Clarifications on Pattern-Matching

1 Patterns we are allowed to match

We need to clarify the implications of the sentence from the tutorial, “There are two kinds of values in Rholang, names and processes. Patterns are names or processes with free variables, which appear to the left of an arrow in a `for` or a `match`.” This has a couple of implications that aren’t explicitly stated (yet) in the tutorial.

1. A program cannot have any free variables. It also can’t have any logical connectives `/\` or `\/` , joins, arrows `=>` , etc. Logical connectives, joins and arrows can be used if they are in patterns within the program, and any variable in a program must be at most locally free. For example, the following code snippets are not valid programs, despite the fact that they are valid components of patterns:

   - `@Nil!(Nil) /\ @Nil!(Nil)`
   - `@Nil!(Nil) ; @Nil!(Nil)`
   - `@Nil!(Nil) , @Nil!(Nil)`
   - `x => @Nil!(Nil)`

2. In the same vein, a process variable does *not* match with anything that is not a process, meaning that it cannot match with a statement that contains a free variable, a join, a logical connective, etc unless those are written in a pattern fully contained in the statement. Likewise, a name variable cannot match with anything that is not a quoted process, in the sense that it cannot contain free variables, joins, logical connectives, etc unless they are correctly written in a pattern fully contained in the quoted process. For example, the following code

   ```
   match for( x <- @Nil){ Nil } { for( x <- y){ Nil } => { y!(Nil) }}
   ```

   will match, returning `@Nil!(Nil)`, but

   ```
           match for( x <- @for(x <- @Nil){ x!(Nil) } ){ Nil }
       { for( x <- for( t <- @Nil ){ y } { Nil } => { y!(Nil) }}
   ```

   will evaluate to the empty process due to not having matched, since `y` cannot match with `x!(Nil)`. Finally,

   ```
   match for( x <- @z!(Nil)){ Nil } { for( x <- y){ Nil } => { y!(Nil) }}
   ```

   won’t compile, due to the globally free variable `z`.

2 Name equivalence

The RHO calculus (on which Rholang is based) says that two names are equivalent when the processes that they quote are equivalent, and that processes are equated via the relations below (and nothing more). Here, `P, Q` and `R` are processes:

- `P | Q = Q | P` (commutativity)
- `P = P | Nil = Nil | P` (identity)
- `(P | Q) | R = P | (Q | R)` (associativity)

Therefore, `@ P | Q = @ Q | P` and `@ P | Nil = @ P = @ Nil | P`, etc.

In Rholang, these relations only apply *to the top level* of any name. In addition to the three given above, we also evaluate expressions on the top level. So for example, as channels,

- `@{10 + 2} = @{5 + 7} = @{12}`, and
- `@for(x <- @Nil){10 + 2} = @for(x <- @Nil){5 + 7} = @for(x <- @Nil){12}`.

Since we use variables, channels also respect α-equivalence, meaning that, for example,

- `@ for( x <- @Nil ){ Nil } = @for( z <- @Nil ){ Nil }`.

In the RHO calculus we don't have to worry about distinguishing the top level from other parts of a channel, but because of things like pattern-matching, we have to in Rholang. This will be relevant in Section 3, where there are restrictions on pattern-matching because of this.

2.1 "Looking through the looking glass": Patterns Within Patterns.

Patterns follow the same rules as channels in terms of equivalence. In the case of -arity matching, joins, logical connectives, etc. the rules apply to each name pattern individually.

Here we need to treat statements that are not on the top level. We reemphasize that the rules for name equivalences described in Section 2 only apply on the top level. When we are matching *patterns within patterns*, these equivalence rules do not apply and we check for an exact text match, up to α-equivalence. This means that the following to channels are *not* equivalent:

```
@for( @for( Nil | x <- @Nil ){ Nil } <- @Nil ){ Nil }

@for( @for( x | Nil <- @Nil ){ Nil } <- @Nil ){ Nil }
```

Note that while the patterns `x | Nil` and `Nil | x` are equivalent, the patterns

```
@for( Nil | x <- @Nil ){ z }

@for( x | Nil <- @Nil ){ z }
```

are *not*. When we are matching *patterns within patterns*, these equivalence rules do not apply and we check for an exact text match, up to α-equivalence. You can only look through the looking glass once: when checking name equivalence or matching a pattern to a name or a process, if a given pattern is part of a larger pattern, there has to be an exact match.

Furthermore, we can't bind variables to parts of patterns. For example, the following send/receive will not match:

```
for( @ for( @x!(y) <- @Nil ){ Nil } <- @Nil ){ Nil } |

    @Nil!( for( @x!(10) <- @Nil ){ Nil } ).
```

Naïvely, one might expect these to match, binding `y` to `10`, but to match with the above receive, one must send something α-equivalent to:

```
@Nil!( for( @x!(y) <- @Nil ){ Nil } ).
```

3 Ambiguities in Patterns and Illegal Moves

When using parallel processes in patterns, it is possible that a pattern will be able to match more than one way to a name or a process, for example when we are matching parallel processes, as in the following code:

```
for( @ x | y <- @Nil ){ Nil } | @Nil!( 10 | 20 ).
```

This send/receive will nondeterministically either bind `x` to `10`, `x` to `10 | 20` or `x` to the empty process `Nil` while `y` binds to whatever `x` does not take, binding to `Nil` if `x` binds to `10 | 20`. (Although as the interpreter stands, Rholang will choose one of `x` and `y`, give it `10 | 20`, and bind the other variable to `Nil`.)

When writing more complicated patterns, one should be careful to use correct parentheses. For example, intuitively the name pattern

```
@{@x!(Nil) | y!(Nil)}
```

can be interpreted as `@{@{x!(Nil) | y}!(Nil)}`, where `x` is a name variable and `y` is a process variable, or as `@{{@x!(Nil)} | {y!(Nil)}}`, where `x` is a process variable and `y` is a name variable. The chosen interpretation is vital, since the way we use `x` and `y` in the body depend on the type (process or name) of each of these terms. If the first interpretation is correct,

```
for( @{@x!(Nil) | y!(Nil)} <- @10 ){ x!("success") | @y!("success") }
```

will compile, while

```
for( @{@x!(Nil) | y!(Nil)} <- @10 ){ @x!("success") | y!("success") }
```

will not. If the second interpretation is correct, the opposite is true. Currently, the second interpretation is correct, but when in doubt it is wise to use parentheses.

We also cannot match parts of arithmetic operations (see Section 2). For example, we cannot write

```
for( @{x + 7} <- @Nil ){ Nil }.
```

We might expect it to match with `@Nil!(5 + 7)`, binding `x` to `5` but when messages are sent, top-level expressions are first evaluated and then sent, so this send will not match with anything (and is syntactically incorrect).

Remember, however, that an arithmetic operation that is in a pattern within a pattern is not evaluated when sent over a channel, and must be matched exactly. For example,

```
@Nil!( for( x <- @{5 + 7}){Nil} )
```

can only match with something that preserves the `5 + 7` intact, such as in the process `for( @for( x <- @{5 + 7}){z} <- @Nil ){ @z!(Nil) }`.

Finally, remember that we cannot bind a free variable to any process or name containing free variables, or containing any out-of-context uses of logical connectives, joins, etc. In particular, it is syntactically incorrect to write:

```
match
match @Nil!(Nil) { x => {@Nil!(x)} }
{
match @Nil!(Nil) { y } => { y }
}
```

since, if it did match, `y` would match to `x => {@Nil!(x)}`, which is neither a process nor a name.