Cycle ingénieur - 2ème année

Programmation fonctionnelle en Haskell

Introduction

2023 - 2024

Plan

La programmation fonctionnelle, c'est quoi ?

Historique de la programmation fonctionnelle

Caractéristiques de Haskell

La programmation fonctionnelle, c'est quoi ?

La programmation fonctionnelle, c'est quoi ?

[...] Functional code is characterized by one thing: the absence of side-effects. It doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function. Every other "functional" thing can be derived from this property. Use it a guide rope as you learn.

[...] La programmation fonctionnelle se caractérise par une chose : l'absence d'effet de bord. Elle ne s'appuie pas sur des données hors de la fonction courante et elle ne change pas de donnée existant hors de la fonction courante. Toute autre caractéristique "fonctionnelle" peut être déduite de cette propriété. Utilisez cela comme fil conducteur lors de votre apprentissage.

Mary Rose Cook, *A practical introduction to functional programming*

Prog. fonctionnelle ≡ pas d'effet de bord

~~*Variable*~~ → **Constante**
Modification d'une variable : effet de bord

~~*Instruction*~~ → **Expression** (typée)
Seul intéret d'une instruction : son effet de bord !

~~*Procédure*~~ → **Fonction pure**
Seul intéret d'une procédure : son effet de bord !

~~*Boucle*~~ → **Récursivité**
Boucle ≡ instruction avec modification de variable(s)

+ Toute fonction est une expression (typée)

Moins d'éléments, donc moins expressif ?

- 1930-1936, Alonzo CHURCH : λ-calcul
  Représentation des langages fonctionnels
- 1936-1937, Alan TURING : machines de TURING
  Représentation des machines actuelles
- 1936-1937, Alan TURING : **Équivalence des représentations**

Avantages

- Moins d'éléments
  ⇒ donc plus simple, plus intuitif
- Plus facile à analyser
  ⇒ tests unitaires simplifiés
- Plus proche de la description mathématique

Inconvénients

- Plus gourmand en ressources (en amélioration)
- Limite d'applicabilité :
  - entrées-sorties
  - génération aléatoire

Historique de la programmation fonctionnelle

Historique des langages

Langages fonctionnels

- 1958 : LISP *(LISt Processor)*
  - 1970 : Scheme
- 1973 : ML *(Meta Language)*
  - 1985 : CaML
    - 1996 : OCaML
  - 1990 : **Haskell**
- 2003 : Scala (langage JVM)
- 2005 : F# (basé sur C#)

Langages impératifs

- 1972 : C
- 1983 : Turbo Pascal
- 1985 : C++ (orienté objet)
- 1995 (orienté objet)
- 2000 : C# (orienté objet)
- 2011 : Kotlin (orienté objet)

Les langages impératifs s'y mettent !

- 2011 : C++ 11 introduit les fonctions anonymes
- 2014 : Java 8 introduit les fonctions anonymes
- 2023 : Java 21 introduit complètement le *pattern-matching*

Nativement présent dans les langages fonctionnels

- Fonctions anonymes = définies à la volée
- Fonctions passables en paramètre

  ⇔ Traitements passables en paramètre

  ⇒ Factorisation du code améliorée

Pourquoi un tel regain d'intérêt ?

Langages classiques peu adaptés aux nouveaux besoins

- Disponibilité accrue et scalabilité
- Applications distribuées et concurrentes
- Interfaces pour utilisateurs finaux et autres applications.

Donc les langages fonctionnels rapportent gros !

StackOverflow 2023 Developer Survey : Top paying technologies
(Médiane globale sur toutes les réponses, 2023)

- n°2 : Erlang, 99 492$
- n°3 : F#, 99 311$
- n°5 : Clojure, Lisp, Scala, 96 381$

Caractéristiques de Haskell

Rappel : typage statique/dynamique

Typage statique : type lié aux variables

Une fois la variable déclarée, le type ne change pas.

Exemples : Java, C++, Scala, OCaML, **Haskell**

Typage dynamique : type lié aux valeurs des variables

Le type peut changer à chaque affectation.

