance the creation of an asynchronous activity terminates locally when the activity has been created.) It is said to terminate globally when it has terminated locally and all activities that it may have spawned at any place (if any) have, recursively, terminated globally.

An X10 computation is initiated as a single activity from the command line. This activity is the *root activity* for the entire computation. The entire computation terminates when (and only when) this activity globally terminates. Thus X10 does not permit the creation of so called "daemon threads" – threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§ 6.4).

6.1. The X10 rooted exception model

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

The presence of a root activity permits X10 to adopt a different model. In any state of the computation, say that an activity A is *a root of* an activity B if A is an ancestor of B and A is suspended at a statement (such as the `finish` statement § 6.3) awaiting the termination of B (and possibly other activities). For every X10 computation, the `root-of` relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the `root-of` tree when the activity terminates (based on the semantics of the statement being executed by the activity).[2] Thus, unlike concurrent languages such as Java no exception is "thrown on the floor".

6.2. Spawning an activity

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

An activity is created by executing the statement:

```
463 Statement ::= AsyncStatement
473 StatementNoShortIf ::= AsyncStatementNoShortIf
481 AsyncStatement ::=
      async PlaceExpressionSingleListopt Statement
491 AsyncStatementNoShortIf ::=
      async PlaceExpressionSingleListopt
         StatementNoShortIf
524   PlaceExpressionSingleListopt ::=
525       | PlaceExpressionSingleList
499   PlaceExpressionSingleList ::=
         ( PlaceExpression )
500   PlaceExpression ::= Expression
```

The place expression `e` is expected to be of type `place`, e.g. `here` or `place.FIRST_PLACE` or `d[p]` for some distribution `d` and point `p` (§ 5). If not, the compiler replaces `e` with `e.location`. (Recall that every expression in X10 has a type; this type is a subtype of the root class `x10.lang.Object`. This class has a field `location` of type `place` recording the place at which the value resides. See the documentation for `x10.lang.Object`.)

Note specifically that the expression `a[i]` when used as a place expression will evaluate to `a[i].location`, which may not be the same place as `a.distribution[i]`. The programmer must be careful to choose the right expression, appropriate for the statement. Accesses to `a[i]` within `Statement` should typically be guarded by the place expression `a.distribution[i]`.

In many cases the compiler may infer the unique place at which the statement is to be executed by an analysis of the types of the variables occuring in the statement. (The place must be such that the statement can be executed safetly, without generating a `BadPlaceException`.) In such cases the programmer may omit the place designator; the compiler will throw an error if it cannot determine the unique designated place.[3]

The statement is subject to the restriction that it must be acceptable as the body of a `void` method for an anoymous inner class declared at that point in the code, which throws no checked exceptions. As such, it may reference variables in lexically enclosing scopes (including `clock` variables, § 7) provided that such variables are (implicitly or explicitly) `final`.

An activity A executes the statement `async (P) S` by launching a new activity B at the designated place, to execute the specified statement. The statement terminates locally as soon as B is launched. The activation path for B is that of A, augmented with information about the line number at which B was spawned. B terminates normally when S terminates normally. It terminates abruptly if S throws an (uncaught) exception. The exception is propagated to A if A is a root activity (see § 6.3), otherwise through A to A's root activity. Note that while an activity is running, exceptions thrown by activities it has already generated may propagate through it up to its root activity.

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the pool of

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

[2] In X10 v0.4x the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

[3] X10 v0.4x does not specify a particular algorithm; this will be fixed in future versions.

activities at the target place and will be executed in sequence or in parallel based on the local scheduler's decisions. If the programmer wishes to sequence their execution s/he must use X10 constructs, such as clocks and `finish` to obtain the desired effect.

6.3. Finish

The statement `finish S` converts global termination to local termination and introduces a root activity.

```
468 Statement ::= FinishStatement
478 StatementNoShortif ::=
      FinishStatementNoShortIf
488 FinishStatement ::= finish Statement
498 FinishStatementNoShortIf ::=
      finish StatementNoShortIf
```

An activity A executes `finish S` by executing `S`. The execution of `S` may spawn other asynchronous activities (here or at other places). Uncaught exceptions thrown or propagated by any activity spawned by `S` are accumulated at `finish S`. `finish S` terminates locally when all activities spawned by `S` terminate globally (either abruptly or normally). If `S` terminates normally, then `finish S` terminates normally and A continues execution with the next statement after `finish S`. If `S` terminates abruptly, then `finish S` terminates abruptly and throws a single exception formed from the collection of exceptions accumulated at `finish S`.

Thus a `finish S` statement serves as a collection point for uncaught exceptions generated during the execution of `S`.

Note that repeatedly `finish`ing a statement has no effect after the first `finish`: the behavior of `finish finish S` is indistinguishable from `finish S`.

Future extensions. The semantics of `finish S` is conjunctive; it terminates when all the activities created during the execution of `S` (recursively) terminate. In many situations (e.g. nondeterministic search) it is natural to require a statement to terminate when any *one* of the activities it has spawned succeeds. The other activities may then be safely aborted. Future versions of the language may introduce a `finishone S` construct to support such speculative or nondeterministic computation.

6.4. Initial activity

An X10 computation is initiated from the command line on the presentation of a classname `C`. The class must have a `public static void main(String[] a)` method, otherwise an exception is thrown and the computation terminates. The single statement

```
async (place.FIRST_PLACE)  finish C.main(s);
```

is executed where `s` is an array of strings created from command line arguments. This single activity is the root activity for the entire computation.

6.5. Asynchronous Expression and Futures

X10 provides syntactic support for *asynchronous expressions*, also known as futures:

```
511   Primary ::= FutureExpression
515   FutureExpression ::=
        future PlaceExpressionSingleListopt
            Expression
```

Intuitively such an expression evaluates its body asynchronously at the given place. The resulting value may be obtained from the future returned by this expression, by using the `force` operation.

In more detail, in an expression `future(Q)e`, the place expression `Q` is treated as in an `async` statement. `e` is an expression of some type `T`. `e` may reference only those variables in the enclosing lexical environment which are declared to be `final`.

If the type of `e` is `T@P` then the type of `future (Q)e` is `future<T@P>`. This type `future<T@P>` is defined as if by:

```
public interface future<T@P> {
  T@P force();
  boolean forced();
}
```

(Here we use the syntax for generic classes. X10 v0.4x does not support generic classes in their full generality. In particular, the user may not define generic classes. This is reserved for future extensions to the language.)

Evaluation of `future (Q)e` terminates locally with the creation of a value `f` of type `future<T@P>`. This value may be stored in objects, passed as arguments to methods, returned from method invocation etc.

At any point, the method `forced` may be invoked on `f`. This method returns without blocking, with the value `true` if the asynchronous evaluation of `e` has terminated globally and with the value `false` if it has not.

The method invocation `force` on `f` blocks until the asynchronous evaluation of `e` has terminated globally. If the evaluation terminates successfully with value `v`, then the method invocation returns `v`. If the evaluation terminates abruptly with exception `z`, then the method throws exception `z`. Multiple invocations of `force` (by this or any other activity) do not result in multiple evaluations of `e`. The results of the first evaluation are stored in the future `f` and used to respond to all `force` queries.

```
future<T@P> promise
     = future (a.distribution[3]) { a[3] };
T@P value = promise.force();
```

6.5.1. Implementation notes

Futures are provided in X10 for convenience; they may be programmed using latches, `async` and `finish` as described in § 6.6.3.

6.6. Atomic sections

Languages such as Java use low-level synchronization locks to allow multiple interacting threads to coordinate the mutation of shared data. X10 eschews locks in favor of a very simple high-level construct, the *atomic section*.

A programmer may use atomic sections to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple activities running in the same place.

6.6.1. Unconditional atomic sections

The simplest form of an atomic section is the *unconditional atomic section*:

```
461 Statement ::= AtomicStatement
474 StatementNoShortIf ::=
       AtomicStatementNoShortIf
482 AtomicStatement ::= atomic  Statement
492 AtomicStatementNoShortIf ::=
       atomic StatementNoShortIf
445 MethodModifier ::= atomic
```

`Statement` may include method calls, conditionals etc. It may not include the construction of any `async` activity. It may not include any statement that may potentially block at runtime (e.g. `when`, `force` operations, `next` operations on clocks, `finish`).

Also for the sake of efficient implementation X10 v0.4x requires that the atomic section be *analyzable*, that is, the set of locations that are read and written by the `BlockStatement` are bounded and determined statically.[4] The exact algorithm to be used by the compiler to perform this analysis will be specified in future versions of the language.

All these locations must statically satisfy the *locality condition*: they must belong to the place of the current activity. The compiler checks for this condition by checking whether the statement could be the body of a `void` method annotated with `local` at that point in the code (§ 9.3.2).

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

A compiler is allowed to reorder two atomic sections that have no data-dependency between them, just as it may reorder any two statements which have no data-dependencies between them. For the purposes of data dependency analysis, an atomic section is deemed to have read and written all data at a single program point, the beginning of the atomic section.

[4] A static bound is a constant that depends only on the program text, and is independent of any runtime parameters.

We allow methods of an object to be annotated with `atomic`. Such a method is taken to stand for a method whose body is wrapped within an `atomic` statement.

Note an important property of an (unconditonal) atomic section:

$$\texttt{atomic atomic } S \quad = \quad \texttt{atomic } S \tag{6.1}$$

Further, an atomic section will eventually terminate successfully or thrown an exception; it may not introduce a deadlock.

