/*

```
*/
import cloud.greentea.{*,given}
import cats.{~> as -->, *}
import cats.data._
/*
```

Prenormes

@author Sylvain Ductor, 2022

Déclaration

Une prénorme est un constructeur de normes. Une prénorme est construite en annotant une fonction Scala. Voici différentes déclarations équivalentes en Scala :

```
*/
def increment_def(a : Int) : Int   = a + 1

val increment_verbose : Int => Int = (a : Int) => a + 1

val increment_compact : Int => Int = (_ : Int) + 1
/*
```

Une prénorme est déclarée en annotant une telle fonction Scala avec l'indication du type d'entrée et de retour de la norme :

```
*/
val addOne = ( (a : Int) => a + 1 ).~>[Int,Int]

val addOne_fromDef = increment_fun.~>[Int,Int]

val addOne_fromVerbose = increment_verbose.~>[Int,Int]

val addOne_fromCompact = increment_compact.~>[Int,Int]
/*
```

Le type de retour est automatiquement calculé. Il est possible de le spécifier explicitement, mais ceci est plus laborieux pour le programmeur. Cependant, s'il le spécifie il peut choisir de ne pas fournir l'annotation:

```
*/
val addOne_exp  : (Int ~> Int)[Int => Int]    = (_ : Int) + 1
/*
```

Le type d'une prénorme se décompose en deux parties :

- `(Int => Int)` est le type d'entrée/sortie attendu de la future norme,
- `[Int => Int]` est le type de la fonction Scala supportant la norme.

Assemblage

Des prénormes peuvent être assemblées en prénormes à l'aide des opérateurs *andThen*, *and*, *ior*, *xor*.

```
*/
val addTwo              = addOne andThen addOne

val addThree            = addOne andThen addOne andThen addOne

val addOne2Couple       = addOne and addOne

val addOne2Triple       = addOne and addOne and addOne

val addOneORAddThree    = addOne xor addThree

val addOneOrAddTwo      = addOne ior (addOne andThen addOne)
/*
```

- *andThen* va séquencer le calcul : "addThree" va incrémenter une fois, *puis* une deuxième fois *puis* une troisième fois.
- *and* parallélise le calcul : les types d'entrées et de sorties de "addOne2Couple" sont des couples d'entiers. Le calcul utilisera toujours les capacités de parallélisation fournies par l'interpréteur
- *xor* branche le calcul : il permet à l'utilisateur d'un contrat de choisir entre une branche de gauche et de droite.
- *ior* se comporte tantôt comme *xor* tantôt comme *and* selon l'entrée fournie

Construction de normes

Pour obtenir une norme, il faut spécifier un interpréteur et une modalité. Trois méthodes existent pour spécifier un interpréteur. Nous présentons ici la plus propice à l'utilisation en console et les autres au fil des tutos.

```
*/
given ξ : interpretor.default = interpretor.default()
/*
```

En Scala, le mot clé `given` permet d'intéragir avec le compilateur. Il s'agit en quelque sorte du pattern singleton au niveau du type. Ici, on déclare au compilateur un objet par défaut de type `: interpretor.default`, instancié par l'appel au constructeur `interpretor.default()` (`interpretor.default` est un interpreteur qui se veut un juste compromis entre minimalisme et

efficacité). On pourra optionnellement se référer à cet objet à l'aide de la variable `ξ` (chi), même si cela a peu de chance de nous être utile. Les méthodes que nous verrons ci-dessous nécessite un interpréteur qu'elles vont requérir au compilateur. Si deux interpréteurs différents ont été chargés avec `given` le compilateur détectera une ambiguité et refusera de compiler. Ainsi l'approche par le `given`, bien que la plus ergonomique, nous impose un unique interpreteur durant toute la session.

Une fois l'interpréteur accessible, il faut exprimer la modalité désirée. Imaginons que celle-ci soit `Option`. Une norme est déclarée avec :

```
*/
val addOne_normeContingente  = addOne.|[Option]
/*
```

Félicitation vous venez de déclarer votre première norme!

Considérons la signature de la fonction scala, les types d'entrées/sorties déclarés par la prénorme, le type de l'interpréteur fourni et le type de la modalité demandé. Si la combinaison de ceux-ci n'est pas cohérente le compilateur refusera de compiler. De plus, les types des contextes d'entrées et de sorties (γ, λ) sont automatiquement calculés par le compilateur comme tout ce qui à avoir avec la logique.

Nous fournissons quelques alias:

- Nécessité : `.yes` pour `.|[Id]`
- Possibilité : `.often` pour `.|[Validated[Throwable,?]]`
- Contingence : `.maybe` pour `.|[Option]`
- Impossibilité : `.no` pour `.|[None]`

Exécution

C'est à son exécution que le canal d'agents est déclaré. Plusieurs routines ont été définies pour faciliter l'appel d'une norme. Nous présentons ici `execWith` qui ne considère pas de canal d'agent. Les autres seront introduites dans le tuto sur les holons.

```
*/
val addOne_simple  = addOne.yes.execWith(1)

val addOne_couple  = addOne2Couple.yes.execWith((1,2))

val addOne_xor     = addOneORAddThree.yes.execWith(Left(1))// (1.left)

val addOne_ior     = addOneOrAddTwo.yes.execWith(Ior.Right(2)) //(2.right)

val addOne_iorB     = addOneOrAddTwo.yes.execWith(Ior.Both(1,2))

val addOne_andthen = addThree.yes.execWith(1)
/*
```

Une fois l'entrée chargée, les valeurs ci-dessus sont virtuellement des résultats encapsulés dans leur environnement d'exécution. `getResult()` va *effectivement* effectuer le calcul et en extraire le résultat.

```
*/
@main
def run_tuto_prenorme = println(addOne_simple.getResult())
/*
```

*/