Concerning multiplicities

Make multiplicities in signatures explicit. For example, write

```
abstract sig Place extends Node
{
  tokens : one Int
}
```

instead of

```
abstract sig Place extends Node
{
  tokens : Int
}
```

That applies also to the "target side" of relational bounding expressions. For example, write

```
abstract sig Obj
{
  get : FName -> set Obj
}
```

instead of

```
abstract sig Obj
{
  get : FName -> Obj
}
```

Likewise, be explicit about what the intended multiplicity of arguments is in predicate definitions, if it is not `one`. For example, write

```
pred concurrentlyEnabled[ts : set Transition] {
  all p : Place | p.tokens ≥ (sum t : ts | p.flow[t])
}
```

instead of

```
pred concurrentlyEnabled[ts : Transition] {
  all p : Place | p.tokens ≥ (sum t : ts | p.flow[t])
}
```

Analogously for function definitions.

Concerning "type-checking" of function/predicate use

Even though Alloy does not actually check the multiplicities, in particular, scalar vs. actual sets, make sure that all uses are consistent with intended multiplicities of arguments in definitions. For example, given the definition

```
pred enabled[t : Transition] {
  all p : Place | p.tokens ≥ p.flow[t]
}
```

do not do something like

```
pred someOtherPredicate[ts : set Transition] {
  enabled[ts] and ...
}
```

Concerning implicit summing of integer sets

Do not rely, except in very clear circumstances, on the implicit summing of integer sets in comparison expressions. That is, in a comparison like `a`$\geq$`b`, where `a` or `b` might be an actual set (with possibly more than one element), rather write explicitly `a.sum`$\geq$`b.sum`. However,

```
pred enabled[t : Transition] {
  all p : Place | p.tokens ≥ p.flow[t]
}
```

is okay, if by the definition of `flow` we know that `p.flow[t]` is empty or a singleton, so we can avoid writing more explicitly `p.flow[t].sum`.

Concerning arithmetic comparisons

Always be as strict as possible. For example, write

```
pred outComing[t : Transition] {
  (sum p : Place | t.flow[p]) > 0
}
```

instead of

```
pred outComing[t : Transition] {
  (sum p : Place | t.flow[p]) ≠ 0
}
```

Concerning precedences in expressions

Add extra parentheses even when Alloy's precedence rule "`&` binds tighter than `=`" would make if possible to avoid them. For example, write

```
fact {
  all p : Place | (p.flow.Int & Place) = none
}
```

instead of

```
fact {
  all p : Place | p.flow.Int & Place = none
}
```

Concerning logical operators

Always use the verbose forms of logical operators, that is, `not`, `and`, `or`, `implies`, `iff`, instead of their symbolic shorthand forms.

Concerning arithmetic operators

Use `plus` and `minus` instead of `add` and `sub`.

Concerning scope

Always make the scope used explicit. For example, write

```
run show for 3
```

instead of

```
run show
```