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

Example

The following class method implements a (generic) compare and swap (CAS) operation:

```
public atomic boolean CAS( Object target,
                           Object old,
                           Object new) {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

6.6.2. Conditional atomic sections

Conditional atomic sections are of the form:

```
465 Statement ::= WhenStatement
475 StatementNoShortIf ::= WhenStatementNoShortIf
483 WhenStatement ::=
          when ( Expression ) Statement
484     | WhenStatement
          or ( Expression ) Statement
```

In such a statement the one or more expressions are called *guards* and must be `boolean` expressions. The statements are the corresponding *guarded statements*. The first pair of expression and statement is called the *main clause* and the additional pairs are called *auxiliary clauses*. A statement must have a main clause and may have no auxiliary clauses.

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

We note two common abbreviations. The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends. Second, `when (c) {;}` may be abbreviated to `await(c);` – it simply indicates that the thread must await the occurrence of a certain condition before proceeding.

Conditions on `when` clauses. For the sake of efficient implementation certain restrictions are placed on the guards and statements in a conditional atomic section. First, guards are required not to have side-effects, not to spawn asynchronous activities and to have a statically determinable upper bound on their execution. These conditions are expected to be checked statically by the compiler.

Second, as for unconditional atomic sections, the set of memory locations accessed by a guarded statements are required to be bounded and statically anlayzable.

Third, guarded statements are required to be *flat*, that is, they may not contain conditional atomic sections. (The implementation of nested conditional atomic sections may require sophisticated operational techniques such as rollbacks.)

Fourth, X10 guarantees only *weak fairness* when executing conditional atomic sections. Let c be the guard of some conditional atomic section A. A is required to make forward progress only if c is *eventually stable*. That is, any execution $s_1, s_2, \ldots$ of the system is considered illegal only if there is a j such that c holds in all states s_k for $k > j$ and in which A does not execute. Specifically, if the system executes in such a way that c holds only intermmitently (that is, for some state in which c holds there is always a later state in which c does not hold), A is not required to be executed.

Rationale: The guarantee provided by `wait/notify` in Java is no stronger. Indeed conditional atomic sections may be thought of as a replacement for Java's wait/notify functionality.

Sample usage. There are many ways to ensure that a guard is eventually stable. Typically the set of activities are divided into those that may enable a condition and those that are blocked on the condition. Then it is sufficient to require that the threads that may enable a condition do not disable it once it is enabled. Instead the condition may be disabled in a guarded statement guarded by the condition. This will ensure forward progress, given the weak-fairness guarantee.

6.6.3. Examples

Bounded buffer. The following class shows how to implement a bounded buffer of size 1 in X10 for repeated communication between a sender and a receiver.

```
class OneBuffer {
  nullable Object datum = null;
  boolean filled = false;
  public
    void send(Object v) {
      when ( !filled ) {
        this.datum = v;
        this.filled = true;
    }
  }
  public
    Object receive() {
      when ( filled ) {
        Object v  = datum;
        datum = null;
        filled = false;
        return v;
      }
    }
  }
}
```

Implementing a future with a latch. The following class shows how to implement a *latch*. A latch is an object that is initially created in an *unlatched* state. During its lifetime it may transition once to a *forced* state. Once forced, it stays forced for its lifetime. The latch may be queried to determine if it is forced, and if so, an associated value may be retrieved. Below, we will consider a latch set when some activity invokes a `setValue` method on it. This method provides two values, a normal value and an exceptional value. The method `force` blocks until the latch is set. If an exceptional value was specified when the latch was set, that value is thrown on any attempt to read the latch. Otherwise the normal value is returned.

```
public interface future {
   boolean forced();
   Object force();
}
public class Latch implements future {
  boolean forced = false;
  nullable boxed result = null;
  nullable exception z = null;

  public atomic
   boolean setValue( nullable Object val ) {
   return setValue( val, null);
    }
   public atomic
   boolean setValue( nullable exception z ) {
        return setValue( null, z);
    }
    public atomic
    boolean setValue( nullable Object val,
                      nullable exception z ) {
        if ( forced ) return false;
        // these assignment happens only once.
        this.result = val;
        this.z = z;
        this.forced = true;
        return true;
    }
    public atomic boolean forced() {
        return forced;
    }
    public Object force() {
        when ( forced ) {
            if (z != null) throw z;
            return result;
        }
    }
}
```

Latches, `async` operations and `finish` operations may be used to implement futures as follows. The expression `future(P) {e}` can be translated to:

```
new RunnableLatch() {
    public Latch run() {
       Latch L = new Latch();
       async ( P ) {
          Object X;
          try {
              finish X = e;
```

```
            async ( L ) {
               L.setValue( X );
            }
         } catch ( exception Z ) {
            async ( L ) {
              L.setValue( Z );
            }
         }
      }
      return l;
   }
 }.run()
```

Here we assume that `RunnableLatch` is an interface defined by:

```
public interface RunnableLatch {
   Latch run();
}
```

We use the standard Java idiom of wrapping the core translation inside an inner class definition/method invocation pair (i.e. `new RunnableLatch() .....run()`) so as to keep the resulting expression completely self-contained, while executing statements inside the evaluation of an expression.

Execution of a `future(P) {e}` causes a new latch to be created, and an `async` activity spawned at `P`. The activity attempts to `finish` the assigned `x = e`, where `x` is a local variable. This may cause new activities to be spawned, based on `e`. If the assignment terminates successfully, another activity is spawned to invoke the `setValue` method on the latch. Exceptions thrown by these activities (if any) are accumulated at the `finish` statement and thrown after global termination of all activities spawned by `x=e`. The exception will be caught by the `catch` clause and stored with the latch.

A future to execute a statement. Consider an expression `onFinish {S}`. This should return a `boolean` latch which should be forced when `S` has terminated globally. Unlike `finish S`, the evaluation of `onFinish {S}` should locally terminate immediately, returning a latch. The latch may be passed around in method invocations and stored in objects. An activity may perform `force/forced` method invocations on the latch whenever it desires to determine whether `S` has terminated.

Such an expression can be written as:

```
new RunnableLatch() {
    public Latch run() {
       Latch L = new Latch();
       async ( here ) {
          try {
              finish S;
              L.setValue( true );
          } catch ( exception Z ) {
               L.setValue( Z );
          }
       }
       return L;
    }
 }.run()
```

6.7. Remote Method Invocation

We introduce shorthand for remote method invocation:

```
507   MethodInvocation ::=
        Primary -> identifier ( ArgumentListopt )
```

If the method named is `void` the expression `o -> m(a1,...,ak)` is equivalent to:

```
finish async (o) {o.m(a1,...,ak);}
```

Otherwise the expression is equivalent to

```
future (o){o.m(a1, ..., ak);}.force()
```

6.8. Iteration

We introduce k-dimensional versions of iteration operations `for` and `foreach`:

```
189 Statement ::= ForStatement
206 StatementNoShortIf ::=
      ForStatementNoShortIf
236 ForStatement ::= EnhancedForStatement
239 ForStatementNoShortIf ::=
      EnhancedForStatementNoShortIf
466 Statement ::= ForEachStatement
476 StatementNoShortIf ::=
      ForEachStatementNoShortIf
487 EnhancedForStatement ::=
       for ( FormalParameter : Expression )
         Statement
487 EnhancedForStatementNoShortIf ::=
       for ( FormalParameter : Expression )
           StatementNoShortIf
485  ForEachStatement ::=
       foreach ( FormalParameter : Expression )
           Statement
495  ForEachStatementNoShortIf ::=
        foreach ( FormalParameter : Expression )
          StatementNoShortIf
```

In both statements, the expression is intended to be of type `region`. Expressions `e` of type `distribution` and `array` are also accepted, and treated as if they were `e.region`. The compiler throws a type error in all other cases.

The formal parameter must be of type `point`. Exploded syntax may be used (§ 11.3). The parameter is considered implicitly final, as are all the exploded variables.

An activity executes a `for` statement by enumerating the points in the region in canonical order. The activity executes the body of the loop with the formal parameter(s) bound to the given point. If the body locally terminates successfully, the activity continues with the next iteration, terminating successfully when all points have been visited. If an iteration throws an exception then the for statement throws an exception and terminats abruptly.

An activity executes a `foreach` statement in a similar fashion except that separate `async` activities are launched in parallel in the local place for each point in the region. The statement

terminates locally when all the activities have been spawned. It never throws an exception, though exceptions thrown by the spawned activities are propagated through to the root activity.

In a similar fashion we introduce the syntax:

```
467 Statement ::= AtEachStatement
477 StatementNoShortIf ::=
      AtEachStatementNoShortIf
486 AtEachStatement ::=
      ateach ( FormalParameter : Expression )
         Statement
496 AtEachStatementNoShortIf ::=
      ateach ( FormalParameter : Expression )
          StatementNoShortIf
```

Here the expression is intendd to be of type `distribution`. Expressions `e` of type `array` are also accepted, and treated as if they were `e.distribution`. The compiler throws a type error in all other cases. This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, `ateach( point p[i1,...,ik]: A) S` may be thought of as standing for:

```
foreach (point p[i1,...,ik] : A)
  async (A.distribution[p]) {S}
```

7. Clocks

The standard library for X10, `x10.lang` defines a `final value class`, `clock` intended for repeated quiescence detection of arbitrary, data-dependent collection of activities. Clocks are a generalization of *barriers*. They permit dynamically created activities to register and deregister. An activity may be registered with multiple clocks at the same time. In particular, nested clocks are permitted: an activity may create a nested clock and within one phase of the outer clock schedule activities to run to completion on the nested clock. Neverthless the design of clocks ensures that deadlock cannot be introduced by using clock operations.

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

The key invariants associated with clocks may be identified as follows. At any stage of the computation, a clock has zero or more *registered* activities. An activity may use only those clocks it is registered with. An activity may be subscribed to zero or more clocks. An activity is registered with a clock when it is created. During its lifetime the only additional clocks it is registered with are exactly those that it creates. In particular it is not possible for an activity to obtain access to an "old" clock (one that existed when the activity was created) by reading a data-structure. This is accomplished by requiring that clocks cannot be stored in objects, only in "flow" variables that live on the stack.

An activity may perform the following operations on a clock `c`. It may *unregister* with `c`; after this, it may perform no further actions on `c` for its lifetime. It may *check* to see if it is unregistered on a clock. It may *register* a newly forked activity with `c`. It may *mark* a statement `S` for completion in the current phase by executing the statement `now(c) S`. It may *continue* the clock by executing `c.resume();`. This indicate to `c` that it has finished marking all statements it wishes to perform in the current phase. Finally, it may *block* (through the statement `next`) on all the clocks that it is registered with. (This operation implicitly `continue`'s all clocks for the activity.) It will resume from this statement only when all these clocks are ready to advance to the next phase.

A clock becomes ready to advance to the next phase when every activity registered with the clock has executed at least one `continue` operation on that clock and all statements marked for completion in the current phase have been completed.

Though clocks introduce a blocking statement (`next`) an important property of X10 is that clocks cannot introduce deadlocks. That is, the system cannot reach a quiescent state (in which no activity is progressing) from which it is unable to progress. For, before blocking each activity continues all clocks it is registered with. Thus if a configuration were to be stuck (that is, no activity can progress) all clocks will have been continued. But this implies that all activities blocked on `next` may continue and the configuration is not stuck.

7.1. Clock operations

The special statements introduced for clock operations are listed below.

