Mixfix Operators

A type name, function name, or constructor name can comprise one or more name parts if we separate them with underscore characters `_`, and the resulting name can be used as an operator. From left to right, each argument goes in the place of each underscore `_`.

For instance, we can join with underscores the name parts `if`, `then`, and `else` into a single name `if_then_else_`. The application of the function name `if_then_else_` to some arguments named `x`, `y`, and `z` can still be written as:

- a standard application by using the full name `if_then_else_ x y z`
- an operator application by placing the arguments between the name parts `if x then y else z`, leaving a space between arguments and part names
- other *sections* of the full name, for instance leaving one or two underscores:
  - `(if_then y else z) x`
  - `(if x then_else z) y`
  - `if x then y else_ z`
  - `if x then_else_ y z`
  - `if_then y else_ x z`
  - `(if_then_else z) x y`

Examples of type names, function names, and constructor names as mixfix operators:

```
-- Example type name _⇒_
_⇒_   : Bool → Bool → Bool
true  ⇒ b = b
false ⇒ _ = true

-- Example function name _and_
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false

-- Example function name if_then_else_
if_then_else_ : {A : Set} → Bool → A → A → A
if true then x else y = x
if false then x else y = y

-- Example constructor name _::_
data List (A : Set) : Set where
  nil  : List A
  _::_ : A → List A → List A
```

Precedence

Consider the expression `false and true ⇒ false`. Depending on which of `_and_` and `_⇒_` has more precedence, it can either be read as `(false and true) ⇒ false` (which is `true`), or as `false and (true ⇒ false)` (which is `false`).

Each operator is associated to a precedence, which is a floating point number (can be negative and fractional!). The default precedence for an operator is 20.

> **Note**
>
> Please note that `->` is directly handled in the parser. As a result, the precedence of `->` is lower than any precedence you may declare with `infixl` and `infixr`.

If we give `_and_` more precedence than `_⇒_`, then we will get the first result:

```
infix 30 _and_
-- infix 20 _⇒_ (default)

p-and : {x y z : Bool} →  x and y ⇒ z  ≡  (x and y) ⇒ z
p-and = refl

e-and : false and true ⇒ false  ≡  true
e-and = refl
```

But, if we declare a new operator `_and'_` and give it less precedence than `_⇒_`, then we will get the second result:

```
_and'_ : Bool → Bool → Bool
_and'_ = _and_
infix 15 _and'_
-- infix 20 _⇒_ (default)

p-⇒ : {x y z : Bool} →  x and' y ⇒ z  ≡  x and' (y ⇒ z)
p-⇒ = refl

e-⇒ : false and' true ⇒ false  ≡  false
e-⇒ = refl
```

Fixities can be changed when importing with a `renaming` directive:

```
open M using (_•_)
open M renaming (_•_ to infixl 10 _*_)
```

This code brings two instances of the operator `_•_` in scope:

- the first named `_•_` and with its original fixity
- the second named `_*_` and with the fixity changed to act like a left associative operator of precedence 10.

Associativity

Consider the expression `true ⇒ false ⇒ false`. Depending on whether `_⇒_` associates to the left or to the right, it can be read as `(false ⇒ true) ⇒ false = false`, or `false ⇒ (true ⇒ false) = true`, respectively.

If we declare an operator `_⇒_` as `infixr`, it will associate to the right:

```
infixr 20 _⇒_

p-right : {x y z : Bool} →  x ⇒ y ⇒ z  ≡  x ⇒ (y ⇒ z)
p-right = refl

e-right : false ⇒ true ⇒ false  ≡  true
e-right = refl
```

If we declare an operator `_⇒'_` as `infixl`, it will associate to the left:

```
infixl 20 _⇒'_

_⇒'_ : Bool → Bool → Bool
_⇒'_ = _⇒_

p-left : {x y z : Bool} →  x ⇒' y ⇒' z  ≡  (x ⇒' y) ⇒' z
p-left = refl

e-left : false ⇒' true ⇒' false  ≡  false
e-left = refl
```

Ambiguity and Scope

If you have not yet declared the fixity of an operator, Agda will complain if you try to use ambiguously:

```
e-ambiguous : Bool
e-ambiguous = true ⇒ true ⇒ true
```

```
Could not parse the application true ⇒ true ⇒ true
Operators used in the grammar:
  ⇒ (infix operator, level 20)
```

Fixity declarations may appear anywhere in a module that other declarations may appear. They then apply to the entire scope in which they appear (i.e. before and after, but not outside).

Operators in telescopes

Agda does not yet support declaring the fixity of operators declared in telescopes, see *Issue #1235 <https://github.com/agda/agda/issues/1235>*.

However, the following hack currently works:

```
module _ {A : Set} (_+_ : A → A → A) (let infixl 5 _+_; _+_ = _+_) where
```