Le langage

Tout d'abord il est intéressant de comprendre comment fonctionne le langage Nix car c'est vraiment la base de tout le reste. On va donc passer en revue les bases nécessaires de la programmation fonctionnelle et la syntaxe de Nix.

On va d'abord passer en revue toutes les bases du langage, puis on va voir la fonction `derivation`, c'est là que ça va devenir vraiment intéressant.

> **Tip** : Utilisez la commande `nix repl` pour tester la syntaxe de Nix et les exemples donnés ici.

Types de données

Voici la liste des types primitifs et comment ils sont représentés en Nix :

Type	Description	Exemple
String	Chaine de caractère	`''hello world''` ou `"hello world"`
Nombre	Nombre entier ou non	`123` ou `12.4`
Chemin	Chemin vers un fichier ou dossier (doit contenir un `/`)	`./builder.sh` ou `/bin/bash`
Booléen	Vrai ou faux	`true` ou `false`
Null	Rien du tout	`null`

Ensuite voici les autres types :

Type	Description	Exemple
Liste	Liste d'expressions Nix (notez qu'il n'y a pas de `,` mais seulement un espace)	`[ 123 ./foo.bar "abc" ]`
Ensemble d'attributs	Association de clé-valeur	`{ x = 123; y = 56; file = ./builder.sh; name = "something"; }`, `{ x = 123; y = 56; }.x` retourne 123

Les différentes expressions

- Les commentaires se font soit sur une ligne avec `#`, soit avec `/* */`

```
nix-repl> # hello world
        /* hello world */
        1 + 1
2
```

- Ensemble d'attributs récursifs qui permettent de référencer d'autres attributs du même set depuis le set lui-même

```
# x référence y, et y référence 123
rec {
  x = y;
  y = 123;
}.x # retourne 123
```

- Expressions "let-in" qui permet de définir des variables ou fonctions locales qui sont ensuite utilisées dans le "in"

```
# Le let défini les variables locales x et y
let
  x = "foo";
  y = "bar";

# Qui sont ensuite utilisée dans le in, ce qui donne donc "foobar"
in x + y
```

- L'héritage des attributs avec `inherit` permet d'éviter de se répèter

```
#  Variable locale x et y
let
  x = 123;
  y = 456;

# Dans un set
in {
  inherit x y; # équivaut à dire "x = x; y = y;"
  # soit définir l'attribut "x" du nouveau set comme étant la valeur de la variable locale "x" définie dans let
```

```
} # ceci retournera donc { x = 123; y = 456; }
```

```
# Donc
let
  x = 123;
  y = 456;
in { inherit x y; }

# est la même chose que
let
  x = 123;
  y = 456;
in {
  x = x;
  y = y;
}
```

- On peut aussi créer des fonctions que l'on peut mettre dans des variables

```
# concat est le nom de la variable
# x et y sont les paramètres de la fonction
# x + y est le corps de la fonction
nix-repl> concat = x: y: x + y

# On peut appeller la fonction en donnant le nom de la fonction, puis ses attributs séparés par des espaces
nix-repl> concat "hello " "world"
"hello world"

# Mais on peut aussi répondre à moitié à la fonction (pour plus d'info, renseignez-vous sur la programmation fonctionnelle
nix-repl> half-concat = concat "hello"
nix-repl> half-concat "world"
"hello world"

# Mais on peut aussi utiliser des sets d'arguments lorsque l'on veut tous les attributs d'un coup
nix-repl> concat = { a, b }: a + b
nix-repl> concat { a = "hello "; b = "world"; }
"hello world"
```

- Les conditions ne se font que par des genre de ternaires `if then else`

```
nix-repl> if true then "hello" else "foo"
"hello"
```

- Pour empécher encore plus la redondance de code, on peut utiliser `with` qui va en quelque sorte "transformer" les attributs d'un set en variables locales

```
# Ceci va retourner { a = "foo"; b = "bar"; }
let
  my-set = { x = "foo"; y = "bar"; };
in
  with my-set; {
    a = x;
    b = y;
  }

# Ceci va aussi retourner { a = "foo"; b = "bar"; }
let
  my-set = { x = "foo"; y = "bar"; };
in
  {
    a = my-set.x;
    b = my-set.y;
  }
```

- Pour simplifier les choses il est aussi possible d'utiliser l'interpolation de strings

```
nix-repl> foo = "hello"
nix-repl> bar = "world"

# ...pour les chemins de fichiers ou dossier
nix-repl> ./${foo}-${bar}.nix
/home/snowcode/hello-world.nix

# ...pour les strings
nix-repl> "${foo}"
"hello"
nix-repl> ''${bar}''
"world"

# ...pour les clé des attributs
```

```
nix-repl> { ${foo} = "foo"; ${bar} = "bar"; }
{ hello = "foo"; world = "bar"; }
```

Quelques opérations

Par convention ici les set d'attributs seront appellé `attrset`, les noms des clés seront appellé `attrpath`.

Opération	Syntaxe	Retourne
Sélection d'un attribut	`attrset.attrpath`	La valeur
Test si a un certain attribut	`attrset ? attrpath`	Booléen
Concaténation de string	`string + string`	String
Concaténation de path	`path + path`	Path
Concaténation de path et string	`path + string`	Path
Concaténation de string et path	`string + path`	String
Combiner un set avec un autre (le deuxième à priorité si conflit)	`attrset1 // attrset2`	Set d'attributs
Comparaisons	`< == > <= >= !=`	Booléen
Opérateurs booléens	`|| && !`	Booléen

Revision #4
Created 8 July 2023 10:09:21 by SnowCode
Updated 10 July 2023 18:01:46 by SnowCode