```
462 Statement ::= ClockedStatement
472 StatementNoShortIf ::=
      ClockedStatementNoShortIf
479 NowStatement ::=
      now ( Clock ) Statement
480 ClockedStatement ::=
      clocked ( ClockList ) Statement
490 ClockedStatementNoShortIf ::=
      clocked ( ClockList )
         StatementNoShortIf
501 NextStatement ::= next ;
```

Note that `x10.lang.clock` provides several useful methods on clocks too (e.g. `drop`).

7.1.1. Creating new clocks

Clocks are created using the nullary constructor for `x10.lang.clock` via a factory method:

```
clock timeSynchronizer = clock.factory.clock();
```

All clocked variables are implicitly final. The initializer for a local variable declaration of type `clock` must be a new clock expression. Thus X10 does not permit aliasing of clocks. Clocks are created in the place global heap and hence outlive the lifetime of the creating activity. Clocks are instances of value classes, hence may be freely copied from place to place. (Clock instances typically contain references to mutable state that maintains the current state of the clock.)

The current activity is automatically registered with the newly created clock. It may deregister using the `deregister` method on clocks (see the documentation of `x10.lang.clock`). All activities are automatically deregistered from all clocks they are registered with on termination (normal or abrupt).

7.1.2. Registering new activities on clocks

The programmer may explicitly indicate the set of currently visible clocks that are to be used inside a statement using the `clocked` statement:

```
462 Statement ::= ClockedStatement
472 StatementNoShortIf ::=
       ClockedStatementNoShortIf
480 ClockedStatement ::=
       clocked ( ClockListopt )
         Statement
490 ClockedStatementNoShortIf ::=
       clocked ( ClockListopt )
         StatementNoShortIf
503 ClockList ::= Clock
504 ClockList ::= ClockList , Clock
ClockListopt ::=
   | ClockList
```

If the statement is an `async` or `foreach` or `ateach` then each newly created activity is registered with the given (possibly empty) set of old clocks. If an `async`/`foreach`/`ateach` does not occur immediately inside a `clocked` statement none of its newly created activity or activities are registered with any old clock.)

7.1.3. Continuing clocks

X10 permits *split phase* clocks. An activity may wish to indicate that it has completed whatever work it wishes to perform in the current phase of a clock `c` it is registered with, without suspending all activity. It may do so by executing the method invocation:

```
c.continue();
```

on a clock `c` it is registered with. (Nothing happens if the activity invokes this method on a clock it is not registered with.) Execution of this statement indicates that the activity will post no further statements on `c`, or perform any other actions on `c` (for instance create an activity that is registered with `c`) until the clock has advanced.

The compiler should issue an error if any activity has a potentially live execution path from a `continue` statement to a `now`, `clocked`, `drop` or `continue` statement on the same clock that does not go through a `next` statement.

7.1.4. Advancing clocks

An activity may execute the statement

```
next;
```

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

An X10 computation is said to be *quiescent* on a clock `c` if each activity registered with `c` has continued `c`. Note that once a computation is quiescent on `c`, it will remain quiescent on c forever (unless the system takes some action), since no other activity can become registered with `c`. That is, quiescence on a clock is a *stable property*.

Once the implementation has detected quiecence on `c`, the system marks all activities registered with `c` as being able to progress on `c`. An activity blocked on `next` resumes execution once it is marked for progress by all the clocks it is registered with.

7.1.5. Dropping clocks

An activity may drop a clock by executing:

```
c.drop();
```

The method does nothing if the activity has already dropped `c`. The compiler must ensure conservatively that after dropping `c` no activity can execute a statement on `c`.

7.1.6. Checking for dropped clocks

An activity may check that a clock is dropped by executing:

```
c.dropped()
```

This call returns a `boolean` value: `true` iff the activity has already executed `c.drop()`.

7.1.7. Posting statements on a clock

X10 provides syntactic support for a common idiom. Often it may be necessary for an activity A to require that a certain set of statements be executed to completion before a clock c can move forward, without the activity A actually waiting for the completion of the statement. We introduce the syntax:

```
461 Statement ::= NowStatement
471 StatementNoShortIf ::=
       NowStatementNoShortIf
479 NowStatement ::=
       now ( Clock ) Statement
489 NowStatementNoShortIf ::=
       now ( Clock ) StatementNoShortIf
```

In a statement `now(c) S;` the statement `S` must not be clocked on any old clock (including `c`). (Thus any `next` statement it executes may not involve suspending on `c` or any old clock.) However, `S` may create new internal clocks.

Execution of the statement `now (c) S;` causes the statement `S` to be executed in a local async activity, and the statement terminates locally. However, the clock `c` may not advance until `S` has terminated globally (either normally or abruptly). We say that the statement `S` has been *posted* on `c`.

Such a statement may be considered as shorthand for

```
clocked(c) async(here) {
   finish clocked() S;
   next;
}
```

An activity is spawned locally, clocked on the single clock `c`. Thus the execution of `next` by the activity will cause suspension of the activity until `c` is ready to tick. Furthermore, `c` cannot progress until this activity has executed `next`. The activity executes the statement `S` to completion without allowing it to access any old clocks.

7.1.8. Program equivalences

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

$$\texttt{now(c) now(d) S} = \texttt{now(d) now(c) S} \quad (7.1)$$

$$\texttt{now(c) now(c) S} = \texttt{now(c) S} \quad (7.2)$$

$$\texttt{c.continue(); next;} = \texttt{next;} \quad (7.3)$$

$$\begin{array}{l}\texttt{c.continue();}\\ \texttt{d.continue();}\end{array} = \begin{array}{l}\texttt{d.continue();}\\ \texttt{c.continue;}\end{array} \quad (7.4)$$

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

8. Interfaces

X10 v0.4x interfaces are essentially the same Java interfaces [6, §9]. An interface primarily specifies signatures for public methods. It may extend multiple interfaces.

Future version of X10 will introduce additional structure in interface definitions that will allow the programmer to state additional properties of classes that implement that interface. For instance a method may be declared `pure` to indicate that its evaluation cannot have any side-effects. A method may be declared `local` to indicate that its execution is confined purely to the current place (no communication with other places). Similarly, behavioral properties of the method as they relate to the usage of clocks of the current activity may be specified.

9. Classes

X10 classes are essentially the same as Java classes [6, §8]. Classes are structured in a single-inheritance code hierarchy, may implement multiple interfaces, may have static and instance fields, may have static and instance methods, may have constructors, may have static and instance initializers, may have static and instance inner classes and interfaces. X10 does not permit mutable static state, so the role of static methods and initializers is quite limited. Instead programmers should use singleton classes to carry mutable static state.

Method signatures may specify checked exceptions. Method definitions may be overridden by subclasses; the overriding definition may have a declared return type that is a subclass of the return type of the definition being overridden. Multiple methods with the same name but different signatures may be provided on a class (ad hoc polymorphism). The public/private/protected/package-protected access modification framework may be used.

Because of its different concurrency model, X10 does not support `transient` and `volatile` field modifiers.

9.1. Reference classes

A reference class is declared with the optional keyword `reference` preceding `class` in a class declaration. Reference class declarations may be used to construct reference types (§ 3.1). Reference classes may have mutable fields. Instances of a reference class are always created in a fixed place and in X10 v0.4x stay there for the lifetime of the object. (Future versions of X10 may support object migration.) Variables declared at a reference type always store a reference to the object, regardless of whether the object is local or remote.

9.2. Value classes

X10 singles out a certain set of classes for additional support. A class is said to be *stateless* if all of its fields are declared to be `final` (§ 3.6.1), otherwise it is *stateful*. (X10 has syntax for specifying an array class with final fields, unlike Java.) A *stateless (stateful) object* is an instance of a stateless (stateful) class.

X10 allows the programmer to signify that a class (and all its descendents) are stateless. Such a class is called a *value class*. The programmer specifies a value class by prefixing the modifier `value` before the keyword `class` in a class declaration. (A class not declared to be a value class will be called a *reference class*.) Each instance field of a value class is treated as `final`. It is legal (but neither required nor recommended) for fields in a value class to be declared final. It is a compile-time error for a value class to inherit from a stateful class or for a reference class to inherit from a value class. For brevity, the X10 compiler allows the programmer to omit the keyword `class` after `value` in a value class declaration.

```
447  ClassDeclaration ::= ValueClassDeclaration
448  ValueClassDeclaration ::=
       ClassModifiersopt value identifier Superopt
          Interfacesopt ClassBody
449  | ClassModifiersopt value class identifier
          Superopt Interfacesopt ClassBody
```

The `nullable` type-constructor (§ 3.2) can be used to declare variables whose value may be `null` or a value type.

X10 provides a built in `final` definition for `.equals()` for a value type, namely stable equality (`==`, § 11.4). The programmer is free to override `.equals` with his/her own definition. (The behavior of `==` cannot be overridden however.)

9.2.1. Representation

Since value objects do not contain any updatable locations, they can be freely copied from place to place. An implementation may use copying techniques even within a place to implement value types, rather than references. This is transparent to the programmer.

More explicitly, X10 guarantees that an implementation must always behave as if a variable of a reference type takes up as much space as needed to store a reference that is either null or is

bound to an object allocated on the (appropriate) heap. However, X10 makes no such guarantees about the representation of a variable of value type. The implementation is free to behave as if the value is stored "inline", allocated on the heap (and a reference stored in the variable) or use any other scheme (such as structure-sharing) it may deem appropriate. Indeed, an implementation may even dynamically change the representation of an object of a value type, or dynamically use different representations for different instances (that is, implement automatic box/unboxing of values).

Implementations are strongly encouraged to implement value types as space-efficiently as possible (e.g. inlining them or passing them in registers, as appropriate). Implementations are expected to cache values of remote final value variables by default. If a value is large, the programmer may wish to consider spawning a remote activity (at the place the value was created) rather than referencing the containing variable (thus forcing it to be cached).

9.2.2. Example

```
value LinkedList  {
  Object first;
  nullable LinkedList rest;
  public
     LinkedList(Object first) {
     this(first, null);
  }
  public
    LinkedList(Object first,
               nullable LinkedList rest) {
    this.first = first;
    this.rest = rest;
  }
  public
    Object first() {
    return first;
  }
  public
    nullable LinkedList rest() {
    return rest;
  }
  public
    void append(LinkedList l) {
    return (this.rest == null)
        ? new LinkedList(this.first, l)
        : this.rest.append(l);
  }
}
```

9.3. Method annotations

9.3.1. `atomic` annotation

A method may be declared `atomic`.

```
445   MethodModifier ::= atomic
```

Such a method is treated as if the statement in its body is wrapped implicitly in an `atomic` statement.

9.3.2. `local` annotation

A method may be declared `local`.