Exemples : Javascript, Python, Ruby, Groovy

Fiche d'identité de Haskell

Historique

- 1987, Portland (États-Unis) : Conférence FPCA (*Functional Programming and Computer Architecture*)
  Objectif : harmoniser les différentes propositions
- **Avril 1990 : Haskell 1.0**
- Février 1999 : Haskell 98
- Juillet 2010 : Haskell 2010
- Prochainement : Haskell 2020

Fonctionnalités

- Langage fonctionnel
- Typage statique fort
- Évaluation non-stricte par défaut

Environnement Haskell

Compilateur GHC

- Compilateur Haskell le plus utilisé
- Extension de fichier : `.hs`
- Créé en 1992
- Licence BSD
- Dernière version : **9.4.8** (10 novembre 2023)

Outil de *build* Stack pour des projets Haskell

- Compilation continue, tests, déploiement
- Compilation incrémentale et tests
- Gestion des dépendances (à l'aide du dépôt *Hackage*)
- Mode interactif (console) disponible
- Utilise *Cabal* (système de paquetage)

Stack : prise en main

Installation

- sur Ubuntu : `sudo apt install haskell-stack`

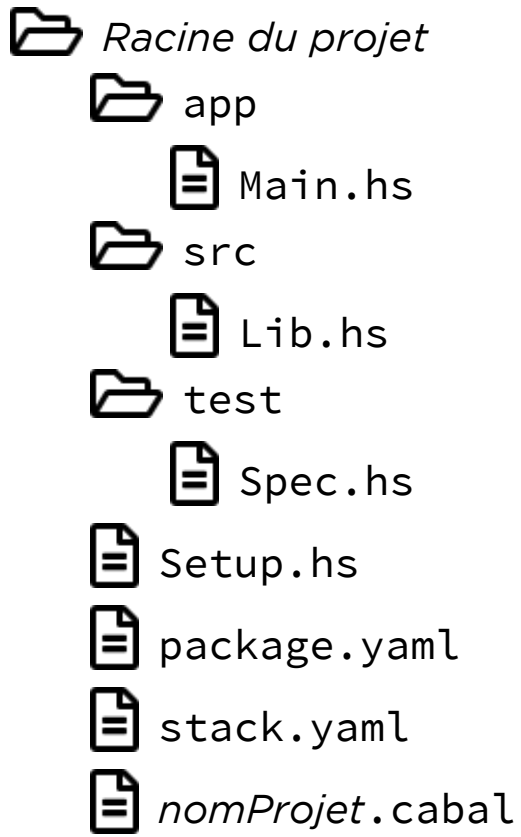
Mise à jour

- `stack upgrade` met à jour la version de Stack
- `stack update` met à jour la liste des paquetages téléchargeables (*Hackage*)

Création d'un projet : `stack new` *nomProjet*

- Crée le répertoire *nomProjet* avec son contenu

Hiérarchie d'un projet Stack vierge



Fichiers de configuration

Fichier `package.yaml`

- Format YAML (YAML isn't A Markup Language)
- Informations sur le projet (nom, version, auteur, licence, ...)
- Options du compilateur GHC globales ou par composant
- Dépendances globales ou par composant

Fichier `stack.yaml`

- Format YAML (YAML isn't A Markup Language)
- Version de GHC à utiliser
- Version de Stack à utiliser
- Dépendances non standard

Autres fichiers

Fichier *nomProjet*`.cabal`

- Ne pas modifier ce fichier
- Automatiquement regénéré en fonction des fichiers `package.yaml` et `stack.yaml`

Fichiers également créés *(non utilisés par Stack)*

- `.gitignore`, `LICENSE`, `README.md`, `CHANGELOG.md`
- uniquement utiles pour un dépôt du projet sur `git`

Stack : utilisation dans un projet

Compilation et exécution

- `stack build` compile tous les fichiers des répertoire `src` et `app`
- `stack run` lance l'exécutable
- `stack haddock` génère la documentation
- `stack test` compile et exécute tous les fichiers du répertoire `test`

Nettoyage

- `stack clean` supprime les fichiers intermédiaires
- `stack purge` supprime tous les fichiers générés

Console interactive

- `stack ghci` lance une console interactive GHCi avec les ressources du projet disponibles

Credits

- Icônes : Font Awesome *(Licence)*