NPL Cheat Sheet

This is a non-exhaustive list of the most important NPL features and how to use them, as briefly as possible. Non-standard whitespace and short identifiers are used to support printability, so the code style should not be emulated.

Protocols

Declaration

```
protocol[party1, party2] Foo(
  var arg: Number,
  var arg2: Text
) { /*..*/ };
```

Instantiation

```
var f = Foo[p, q](42, "b");
// With named arguments:
var f2 = Foo[
  party1 = p,
  party2 = q
](arg2 = "b", arg = 42);
```

`@api`-annotated protocols can also be instantiated from the API.

Permissions

Declaring permissions

```
protocol[a, b] Bar() {
  permission[a] foo(
    x: Text
  ) returns Text {
    /*..*/
    return "foo";
  }
  permission[a | b] bar() { /*..*/ }
}
```

Invoking permissions

Permissions can be invoked from within NPL:

```
var b = Bar[alice, bob]();
var x = b.foo[alice]("x");
b.bar[bob]();
```

`@api`-annotated permissions can also be invoked from the API.

Require conditions

Permissions can be safeguarded by `require` conditions, and are great for validating input:

```
permission[p] spend(
  amount: Number
) {
  require(
    (amount > 0),
    "Amount must be positive"
  );
}
```

States

Declaring states

```
protocol[p, q] MyProtocol() {
  initial state unpaid;
  state billed;
  state reminded;
  final state paid;
  final state forgiven;
}
```

States as guards

States can be used to only allow permission when the parent protocol is in specific states:

```
permission[p] pay(
  amount: Number
) | unpaid, billed, reminded {
  //...
};
```

State transitions

```
permission[p] pay2(
  amount: Number
) | unpaid, billed, reminded {
  if (amount > 42) { become paid; };
}
```

Interacting with states

Protocol *types* expose a `States` enum, and protocol instances expose some helper methods:

```
protocol[p] Qux() {
  initial state a;
  final state b;
}
```

```
var q = Qux[p]();
test.assertEquals(
  listOf(
    Qux.States.a,
    Qux.States.b
  ),
  Qux.States.variants()
);
test.assertEquals(
  optionalOf(Qux.States.a),
  q.initialState()
);
test.assertEquals(
  setOf(Qux.States.b),
  q.finalStates()
);
test.assertEquals(
  optionalOf(Qux.States.a),
  q.activeState()
);
```

Party system

Parties are the entities that interact with protocols. When developing NPL you define the conceptual relations between parties and the logic that dictates their interactions, while the actual parties are bound to the protocol instances at runtime via your Identity and Access Management (IAM) system.

Entity and access

Entity and access are runtime concepts that are used to determine whether a party is allowed to interact with a protocol instance or permission. *Entity* represents the identity of the party and is immutable, while *access* represents the party's *current* access rights and can be changed via the API.

Refer to the standard library reference for examples.

Observers

Observers are parties that are allowed to observe and read a protocol instance. They need to be explicitly added or removed using permissions.

All protocol instances have an implicit (empty) `observers` field of type `Map<Text, Party>`.

```
permission[*newObs & issuer] addObs(
  name: Text
) {
  observers =
    observers.with(name, newObs);
}
permission[issuer] removeObs(
  name: Text
) {
  observers =
    observers.without(name);
}
```

Types

This section is non-exhaustive and only covers the most important types. For a complete list, refer to the standard library reference.

Basic types

```
var a: Number = 42 + 0.01 * 5;
var b: Text =
  "foo" + "bar" + a.toText();
var c: Boolean =
  true || ((5 > 3) && false);
var d: LocalDate =
  localDateOf(2021, 1, 1);
var e: DateTime = dateTimeOf(
  2021, 1, 1, 12, 0,
  valueOfZoneId(
    ZoneId.EUROPE_ZURICH
  )
);
var f: Duration = millis(4)
  .plus(seconds(2))
  .multiplyBy(50)
  .plus(minutes(59))
  .minus(hours(1));
var g: Period = days(42)
  .plus(weeks(2))
  .minus(months(1))
  .multiplyBy(2);
```

Optionals

`Optional<T>` is a typesafe wrapper for values that may be either `Some<T>` or `None`, where `T` is the type of the wrapped value.

```
var s: Optional<Number> =
  optionalOf(42);
var n = optionalOf<Number>();
t.assertEquals(42, s.getOrElse(1));
t.assertEquals(1, n.getOrElse(1));
t.assertEquals(42, s.getOrFail());
t.assertFails(
  function() -> { return
n.getOrFail(); }
);
t.assertTrue(s.isPresent());
t.assertFalse(n.isPresent());
```

Collections

```
var l: List<Number> = listOf(1, 1);
var s: Set<Number> = setOf(1, 2);
var m: Map<Text, Number> = mapOf(
  Pair("foo", 1),
  Pair("bar", 2)
);
```

See the relevant docs or use the IDE to explore the supported receiver methods.

User-defined types

Structs

Structs store arbitrary data in an organized way. They are immutable and belong to particular protocol instances.

```
// Declaration
struct St {
  a: Number,
  b: Text
}
```

```
// Instantiation
var st: St = St(42, "foo");
var st2: St = St(b = "s", a = 42);
// Copying/overwriting
var st3: St = st.copy(b = "t");
```

Enums

```
// Declaration
enum Color { Red, Blue }
```

```
// Instantiation
var blue: Color = Color.Blue;
// Matching
var c: Text = match(blue) {
  Color.Red -> "red"
  Color.Blue -> "blue"
};
```

Unions

```
// Declaration
union U { Number, Text }
```

```
// Instantiation
var u1: U = U(42);
var u2: U = U("foo");
```

Identifiers

Identifiers define opaque, typed identifiers.

```
// Declaration
identifier Id;
```

```
// Instantiation
var id = Id();
// Usage
var m = mapOf(Pair(id, "foo"));
```

Symbols

The `Symbol` type is used to represent `Number` values with a unit.

```
// Declaration
symbol usd;
```

```
// Instantiation
var x: usd = usd(4);
```

Control flow

```
if (a == b) {/*..*/};
if (a != b) {/*..*/} else {/*..*/};
for (item in collection) {/*..*/};
match (c) {
  Color.Blue -> {/*..*/}
  Color.Red -> {/*..*/}
};
match(c) {
  Color.Red -> {/*..*/}
  else -> {/*..*/}
};
```

Functions and lambdas

Functions can be defined on the top-level or within protocols. They support type inference and the body can be an expression.

```
function e(a: Text) returns Text -> {
  return a + "!";
}
function p2(a: Number) -> a + 2;
```

Anonymous functions use the same syntax but omit the name, and can be used as values:

```
var l = function(a: Number) -> a * 3;
t.assertEquals(
  listOf(1, 2, 3).map(l),
  listOf(3, 6, 9)
);
```

Logging

Log the text representation of any value:

```
debug(42);
info("foo");
error(Foo[p, q](42, "b"));
```

Testing

```
@test
function testStuff(t: Test) -> {
  t.assertEquals(1, 1);
}
```

Read the relevant docs or use the IDE autocompletion to learn more about other supported assertions