```
445   MethodModifier ::= local
```

By declaring a method `local` the programmer asserts that the execution of this method results only in local memory accesses, that is reads/writes to locations in the place of the current activity.

The compiler implements the following rules to guarantee that a method declared `local` is in fact local.

Let `o` be any expression occurring in the body of the method. Assume its static datatype is `F`.

- Local methods can only be overridden by local methods.
- If the body of the method contains any field access `o.e`, then the static placetype of `o` must be `here`.

  The programmer can always ensure that this condition is satisfied (albeit at the risk of introducing a runtime exception) by replacing each field access `o.e` with `((F@here) o).e`.
- If the body of the method contains any assignments to fields (e.g. `o.e Op= t`, or `Op o.e` or `o.e Op`) then the static placetype of `o` must be `here`.

  The programmer can always ensure that this condition is satisfied by replacing `o.e Op= t` by `o1.e Op=t` and preceding it (in the same basic block) with the local variable declaration `F@here o1 = (F@here) o` (for some new local variable `o1`). Similarly for `Op o.e` and `o.e Op`.
- Recall that the static placetype of an array access `o[e]` is `o.distribution[e]`. Therefore, any read/write array access `o[e]` must be guarded by the condition `o.distribution[e] == here`. (Since `e` may have side-effects, the compiler must ensure that the place check uses the value returned by the same expression evaluation that is used to access the array element.)
- If the body of the method contains any method invocation `o.m(t1,...,tk)` then the method invoked must be local. Additionally, the static place type of `o` must be `here`. As above, the programmer can always ensure the second condition is satisfied by writing such a method invocation as `((F@here) o).m(t1,...,tk)`.

Note that reads/writes to local variables or method parameters are always local, hence the compiler does not have to check any extra conditions.

A method declared `atomic` is automatically declared to be `local`.

10. Arrays

An array is a mapping from a distribution to a range data type. Multiple arrays may be declared with the same underlying distribution.

Each array has a field `a.distribution` which may be used to obtain the underlying distribution.

The distribution underlying an array `a` may be obtained through the field `a.distribution`.

10.1. Regions

A region is a set of indices (called *points*). X10 provides a built-in value class, `x10.lang.region`, to allow the creation of new regions and to perform operations on regions. This class is `final` in X10 v0.4x; future versions of the language may permit user-definable regions. Since regions play a dual role (values as well as types), variables of type `region` must be initialized and are implicitly `final`. Regions are first-class objects – they may be stored in fields of objects, passed as arguments to methods, returned from methods etc.

Each region `R` has a constant rank, `R.rank`, which is a non-negative integer. The literal `[]` represents the *empty region* and has rank `0`.

For instance:

```
region E = 1:100;
region R = [0:99, -1:MAX_HEIGHT];
region R = region.factory.upperTriangular(N);
region R = region.factory.banded(N, K);
   // A square region.
region R = [E, E];
   // Same region as above.
region R = [100, 100];
   // Represents  the empty region
region Null = [];
   // Represents the intersection of two regions
region AandB = A && B;
  // represents the union of two regions
region AOrB = A || B;
```

A region may be constructed using a comma-separated list of regions (§ 11.2) within square brackets, as above and represents the Cartesian product of each of the arguments. The bound of a dimension may be any final variable of a fixed-point numeric type. X10 v0.4x does not support hierarchical regions.

Various built-in regions are provided through factory methods on `region`. For instance:

- `region.factory.upperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `region.factory.lowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular `N x N` matrix.
- `region.banded(N, K)` returns a region corresponding to the non-zero indices in a banded `N x N` matrix where the width of the band is `K`

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of a region `R=[1:2,1:2]` are ordered as

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

Sequential iteration statements such as `for` (§ 6.8) iterate over the points in a region in the canonical order.

A region is said to be *convex* if it is of the form `[T1,..., Tk]` for some set of enumerations `Ti`. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them. (Note that `||` may produce non-convex regions from convex regions, e.g. `[1,1] || [3,3]` is a non-convex region.)

For each region `R`, the *convex closure* of `R` is the smallest convex region enclosing `R`. For each integer `i` less than `R.rank`, the term `R[i]` represents the enumeration in the `i`th dimension of the convex closure of `R`. It may be used in a type expression wherever an enumeration may be used.

Region variables can be declared and used within user programs. They are implicitly `final` since they can be used within type expressions (and hence must not take on different values at runtime). That is, X10 does not permit the declaration of mutable `region` variables.

10.1.1. Operations on Regions

Various non side-effecting operators (i.e. pure functions) are provided on regions. These allow the programmer to express sparse as well as dense regions.

Let `R` be a region. A subset of `R` is also called a *sub-region*.

Let `R1` and `R2` be two regions.

`R1 && R2` is the intersection of `R1` and `R2`.

`R1 || R2` is the union of the `R1` and `R2`.

`R1 - R2` is the set difference of `R1` and `R2`.

Two regions are equal if they represent the same set of points.

10.2. Distributions

A *distribution* is a mapping from a region to a set of places. X10 provides a built-in value class, `x10.lang.distribution`, to allow the creation of new distributions and to perform operations on distributions. This class is `final` in X10 v0.4x; future versions of the language may permit user-definable distributions. Since distributions play a dual role (values as well as types), variables of type `distribution` must be initialized and are implicitly `final`.

The *rank* of a distribution is the rank of the underlying region.

```
region R = [1:100]
distribution D = distribtion.factory.block(R);
distribution D = distribution.factory.cyclic(R);
distribution D = R -> here;
distribution D = distribution.factory.random(R);
```

Let `D` be a distribution. `D.region` denotes the underlying region. `D.places` is the set of places constituting the range of `D` (viewed as a function). Given a point `p`, the expression `D[p]` represents the application of `D` to `p`, that is, the place that `p` is mapped to by `D`.

When operated on as a distribution, a region `R` implicitly behaves as the distribution mapping each item in `R` to `here` (i.e. `R->here`, see below). Conversely, when used in a context expecting a region, a distribution `D` should be thought of as standing for `D.region`.

10.2.1. Operations returning distributions

Let R be a region, Q a set of places {p1,...,pk} (enumerated in canonical order), and P a place. All the operations described below are static methods on the class distribution.

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

Constant distributions. The distribution R->P maps every point in R to P.

Block distributions. The distribution block(R, Q) distributes the elements of R (in order) over the set of places Q in blocks as follows. Let p equal R div N and q equal R mod N, where N is the size of Q. The first q places get successive blocks of size $(p+1)$ and the remaining places get blocks of size p.

The distribution block(R) is the same distribution as block(R, place.places).

Cyclic distributions. The distribution cyclic(R, Q) distributes the points in R cyclically across places in Q in order.

The distribution cyclic(R) is the same distribution as cyclic(R, place.places).

Thus the distribution cyclic(place.MAX_PLACES) provides a $1-1$ mapping from the region place.MAX_PLACES to the set of all places and is the same as the distribution unique(place.places).

Block cyclic distributions. The distribution blockCyclic(R, N, Q) distributes the elements of R cyclically over the set of places Q in blocks of size N.

Arbitrary distributions. The distribution arbitrary(R,Q) arbitrarily allocates points in R to Q. As above, arbitrary(R) is the same distribution as arbitrary(R, place.places).

Domain Restriction. If D is a distribution and R is a sub-region of D.domain, then D | R represents the restriction of D to R. The compiler throws an error if it cannot determine that R is a sub-region of D.domain.

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

Note that D | here does not necessarily contain adjacent points. For instance, if D is a cyclic distribution, D | here will typically contain points that are P apart, where P is the number of places.

10.2.2. User-defined distributions

Future versions of X10 may provide user-defined distributions, in a way that supports static reasoning.

10.2.3. Operations on Distributions

A *sub-distribution* of D is any distribution E defined on some subset of the domain of D, which agrees with D on all points in its domain. We also say that D is a *super-distribution* of E. A distribution D1 *is larger than* D2 if D1 is a super-distribution of D2.

Let D1 and D2 be two distributions.

Intersection of distributions. D1 && D2, the intersection of D1 and D2, is the largest common sub-distribution of D1 and D2.

Asymmetric union of distributions. D1.overlay(D2, the asymmetric union of D1 and D2, is the distribution whose domain is the union of the regions of D1 and D2, and whose value at each point p in its domain is D1[p] if p lies in D.domain otherwise it is D2[p].

Disjoint union of distributions. D1 || D2, the disjoint union of D1 and D2, is defined only if the domains of D1 and D2 are disjoint. Its value is D1.overlay(D2) (or equivalently D2.overlay(D1). (It is the least super-distribution of D1 and D2.)

Difference of distributions. D1 - D2 is the largest sub-distribution of D1 whose domain is disjoint from that of D2.

10.2.4. Example

```
double[D] dotProduct(T[D] a, T[D] b) {
return (new T[1:D.places] (point j) {
    return (new T[D | here] (point i) {
        return a[i]*b[i];
    }).sum();
}).sum();
}
```

This code returns the inner product of two T vectors defined over the same (otherwise unknown) distribution. The result is the sum reduction of an array of T with one element at each place in the range of D. The value of this array at each point is the sum reduction of the array formed by multiplying the corresponding elements of a and b in the local sub-array at the current place.

10.3. Array initializer

```
450 ArrayCreationExpression ::=
      new ArrayBaseType Unsafeopt []
        ArrayInitializer
451   | new ArrayBaseType Unsafeopt
```

```
      [ Expression ]
452  | new ArrayBaseType Unsafeopt
      [ Expression ] Expression
453  | new ArrayBaseType Unsafeopt
      [ Expression ]
      (FormalParameter) MethodBody
454  | new ArrayBaseType value Unsafeopt
      [ Expression ]
455  | new ArrayBaseType value Unsafeopt
      [ Expression ] Expression
456  | new ArrayBaseType value Unsafeopt
      [ Expression ]
      ( FormalParameter ) MethodBody
457 ArrayBaseType ::= PrimitiveType
458  | ClassOrInterfaceType
530 Unsafeopt ::=
531  | unsafe
```

An array may be declared `unsafe` if it is intended to be allocated in an unmanaged region (e.g. for communication with native code). A `value` array is an immutable array. An array creation must take either an int as an argument or a distribution. In the first case an array is created over the distribution `[0:N-1]->here`; in the second over the given distribution. An array creation operation may also specify an initializer using functional syntax or the abbreviated formalparameter/methodbody syntax. The formal parameter may contain exploded parameters (Section 11.3). The function is applied in parallel at all points in the domain of the distribution. The array construction operation terminates locally only when the array has been fully created and initialized (at all places in the range of the distribution).

For instance:

```
int[.] data
    = new int[1000->here]
         new intArray.pointwiseOp(){
             public int apply(point p[i]){
               return i;
            }
     };
int[.] data2
    = new int value [[1:1000,1:1000]->here]
        (point p[i,j]){ return i*j; };
```

The first declaration stores in `data` a reference to a mutable array with `1000` elements each of which is located in the same place as the array. Each array component is initialized to `i`.

The second declaration stores in `data2` an (immutable) 2-d array over `[1:1000, 1:1000]` initialized with `i*j` at point `[i,j]`. It uses a more abbreviated form to specify the array initializer function.

Other examples:

```
int[.] data
   = new int[1000](point [i]){return i*i;};
float[D] d
   = new float[D] (point [i]){return 10.0*i;};
float[D] result
   = new float[D]
     (point [i,j]) {return i + j; };
```

10.4. Operations on arrays

In the following let `a` be an array with distribution `D` and base type `T`. `a` may be mutable or immutable, unless indicated otherwise.

10.4.1. Element operations

The value of `a` at a point `p` in its region of definition is obtained by using the indexing operation `a[p]`. This operation may be used on the left hand side of an assignment operation to update the value. The operator assignments `a[i] op= e` are also available in X10.

10.4.2. Constant promotion

For a distribution `D` and a constant or final variable `v` of type `T` the expression `new T[D](point p) { return v;} D v` denotes the mutable array with distribution `D` and base type `T` initialized with `v`.

10.4.3. Restriction of an array

Let `D1` be a sub-distribution of `D`. Then `a | D1` represents the sub-array of `a` with the distribution `D1`.

Recall that a rich set of operators are available on distributions (§ 10.2) to obtain sub-distributions (e.g. restricting to a subregion, to a specific place etc).

10.4.4. Assembling an array

Let `a1,a2` be arrays of the same base type `T` defined over distributions `D1` and `D2` respectively. Assume that both arrays are value or reference arrays.

Assembling arrays over disjoint regions If `D1` and `D2` are disjoint then the expression `a1 || a2` denotes the unique array of base type `T` defined over the distribution `D1 || D2` such that its value at point `p` is `a1[p]` if `p` lies in `D1` and `a2[p]` otherwise. This array is a reference (value) array if `a1` is.

Overlaying an array on another The expression `a1.overlay(a2)` (read: the array `a1` *overlaid with* `a2`) represents an array whose underlying region is the union of that of `a1` and `a2` and whose distribution maps each point `p` in this region to `D2[p]` if that is defined and to `D1[p]` otherwise. The value `a1.overlay(a2)[p]` is `a2[p]` if it is defined and `a1[p]` otherwise.

This array is a reference (value) array if `a1` is.

10.4.5. Global operations

Pointwise operations Suppose that `m` is an operation defined on type `T` that takes an argument of type `S` and returns a value of type `R`. Such an operation can be lifted pointwise to operate on a `T` array and an `S` array defined over the same distribution `D` to return an `R` array defined over `D`, using the `lift` operation, `a.lift(f, b)`.

Reductions Let `f` be a `binaryOp` defined on type `T` (e.g. see the specification of the classes `x10.lang.intArray`). Let `a` be a value or reference array over base type `T`. Then the operation `a.reduce(f)` returns a value of type `T` obtained by performing `m` on all points in `a` in some order, and in parallel.

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

Scans Let `m` be a reduction operator defined on type `T`. Let `a` be a value or reference array over base type `T` and distribution `D`. Then the operation `a||m()` returns an array of base type `T` and distribution `D` whose `i`th element (in canonical order) is obtained by performing the reduction `m` on the first `i` elements of `a` (in canonical order).

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

Other operations on arrays may be found in `x10.lang.intArray` and other related classes.

11. Statements and Expressions

X10 inherits all the standard statements of Java, with the expected semantics:

```
EmptyStatement      LabeledStatement
ExpressionStatement IfStatement
SwitchStatement     WhileDo
DoWhile             ForLoop
BreakStatement      ContinueStatement
ReturnStatement   ThrowStatement
TryStatement
```

We focus on the new statements in X10.

11.1. Assignment

X10 supports assignment `l = r` to array variables. In this case `r` must have the same distribution `D` as `l`. This statement involves control communication between the sites hosting `D`. Each site performs the assignment(s) of array components locally. The assignment terminates when assignment has terminated at all sites hosting `D`.

11.2. Point and region construction

X10 specifies a simple syntax for the construction of points and regions.

```
281   ArgumentList ::= Expression
282      | ArgumentList , Expression
512   Primary ::= [ ArgumentList ]
```

Each element in the argument list must be either of type `int` or of type `region`. In the former case the expression `[ a1,..., ak ]` is treated as syntactic shorthand for

```
point.factory.point(a1,..., ak)
```

and in the latter case as shorthand for

```
region.factory.region(a1,..., ak)
```

11.3. Exploded variable declarations

X10 permits a richer form of specification for variable declarators in method arguments, local variables and loop variables (the "exploded" or *destructuring* syntax).

```
81    VariableDeclaratorId ::=
           identifier [ IdentifierList ]
82       | [ IdentifierList ]
```

In X10 v0.4x the `VariableDeclaratorId` must be declared at type `x10.lang.point`. Intuitively, this syntax allows a point to be "destructured" into its corresponding `int` indices in a pattern-matching style. The kth identifier in the `IdentifierList` is treated as a `final` variable of type `int` that is initialized with the value of the kth index of the point. The second form of the syntax (Rule 82) permits the specification of only the index variables.

Future versions of the language may allow destructuring syntax for all value classes.

Example. The following example succeeds when executed.

```
public class Array1Exploded {
  public int select(point p[i,j], point [k,l]) {
      return i+k;
  }
  public boolean run() {
    distribution d =  [1:10, 1:10] -> here;
    int[.] ia = new int[d];
    for(point p[i,j]: [1:10,1:10]) {
        if(ia[p]!=0) return false;
        ia[p] = i+j;
    }
    for(point p[i,j]: d) {
      point q1 = [i,j];
      if (i != q1[0]) return false;
      if ( j != q1[1]) return false;
      if(ia[i,j]!= i+j) return false;
      if(ia[i,j]!=ia[p]) return false;
      if(ia[q1]!=ia[p]) return false;
   }
    if (! (4 == select([1,2],[3,4]))) return false;
     return true;
   }

  public static void main(String args[]) {
     boolean b= (new Array1Exploded()).run();
     System.out.println("++++++ "
                        + (b? "Test succeeded."
                           :"Test failed."));
     System.exit(b?0:1);
  }
}
```

11.4. Expressions

X10 inherits all the standard expressions of Java [6, § 15] – as modified to permit generics [4] – with the expected semantics, unless otherwise mentioned below:

```
Assignment MethodInvocation
Cast Class
ClassInstanceCreationExpression FieldAccessExpression
ArrayCreationExpression ArrayAccessExpression
PostfixExpression PrefixExpression
InfixExpression UnaryOperators
MultiplicativeOperators AdditiveOperators
ShiftOperators RelationalOperators
EqualityOperators BitwiseOperators
ConditionalOperators AssignmentOperators
```

Expressions are evaluated in the same order as they would in Java (primarily left-to-right).

We focus on the expressions in X10 which have a different semantics.

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

```
306 UnaryExpressionNotPlusMinus ::=
      CastExpression
506 CastExpression ::=
      ( Type ) UnaryExpressionNotPlusMinus
```

The result of this operation is a value of the given type if the cast is permissible at runtime. Both the data type and place type of the value are checked. Data type conversion is checked according to the rules of the Java language (e.g.[6, §5.5]). If the value cannot be cast to the appropriate data type, a `ClassCast` exception is thrown. Otherwise, if the value cannot be cast to the appropriate place type a `BadPlaceException` is thrown.

Any attempt to cast an expression of a reference type to a value type (or vice versa) results in a compile-time error. Some casts – such as those that seek to cast a value of a subtype to a supertype – are known to succeed at compile-time. Such casts should not cause extra computational overhead at runtime.

`instanceof` operator This operator takes two arguments; the first should be a `RelationalExpression` and the second a `Type`. At run time, the result of this operator is `true` if the `RelationalExpression` can be cast to `Type` without a `ClassCastException` being thrown. Otherwise the result is `false`.

Stable equality. Reference equality (`==`, `!=`) is replaced in X10 by the notion of stable equality so that it can apply uniformly to value and reference types.

Two values may be compared with the infix predicate `==`. The call returns the value `true` if and only if no action taken by any user program can distinguish between the two values. In more detail the rules are as follows.

If the values have a reference type, then both must be references to the same object.

If the values have a value type then they must be structurally equal, that is, they must be instances of the same value class or value array data type and all their fields or components must be `==`.

If one of the values is `null` then the predicate succeeds iff the other value is also `null`.

The predicate `!=` returns `true` (`false`) on two arguments if and only if the predicate `==` returns `false` (`true`) on the same arguments.

11.5. Linking with native code

X10 v0.4x supports a simple facility to permit the efficient intra-thread communication of an array of primitive type to code written in the language `C`. The array must be a "local" array. The primary intent of this design is to permit the reuse of native code that efficiently implements some numeric array/matrix calculation.

Future language releases are expected to support similar bindings to FORTRAN, and to support parallel native processing of distributed X10 arrays.

The interface consists of two parts. First, an array intended to be communicated to native code must be created as an `unsafe` array:

```
450 ArrayCreationExpression ::=
      new ArrayBaseType Unsafeopt [ ]
        ArrayInitializer
451   | new ArrayBaseType Unsafeopt [ Expression ]
452   | new ArrayBaseType Unsafeopt
          [ Expression ] Expression
453   | new ArrayBaseType Unsafeopt [ Expression ]
          ( FormalParameter ) MethodBody
454   | new ArrayBaseType value
           Unsafeopt [ Expression ]
455   | new ArrayBaseType value
           Unsafeopt [ Expression ] Expression
456   | new ArrayBaseType value
        Unsafeopt [ Expression ]
          ( FormalParameter ) MethodBody
530   Unsafeopt ::=
531     | unsafe
```

Unsafe arrays can be of any dimension. However, X10 v0.4x requires that unsafe arrays be of a primitive type, and local (i.e. with an underlying distribution that maps all elements in its region to `here`).

Unsafe arrays are allocated in a special array of memory that permits their efficient transmission to natively linked code.

Second, the X10 programmer may specify that certain methods are to be implemented natively by using the keyword `extern`:

```
446   MethodModifier ::= extern
```

Such a method must have the statement "`;`" as its body. X10 v0.4x requires that the method be `static`; this restriction is likely to be lifted in the future. Primitive types in the method argument are translated to their corresponding JNI type (e.g. `float` is translated to `jfloat`, `double` to `jdouble` etc). The

only non-primitive type permitted in an `extern` method is an (unsafe) array. This is passed at type `jlong` as an eight byte address into the unsafe region which contains the data for the array. (`jlong` is not the same as `long` on 32-bit machines.)

Since only the starting address of an array is passed, if the array is multidimensional, the user must explicitly communicate (or have a guarantee of) the rank of the passed array, and must either typecast or explicitly code the address calculation. Note that all X10 arrays are created in row-major order, and so any native routine must also access them in the same order.

For each class `C` that contains an `extern` method, the X10 compiler generates a text file `C_x10stub.c`. This file contains generated `C` stub functions which are called from the `extern` routines. The name of the stub function is derived from the name of the `extern` method. If the method is `C.process()`, the stub function will be `Java_C_C_process()`. The name is suffixed with the signature of the method if the method is overloaded.

The programmer must write `C` code to implement the native method, using the methods in the `C` stub file to call the actual native method. The programmer must compile these files and link them into a dynamically linked library (DLL). Note that the `jni.h` header file must be in the include path. The programmer must ensure this library is loaded by the program before the method is called e.g. add a `System.loadlibrary` call (in a static initializer of the X10 class).

Example. The following class illustrates the use of `unsafe` and native linking.

