X10: Cheat Sheet

Allgemein:

X10 knows classes, interfaces, structs, functions as first-class objects (functional).
Computation is performed in multiple places which contains data and that can be operated remotely:
Distribution ↔ Places; Concurrency ↔ Activities

async S

Stmt ::= `async` Stmt

- … creates a new child activity that executes a statement S
  - returns immediately
  - S may reference variables in enclosing blocks
  - Activities cannot be named
  - Activity cannot be aborted or cancelled

finish S

Stmt ::= `finish` Stmt

- Execute S, but wait until all (transitively) spawned asyncs have terminated
  - → finish is useful for expressing "synchronous" operations on (local or) remote data

---

- `for(…) + async` statements

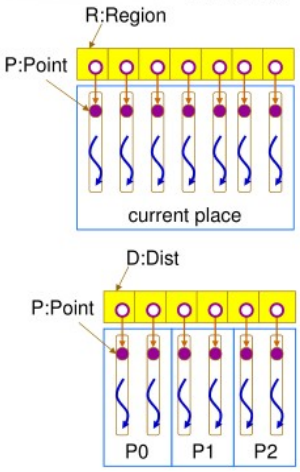
```
for (P in R) async S;
```

  - the statements launche separate **async** activities **in parallel** in the **local place**
  - for each object returned by the iteration
- `for(…)+ async + at` statements

```
for (P in D) async at
                (P) S;
```

  - the statement differs from `for + async` only in that each activity is spawned at the place specified by the distribution for the point

These are the most convenient ways to spawn activities



atomic S

Stmt ::= `atomic` Stmt
MethodModifier ::= `atomic`

- Execute statement S atomically
  - atomic blocks are conceptually executed in a serialized order with respect to all other atomic blocks
  - a method can be made atomic, too
- An atomic block body (S) –
  - must be non-blocking
  - must not create concurrent activities
    - i.e. is sequential
  - must not access remote data
    - i.e. operates locally
  - → restrictions are checked dynamically

---

at (p) S

Stmt ::= `at (p)` Stmt

- Execute statement *S* at Place *p*
  - parent activity is blocked until *S* completes
- Deep copy of local object graph to target place
  - the variables referenced from *S* define the roots of the graph to be copied
- *Place.places()* delivers an iterator for all available places
  - a specific place n can be referenced via *Place(n)*
- `here.next()` delivers the next place
  - called on the last place it delivers the first place

when (E) S

WhenStmt ::= `when` (Expr) Stmt
| WhenStmt `or` (Expr) Stmt

- Activity suspe
  which the guard E is true
  - → in that state, S is executed atomically
- The guard E is a boolean expression
  - must be nonblocking
  - must not create concurrent activities
    - i.e. it is sequential
  - must not access remote data
    - i.e. it operates locally
  - must not have side-effects
    - i.e. it is const

Distributed Object Model:

- Objects live in a single place
  - they can only be accessed in the place where they live (PGAS!)
- *GlobalRef[T]* used to create remote pointers across "at's boundaries"
  - a GlobalRef can be freely copied
    - reference can be accessed via the `()` operator
    - only at its home place

```
class C {
  var x:int;
  def this(n:int) {x = n;}
}

  // someplace else
  // Incr. remote counter
def inc(c:GlobalRef[C]) {
  at (c.home) c().x++;
}

// Create GR of C
def make(init:int) {
  val c = new C(init);
  return GlobalRef[C](c);
}
```

DistArrays:

- A Dist (distribution) maps every Point in its Region to a Place
  - in other words, it spreads data automatically over many places
  - DistArray name is visible at all places
- A DistArray (distributed array) is defined over a Dist and maps every Point in its distribution to a corresponding data element
  - data elements in the DistArray may only be accessed in their home location
    - i.e. the place to which the Dist maps their Points
  - example, create a distributed array with 1000 Ints –

```
val array = new Array[Int](1..1000, ([i]:Point) => 0);
val dist = Dist.makeBlock(array.region);
val distArray = DistArray.make(dist, ([i]:Point) => array(i));
```

    - a BlockDistribution maps the cells as evenly as possible to the places
- Key operations similar to Array
  - but bulk operations (map, reduce, etc) are distributed operations