```
public class IntArrayExternUnsafe {
  public static extern
      void process(int [.] yy, int size);
  static System.loadLibrary("IntArrayExternUnsafe");
  public static void main(String args[]) {
     boolean b= (new IntArrayExternUnsafe()).run();
     System.out.println("++++++ Test "
                        +(b?"succeeded.":"failed."));
     System.exit(b?0:1);
  }
  public boolean run(){
    int high = 10;
    boolean verified=false;
    distribution d= (0:high) -> here;
    int [.] y = new int unsafe[d];
    for( int j=0;j < 10;++j)
        y[j] = j;
    process(y,high);
    for(int j=0;j < 10;++j){
      int expected = j+100;
      if(y[j] != expected){
        System.out.println("y["+j+"]="
                           +y[j]+" != "+expected);
        return false;
       }
    }
    return true;
  }
}
```

The programmer may then write the `C` code thus:

```
void IntArrayExternUnsafe_process(jlong yy,
                                  signed int size){
  int i;
  int* array = (int *)(long)yy;
  for(i = 0;i < size;++i){
    array[i] += 100;
  }
}
/* automatically generated in _x10stub.c*/
void
 Java_IntArrayExternUnsafe_IntArrayExternUnsafe_process
 (JNIEnv *env,  jobject obj,jlong yy,jint size){
   IntArrayExternUnsafe_process(yy,size);
}
```

This code may be linked with the stub file (or textually placed in it). The programmer must then compile and link the `C` code and ensure that the DLL is on the appropriate classpath.

EXAMPLE

This example illustrates 2-d Jacobi iteration.

```
public class Jacobi  {
  const int N=6;
  const double epsilon = 0.002;
  const double epsilon2 = 0.000000001;
  const region R = [0:N+1, 0:N+1];
  const region RInner= [1:N, 1:N];
  const distribution D = distribution.factory.block(R);
  const distribution DInner = D | RInner;
  const distribution DBoundary = D - RInner;
  const int EXPECTED_ITERS=97;
  const double EXPECTED_ERR=0.0018673382039402497;

  double[D] B = new double[D] (point p[i,j])
      {return DBoundary.contains(p)
                ? (N-1)/2 : N*(i-1)+(j-1);};

  public boolean run() {
     int iters = 0;
     double err;
     while(true) {
       double[.] Temp =
          new double[DInner] (point [i,j])
            {return (read(i+1,j)+read(i-1,j)
                     +read(i,j+1)+read(i,j-1))/4.0;};
       if((err=((B | DInner) - Temp).abs().sum())
           < epsilon)
          break;
       B.update(Temp);
       iters++;
     }
     System.out.println("Error="+err);
     System.out.println("Iterations="+iters);
     return Math.abs(err-EXPECTED_ERR)<epsilon2
         && iters==EXPECTED_ITERS;
  }
  public double read(final int i, final int j) {
     return future(D[i,j]) B[i,j].force();
  }
  public static void main(String args[]) {
     boolean b= (new Jacobi()).run();
     System.out.println("++++++ "
                        + (b? "Test succeeded."
                            :"Test failed."));
     System.exit(b?0:1);
  }
}
```

X10 SYNTAX

This section contains the complete grammar for X10. This includes all the new constructs in X10 discussed in the main body of this reference manual, as well as constructs obtained from Java which behave essentially identically to the corresponding `java` constructs.

Note that in this version of the grammar productions for the same non-terminal may occur non-contiguously. For instance `MethodModifier` is defined on lines `111--119` and `445-446`. This will be corrected in future versions of the grammar.

```
0     $accept ::= CompilationUnit
1     identifier ::= IDENTIFIER
2     PrimitiveType ::= NumericType
3      | boolean
4     NumericType ::= IntegralType
5      | FloatingPointType
6     IntegralType ::= byte
7      | char
8      | short
9      | int
10     | long
11    FloatingPointType ::= float
12     | double
13    ClassType ::= TypeName
14    InterfaceType ::= TypeName
15    TypeName ::= identifier
16     | TypeName . identifier
17    ClassName ::= TypeName
18    ArrayType ::= Type [ ]
19    PackageName ::= identifier
20      | PackageName . identifier
21    ExpressionName ::= identifier
22      | here
23      | AmbiguousName . identifier
24    MethodName ::= identifier
25      | AmbiguousName . identifier
26    PackageOrTypeName ::= identifier
27      | PackageOrTypeName . identifier
28    AmbiguousName ::= identifier
29      | AmbiguousName . identifier
30    CompilationUnit ::= PackageDeclarationopt ImportDeclarationsopt TypeDeclarationsopt
31    ImportDeclarations ::= ImportDeclaration
32      | ImportDeclarations ImportDeclaration
33    TypeDeclarations ::= TypeDeclaration
34      | TypeDeclarations TypeDeclaration
35    PackageDeclaration ::= package PackageName ;
36    ImportDeclaration ::= SingleTypeImportDeclaration
37      | TypeImportOnDemandDeclaration
38      | SingleStaticImportDeclaration
39      | StaticImportOnDemandDeclaration
40    SingleTypeImportDeclaration ::= import TypeName ;
41    TypeImportOnDemandDeclaration ::= import PackageOrTypeName . * ;
42    SingleStaticImportDeclaration ::= import static TypeName . identifier ;
43    StaticImportOnDemandDeclaration ::= import static TypeName . * ;
44    TypeDeclaration ::= ClassDeclaration
45      | InterfaceDeclaration
46      | ;
47    ClassDeclaration ::= NormalClassDeclaration
48    NormalClassDeclaration ::= ClassModifiersopt class identifier Superopt Interfacesopt ClassBody
49    ClassModifiers ::= ClassModifier
50      | ClassModifiers ClassModifier
51    ClassModifier ::= public
```

```
52      | protected
53      | private
54      | abstract
55      | static
56      | final
57      | strictfp
58    Super ::= extends ClassType
59    Interfaces ::= implements InterfaceTypeList
60    InterfaceTypeList ::= InterfaceType
61      | InterfaceTypeList , InterfaceType
62    ClassBody ::= { ClassBodyDeclarationsopt }
63    ClassBodyDeclarations ::= ClassBodyDeclaration
64      | ClassBodyDeclarations ClassBodyDeclaration
65    ClassBodyDeclaration ::= ClassMemberDeclaration
66      | InstanceInitializer
67      | StaticInitializer
68      | ConstructorDeclaration
69    ClassMemberDeclaration ::= FieldDeclaration
70      | MethodDeclaration
71      | ClassDeclaration
72      | InterfaceDeclaration
73      | ;
74    FieldDeclaration ::= FieldModifiersopt Type VariableDeclarators ;
75    VariableDeclarators ::= VariableDeclarator
76      | VariableDeclarators , VariableDeclarator
77    VariableDeclarator ::= VariableDeclaratorId
78      | VariableDeclaratorId = VariableInitializer
79    VariableDeclaratorId ::= identifier
80      | VariableDeclaratorId [ ]
81      | identifier [ IdentifierList ]
82      | [ IdentifierList ]
83    VariableInitializer ::= Expression
84      | ArrayInitializer
85    FieldModifiers ::= FieldModifier
86      | FieldModifiers FieldModifier
87    FieldModifier ::= public
88      | protected
89      | private
90      | static
91      | final
92      | transient
93      | volatile
94    MethodDeclaration ::= MethodHeader MethodBody
95    MethodHeader ::= MethodModifiersopt ResultType MethodDeclarator Throwsopt
96    ResultType ::= Type
97      | void
98    MethodDeclarator ::= identifier ( FormalParameterListopt )
99      | MethodDeclarator [ ]
100   FormalParameterList ::= LastFormalParameter
101     | FormalParameters , LastFormalParameter
102   FormalParameters ::= FormalParameter
103     | FormalParameters , FormalParameter
104   FormalParameter ::= VariableModifiersopt Type VariableDeclaratorId
105   VariableModifiers ::= VariableModifier
106     | VariableModifiers VariableModifier
107   VariableModifier ::= final
108   LastFormalParameter ::= VariableModifiersopt Type ...opt VariableDeclaratorId
109   MethodModifiers ::= MethodModifier
110     | MethodModifiers MethodModifier
111   MethodModifier ::= public
```

```
112     | protected
113     | private
114     | abstract
115     | static
116     | final
117     | synchronized
118     | native
119     | strictfp
120   Throws ::= throws ExceptionTypeList
121   ExceptionTypeList ::= ExceptionType
122     | ExceptionTypeList , ExceptionType
123   ExceptionType ::= ClassType
124   MethodBody ::= Block
125     | ;
126   InstanceInitializer ::= Block
127   StaticInitializer ::= static Block
128   ConstructorDeclaration ::= ConstructorModifiersopt ConstructorDeclarator Throwsopt ConstructorBody
129   ConstructorDeclarator ::= SimpleTypeName ( FormalParameterListopt )
130   SimpleTypeName ::= identifier
131   ConstructorModifiers ::= ConstructorModifier
132     | ConstructorModifiers ConstructorModifier
133   ConstructorModifier ::= public
134     | protected
135     | private
136   ConstructorBody ::= { ExplicitConstructorInvocationopt BlockStatementsopt }
137   ExplicitConstructorInvocation ::= this ( ArgumentListopt ) ;
138     | super ( ArgumentListopt ) ;
139     | Primary . this ( ArgumentListopt ) ;
140     | Primary . super ( ArgumentListopt ) ;
141   Arguments ::= ( ArgumentListopt )
142   InterfaceDeclaration ::= NormalInterfaceDeclaration
143   NormalInterfaceDeclaration ::= InterfaceModifiersopt interface identifier ExtendsInterfacesopt InterfaceBody
144   InterfaceModifiers ::= InterfaceModifier
145     | InterfaceModifiers InterfaceModifier
146   InterfaceModifier ::= public
147     | protected
148     | private
149     | abstract
150     | static
151     | strictfp
152   ExtendsInterfaces ::= extends InterfaceType
153     | ExtendsInterfaces , InterfaceType
154   InterfaceBody ::= { InterfaceMemberDeclarationsopt }
155   InterfaceMemberDeclarations ::= InterfaceMemberDeclaration
156     | InterfaceMemberDeclarations InterfaceMemberDeclaration
157   InterfaceMemberDeclaration ::= ConstantDeclaration
158     | AbstractMethodDeclaration
159     | ClassDeclaration
160     | InterfaceDeclaration
161     | ;
162   ConstantDeclaration ::= ConstantModifiersopt Type VariableDeclarators
163   ConstantModifiers ::= ConstantModifier
164     | ConstantModifiers ConstantModifier
165   ConstantModifier ::= public
166     | static
167     | final
168   AbstractMethodDeclaration ::= AbstractMethodModifiersopt ResultType MethodDeclarator Throwsopt ;
169   AbstractMethodModifiers ::= AbstractMethodModifier
170     | AbstractMethodModifiers AbstractMethodModifier
171   AbstractMethodModifier ::= public
```

```
172     | abstract
173   ArrayInitializer ::= { VariableInitializersopt ,opt }
174   VariableInitializers ::= VariableInitializer
175     | VariableInitializers , VariableInitializer
176   Block ::= { BlockStatementsopt }
177   BlockStatements ::= BlockStatement
178     | BlockStatements BlockStatement
179   BlockStatement ::= LocalVariableDeclarationStatement
180     | ClassDeclaration
181     | Statement
182   LocalVariableDeclarationStatement ::= LocalVariableDeclaration ;
183   LocalVariableDeclaration ::= VariableModifiersopt Type VariableDeclarators
184   Statement ::= StatementWithoutTrailingSubstatement
185     | LabeledStatement
186     | IfThenStatement
187     | IfThenElseStatement
188     | WhileStatement
189     | ForStatement
190   StatementWithoutTrailingSubstatement ::= Block
191     | EmptyStatement
192     | ExpressionStatement
193     | AssertStatement
194     | SwitchStatement
195     | DoStatement
196     | BreakStatement
197     | ContinueStatement
198     | ReturnStatement
199     | SynchronizedStatement
200     | ThrowStatement
201     | TryStatement
202   StatementNoShortIf ::= StatementWithoutTrailingSubstatement
203     | LabeledStatementNoShortIf
204     | IfThenElseStatementNoShortIf
205     | WhileStatementNoShortIf
206     | ForStatementNoShortIf
207   IfThenStatement ::= if ( Expression ) Statement
208   IfThenElseStatement ::= if ( Expression ) StatementNoShortIf else Statement
209   IfThenElseStatementNoShortIf ::= if ( Expression ) StatementNoShortIf else StatementNoShortIf
210   EmptyStatement ::= ;
211   LabeledStatement ::= identifier : Statement
212   LabeledStatementNoShortIf ::= identifier : StatementNoShortIf
213   ExpressionStatement ::= StatementExpression ;
214   StatementExpression ::= Assignment
215     | PreIncrementExpression
216     | PreDecrementExpression
217     | PostIncrementExpression
218     | PostDecrementExpression
219     | MethodInvocation
220     | ClassInstanceCreationExpression
221   AssertStatement ::= assert Expression ;
222     | assert Expression : Expression ;
223   SwitchStatement ::= switch ( Expression ) SwitchBlock
224   SwitchBlock ::= { SwitchBlockStatementGroupsopt SwitchLabelsopt }
225   SwitchBlockStatementGroups ::= SwitchBlockStatementGroup
226     | SwitchBlockStatementGroups SwitchBlockStatementGroup
227   SwitchBlockStatementGroup ::= SwitchLabels BlockStatements
228   SwitchLabels ::= SwitchLabel
229     | SwitchLabels SwitchLabel
230   SwitchLabel ::= case ConstantExpression :
231     | default :
```

```
232  WhileStatement ::= while ( Expression ) Statement
233  WhileStatementNoShortIf ::= while ( Expression ) StatementNoShortIf
234  DoStatement ::= do Statement while ( Expression ) ;
235  ForStatement ::= BasicForStatement
236    | EnhancedForStatement
237  BasicForStatement ::= for ( ForInitopt ; Expressionopt ; ForUpdateopt ) Statement
238  ForStatementNoShortIf ::= for ( ForInitopt ; Expressionopt ; ForUpdateopt ) StatementNoShortIf
239    | EnhancedForStatementNoShortIf
240  ForInit ::= StatementExpressionList
241    | LocalVariableDeclaration
242  ForUpdate ::= StatementExpressionList
243  StatementExpressionList ::= StatementExpression
244    | StatementExpressionList , StatementExpression
245  BreakStatement ::= break identifieropt ;
246  ContinueStatement ::= continue identifieropt ;
247  ReturnStatement ::= return Expressionopt ;
248  ThrowStatement ::= throw Expression ;
249  SynchronizedStatement ::= synchronized ( Expression ) Block
250  TryStatement ::= try Block Catches
251    | try Block Catchesopt Finally
252  Catches ::= CatchClause
253    | Catches CatchClause
254  CatchClause ::= catch ( FormalParameter ) Block
255  Finally ::= finally Block
256  Primary ::= PrimaryNoNewArray
257    | ArrayCreationExpression
258  PrimaryNoNewArray ::= Literal
259    | Type . class
260    | void . class
261    | this
262    | ClassName . this
263    | ( Expression )
264    | ClassInstanceCreationExpression
265    | FieldAccess
266    | MethodInvocation
267    | ArrayAccess
268  Literal ::= IntegerLiteral
269    | LongLiteral
270    | FloatingPointLiteral
271    | DoubleLiteral
272    | BooleanLiteral
273    | CharacterLiteral
274    | StringLiteral
275    | null
276  BooleanLiteral ::= true
277    | false
278  ClassInstanceCreationExpression ::= new ClassOrInterfaceType ( ArgumentListopt ) ClassBodyopt
279    | Primary . new identifier ( ArgumentListopt ) ClassBodyopt
280    | AmbiguousName . new identifier ( ArgumentListopt ) ClassBodyopt
281  ArgumentList ::= Expression
282    | ArgumentList , Expression
283  FieldAccess ::= Primary . identifier
284    | super . identifier
285    | ClassName . super . identifier
286  MethodInvocation ::= MethodName ( ArgumentListopt )
287    | Primary . identifier ( ArgumentListopt )
288    | super . identifier ( ArgumentListopt )
289    | ClassName . super . identifier ( ArgumentListopt )
290  PostfixExpression ::= Primary
291    | ExpressionName
```

```
292     | PostIncrementExpression
293     | PostDecrementExpression
294   PostIncrementExpression ::= PostfixExpression ++
295   PostDecrementExpression ::= PostfixExpression --
296   UnaryExpression ::= PreIncrementExpression
297     | PreDecrementExpression
298     | + UnaryExpression
299     | - UnaryExpression
300     | UnaryExpressionNotPlusMinus
301   PreIncrementExpression ::= ++ UnaryExpression
302   PreDecrementExpression ::= -- UnaryExpression
303   UnaryExpressionNotPlusMinus ::= PostfixExpression
304     | ~ UnaryExpression
305     | ! UnaryExpression
306     | CastExpression
307   MultiplicativeExpression ::= UnaryExpression
308     | MultiplicativeExpression * UnaryExpression
309     | MultiplicativeExpression / UnaryExpression
310     | MultiplicativeExpression % UnaryExpression
311   AdditiveExpression ::= MultiplicativeExpression
312     | AdditiveExpression + MultiplicativeExpression
313     | AdditiveExpression - MultiplicativeExpression
314   ShiftExpression ::= AdditiveExpression
315     | ShiftExpression << AdditiveExpression
316     | ShiftExpression >> AdditiveExpression
317     | ShiftExpression >>> AdditiveExpression
318   RelationalExpression ::= ShiftExpression
319     | RelationalExpression < ShiftExpression
320     | RelationalExpression GREATER ShiftExpression
321     | RelationalExpression <_= ShiftExpression
322     | RelationalExpression GREATER = ShiftExpression
323   EqualityExpression ::= RelationalExpression
324     | EqualityExpression == RelationalExpression
325     | EqualityExpression != RelationalExpression
326   AndExpression ::= EqualityExpression
327     | AndExpression AND EqualityExpression
328   ExclusiveOrExpression ::= AndExpression
329     | ExclusiveOrExpression XOR AndExpression
330   InclusiveOrExpression ::= ExclusiveOrExpression
331     | InclusiveOrExpression OR ExclusiveOrExpression
332   ConditionalAndExpression ::= InclusiveOrExpression
333     | ConditionalAndExpression AND_AND InclusiveOrExpression
334   ConditionalOrExpression ::= ConditionalAndExpression
335     | ConditionalOrExpression OR_OR ConditionalAndExpression
336   ConditionalExpression ::= ConditionalOrExpression
337     | ConditionalOrExpression QUESTION Expression : ConditionalExpression
338   AssignmentExpression ::= ConditionalExpression
339     | Assignment
340   Assignment ::= LeftHandSide AssignmentOperator AssignmentExpression
341   LeftHandSide ::= ExpressionName
342     | FieldAccess
343     | ArrayAccess
344   AssignmentOperator ::= =
345     | *=
346     | /=
347     | %=
348     | +=
349     | -=
350     | <<=
351     | >>=
```

```
352     | >>>=
353     | &=
354     | ^=
355     | |=
356   Expression ::= AssignmentExpression
357   ConstantExpression ::= Expression
358   Catchesopt ::=
359     | Catches
360   identifieropt ::=
361     | identifier
362   ForUpdateopt ::=
363     | ForUpdate
364   Expressionopt ::=
365     | Expression
366   ForInitopt ::=
367     | ForInit
368   SwitchLabelsopt ::=
369     | SwitchLabels
370   SwitchBlockStatementGroupsopt ::=
371     | SwitchBlockStatementGroups
372   VariableModifiersopt ::=
373     | VariableModifiers
374   VariableInitializersopt ::=
375     | VariableInitializers
376   AbstractMethodModifiersopt ::=
377     | AbstractMethodModifiers
378   ConstantModifiersopt ::=
379     | ConstantModifiers
380   InterfaceMemberDeclarationsopt ::=
381     | InterfaceMemberDeclarations
382   ExtendsInterfacesopt ::=
383     | ExtendsInterfaces
384   InterfaceModifiersopt ::=
385     | InterfaceModifiers
386   ClassBodyopt ::=
387     | ClassBody
388   ,opt ::=
389     | ,
390   ArgumentListopt ::=
391     | ArgumentList
392   BlockStatementsopt ::=
393     | BlockStatements
394   ExplicitConstructorInvocationopt ::=
395     | ExplicitConstructorInvocation
396   ConstructorModifiersopt ::=
397     | ConstructorModifiers
398   ...opt ::=
399     | ...
400   FormalParameterListopt ::=
401     | FormalParameterList
402   Throwsopt ::=
403     | Throws
404   MethodModifiersopt ::=
405     | MethodModifiers
406   FieldModifiersopt ::=
407     | FieldModifiers
408   ClassBodyDeclarationsopt ::=
409     | ClassBodyDeclarations
410   Interfacesopt ::=
411     | Interfaces
```

```
412  Superopt ::=
413    | Super
414  ClassModifiersopt ::=
415    | ClassModifiers
416  TypeDeclarationsopt ::=
417    | TypeDeclarations
418  ImportDeclarationsopt ::=
419    | ImportDeclarations
420  PackageDeclarationopt ::=
421    | PackageDeclaration
422  Type ::= DataType PlaceTypeSpecifieropt
423    | nullable Type
424    | future < Type GREATER
425  DataType ::= PrimitiveType
426  DataType ::= ClassOrInterfaceType
427    | ArrayType
428  PlaceTypeSpecifier ::= AT PlaceType
429  PlaceType ::= placelocal
430    | activitylocal
431    | current
432    | PlaceExpression
433  ClassOrInterfaceType ::= TypeName DepParametersopt
434  DepParameters ::= ( DepParameterExpr )
435  DepParameterExpr ::= ArgumentList WhereClauseopt
436    | WhereClause
437  WhereClause ::= : Expression
438  ArrayType ::= X10ArrayType
439  X10ArrayType ::= Type [ . ]
440    | Type reference [ . ]
441    | Type value [ . ]
442    | Type [ DepParameterExpr ]
443    | Type reference [ DepParameterExpr ]
444    | Type value [ DepParameterExpr ]
445  MethodModifier ::= atomic
446    | extern
447  ClassDeclaration ::= ValueClassDeclaration
448  ValueClassDeclaration ::= ClassModifiersopt value identifier Superopt Interfacesopt ClassBody
449    | ClassModifiersopt value class identifier Superopt Interfacesopt ClassBody
450  ArrayCreationExpression ::= new ArrayBaseType Unsafeopt [ ] ArrayInitializer
451    | new ArrayBaseType Unsafeopt [ Expression ]
452    | new ArrayBaseType Unsafeopt [ Expression ] Expression
453    | new ArrayBaseType Unsafeopt [ Expression ] ( FormalParameter ) MethodBody
454    | new ArrayBaseType value Unsafeopt [ Expression ]
455    | new ArrayBaseType value Unsafeopt [ Expression ] Expression
456    | new ArrayBaseType value Unsafeopt [ Expression ] ( FormalParameter ) MethodBody
457  ArrayBaseType ::= PrimitiveType
458    | ClassOrInterfaceType
459  ArrayAccess ::= ExpressionName [ ArgumentList ]
460    | PrimaryNoNewArray [ ArgumentList ]
461  Statement ::= NowStatement
462    | ClockedStatement
463    | AsyncStatement
464    | AtomicStatement
465    | WhenStatement
466    | ForEachStatement
467    | AtEachStatement
468    | FinishStatement
469  StatementWithoutTrailingSubstatement ::= NextStatement
470    | AwaitStatement
471  StatementNoShortIf ::= NowStatementNoShortIf
```

```
472     | ClockedStatementNoShortIf
473     | AsyncStatementNoShortIf
474     | AtomicStatementNoShortIf
475     | WhenStatementNoShortIf
476     | ForEachStatementNoShortIf
477     | AtEachStatementNoShortIf
478     | FinishStatementNoShortIf
479   NowStatement ::= now ( Clock ) Statement
480   ClockedStatement ::= clocked ( ClockList ) Statement
481   AsyncStatement ::= async PlaceExpressionSingleListopt Statement
482   AtomicStatement ::= atomic PlaceExpressionSingleListopt Statement
483   WhenStatement ::= when ( Expression ) Statement
484     | WhenStatement or ( Expression ) Statement
485   ForEachStatement ::= foreach ( FormalParameter : Expression ) Statement
486   AtEachStatement ::= ateach ( FormalParameter : Expression ) Statement
487   EnhancedForStatement ::= for ( FormalParameter : Expression ) Statement
488   FinishStatement ::= finish Statement
489   NowStatementNoShortIf ::= now ( Clock ) StatementNoShortIf
490   ClockedStatementNoShortIf ::= clocked ( ClockList ) StatementNoShortIf
491   AsyncStatementNoShortIf ::= async PlaceExpressionSingleListopt StatementNoShortIf
492   AtomicStatementNoShortIf ::= atomic StatementNoShortIf
493   WhenStatementNoShortIf ::= when ( Expression ) StatementNoShortIf
494     | WhenStatement or ( Expression ) StatementNoShortIf
495   ForEachStatementNoShortIf ::= foreach ( FormalParameter : Expression ) StatementNoShortIf
496   AtEachStatementNoShortIf ::= ateach ( FormalParameter : Expression ) StatementNoShortIf
497   EnhancedForStatementNoShortIf ::= for ( FormalParameter : Expression ) StatementNoShortIf
498   FinishStatementNoShortIf ::= finish StatementNoShortIf
499   PlaceExpressionSingleList ::= ( PlaceExpression )
500   PlaceExpression ::= Expression
501   NextStatement ::= next ;
502   AwaitStatement ::= await Expression ;
503   ClockList ::= Clock
504     | ClockList , Clock
505   Clock ::= identifier
506   CastExpression ::= ( Type ) UnaryExpressionNotPlusMinus
507   MethodInvocation ::= Primary ARROW identifier ( ArgumentListopt )
508   RelationalExpression ::= RelationalExpression instanceof Type
509   IdentifierList ::= IdentifierList , identifier
510     | identifier
511   Primary ::= FutureExpression
512   Primary ::= [ ArgumentList ]
513   AssignmentExpression ::= Expression ARROW Expression
514   Primary ::= Expression : Expression
515   FutureExpression ::= future PlaceExpressionSingleListopt { Expression }
516   FieldModifier ::= mutable
517     | const
518   PlaceTypeSpecifieropt ::=
519     | PlaceTypeSpecifier
520   DepParametersopt ::=
521     | DepParameters
522   WhereClauseopt ::=
523     | WhereClause
524   PlaceExpressionSingleListopt ::=
525     | PlaceExpressionSingleList
526   ArgumentListopt ::=
527     | ArgumentList
528   DepParametersopt ::=
529     | DepParameters
530   Unsafeopt ::=
531     | unsafe
```

CHANGES FROM V0.32

This is the first reference manual that corresponds to a working implementation. As such a number of details missing from v0.32 have been spelt out. A number of mistakes have been corrected, and clarifications added.

The semantics of exception handling across asynchronous activities has been clarified.

Exploded syntax has been introduced to make it convenient to destructure points.

11.5.1. Limitations

- Exception propagation from an activity to its invoking activity is not yet implemented.
- All the type checking rules are not implemented. Thus if your program is already correct, it will exeute correctly. If it is not correct, it may still execute and give a result.
- The predicate == for value types is not yet implemented.
- "Jagged" arrays are not yet implemented.

11.5.2. Future work

Language needs to be extended to support generic types, with type and value parameters.

Language needs to be extended to support type inference.

Language needs to be extended to support implicit syntax.

REFERENCES

[1] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[2] William Carlson, Tarek El-Ghazawi, Bob Numrich, and Kathy Yelick. Programming in the Partitioned Global Address Space Model, 2003. Presentation at SC 2003, http://www.gwu.edu/ upc/tutorials.html.

[3] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[4] Gilad Bracha et al. Adding Generics to the Java Programming Language, 2001. JSR 014, http://www.jcp.org/en/jsr/detail?id=014.

[5] Vijay Saraswat et al. The x10 Standard Class Library. Technical report, IBM TJ Watson Research Center, New York, 2004. Forthcoming.

[6] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

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

[8] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[9] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[10] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

==, 25
? place type, 8
[] syntax, 24

array
    access, 23
    pointwise operations, 24
    reductions, 24
    restriction, 24
    scans, 24
    union
      asymmetric, 24
      disjoint, 24
array types, 7
arrays, 21
    constant promotion, 23
    distribution, 21
assignment, 24
ateach, 17
atomic sections, 14

class, 19
    reference class, 19
    value class, 19
classcast, 25
clock
    clocked statements, 18
    continue, 18
    creation, 18
    drop, 18
    dropped, 19
    next, 18
    now, 19
clocks, 17
comments, 5
conversions, 11

distribution, 22
    arbitrary, 22
    block, 22
    block cyclic, 22
    constant, 22
    cyclic, 22
    difference, 23
    intersection, 22
    restriction
      domain, 22
      range, 22
    union
      asymmetric, 22
      disjoint, 23
    unique, 22
    user-defined, 22

expressions, 25
extern, 25

final variable, 9
finish, 13
for, 17
foreach, 17

here, 11

identifier, 5
initial activity, 13
instanceof, 25
interfaces, 19

keyword, 5

local
    local, 20
locality condition, 14

names, 11
null, 6
nullable, 6
nullary constructor, 10

Object, 10

packages, 11
place types, 8
place.location, 10
places, 11
    scoped, 8
    shared, 8
promotions, 11

region, 21
    ==, 22
    banded, 21
    convex, 21
    intersection, 22
    lowerTriangular, 21
    set difference, 22
    sub-region, 22
    union, 22
    upperTriangular, 21
remote method invocation, 16
root activity, 12

statements, 24
sub-distribution, 22

type constructors, 6
types, 5

Upper case convention, 11
Upper-case Convention, 6

variable
    final, 9
variable declarator
    exploded, 24

variables, 9

whitespace, 5

X10 productions, 28