Chapitre I

Introduction aux langages et à la programmation

1 Programmer

Que signifie “programmer” ?

Il s’agit évidemment de transcrire une intention humaine de calcul ou d’actions dans un langage approprié, sous la forme d’une suite d’instructions, afin qu’un ordinateur réalise ces calculs et actions.

C’est la description usuelle. Mais il faut ajouter à cela une autre intention que doit avoir le programmeur lorsqu’il écrit un programme : il faut que celui-ci soit lisible, intelligible, testable ... autant de contraintes qui relèvent de la qualité du code et qui doivent être comprises dans l’action de programmer.

Etant donné un langage de programmation, il y a toujours de très nombreuses manières d’aboutir à un programme fonctionnel. Il y en a cependant assez peu qui répondent à l’ensemble de ces contraintes de bonne pratique de programmation. Un programmeur averti utilise chaque instruction à bon escient, utilise des styles de programmation standards, utilise des schémas (design patterns) établis, spécifie au maximum chaque élément de son programme (est-ce que cet objet est constant ? est-ce que j’ai le droit de le copier ? est ce qu’il s’agit d’une redéfinition ? ...), ... et doit choisir avec soin son langage pour l’usage qu’il lui destine.

Son objectif est de ne pas laisser la possiblité à tout programmeur (un autre comme lui-même) d’utiliser son programmes ou ses éléments pour autre chose que ce pour quoi il a/ils ont été conçu-s. Un programmeur-utilisateur qui disposerait de trop de libertés dans l’usage des éléments d’un programme risquerait de provoquer crashes ou pire, comportements indéterminés.

Nous verrons que le typage des données, des objets, lorsqu’il est bien fait, réduit ces problèmes, en particulier dans les langages impératifs. Nous verrons aussi ce qu’apporte le paradigme de la programmation fonctionnelle.

Ce petit topo a pour objectif de rappeler une chose : programmer ne signifie pas juste “apprendre un paquet d’instructions et les sortir dans le bon ordre pour réaliser quelque chose de fonctionnel”, mais aussi sécuriser un programme, s’assurer qu’il termine, que chaque fonction termine, qu’il soit intelligible en tout point, ... C’est vous programmeurs qui, par la façon dont vous présentez votre code, dont vous construisez votre programme, par les algorithmes que vous implémentez, rendez explicites les intentions que vous avez !

Cependant ... cependant, vous découvrirez avec le temps qu’il y a autant de façons de programmer que d’usages. Faire du calcul scientifique ne requiert en général pas la même exigence que concevoir un logiciel doté d’IHM ou une API. Dans le calcul scientifique par exemple, le programmeur n’allouera pas un temps considérable à l’écriture d’un code ultra-sécurisé et utilisera des langages comme Python qui autorisent de prendre de libertés avec la qualité du code. Ces bonnes pratiques que nous allons commencer à vous enseigner doivent pourtant rester une force de rappel pour éviter les problèmes.

2 Langages informatiques

2.a Types et usages des langages informatiques

Il existe de très nombreux langages de programmation informatique. Il peuvent être classés selon leur `paradigme` principalement :

- **langages impératifs:** BASIC, Fortran, Pascal, C, Ada, etc.
- **langages impératifs orienté objet:** Java, C++, C#, etc.
- **langages déclaratifs et fonctionnels:** Lisp (Scheme, Racket ...), F#, Caml, Haskell, Prolog, ASP, etc.
- **langages de structuration/description:** LaTeX, HTML, XML, JSON, etc.

Programmation impérative

Un programme **impératif** est une **machine à états (mutables)**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).
‘

Dans un langage **impératif** on dit à l'ordinateur **quoi faire (what to do)**.

Dans l'exemple ci-dessous, ce programme impératif (en Python) calcule et affiche un automate cellulaire 1D. Il implémente la règle 30. Ce programme contient (volontairement) des effets de bord (nous verrons plus tard ce que c'est, quand nous aborderons la programmation fonctionnelle). Les effets de bord dans ce programme sont l'utilisation dans 2 fonctions (qui ne sont donc pas des fonctions pures) d'une variable externe aux fonctions (nb_x).

```
import numpy as np \# pour créer un tableau 2D (espace / temps)
import matplotlib.pyplot as plt \# pour l'affichage graphique
nb_x = 300 \# nombre de colonnes = x l'espace
nb_t = 100 \# nombre de lignes = t le temps
x_t = np.zeros(shape=(nb_t,nb_x),dtype=float)
x_t[0,49] = 1 \# on ne fait qu'une seule modification de la ligne de 0 : un 1 au centre
x_t[0,99] = 1 \# on ne fait qu'une seule modification de la ligne de 0 : un 1 au centre

def indice(i):
    return i%nb_x

def evol_regle30(xt_t1, xt_t2) :
    for i in range(nb_x) :
        if xt_t1[i]==1 :
            if xt_t1[indice(i-1)]==1 :
                xt_t2[i]=0
            else : \# si la valeur à droite est égale à 0
                xt_t2[i]=1
        else :# si la cellule i est à 0
            if xt_t1[indice(i-1)]+xt_t1[indice(i+1)]==1 :
                xt_t2[i]=1
            else : \# si la somme des valeurs à droite et à gauche est différente de 1
                xt_t2[i]=0


for t in range(0,nb_t-1) : \# calcule pour tous les temps
    evol_regle30(x_t[t,], x_t[t+1,])

fig = plt.figure(figsize=(15, 10))
plt.imshow( x_t , cmap = 'magma' )
plt.title( "Evolution de la règle 30" )
plt.xlabel('espace')
plt.ylabel('temps')
```

Figure I.1: Exemple de programme impératif en Python ...

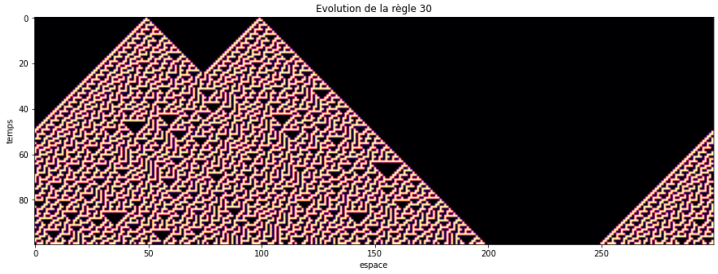

Figure I.2: ... et le résultat de son execution.

Programmation fonctionnelle

La programmation fonctionnelle repose sur la notion fondamentale de **fonction**. Dans le cours dédié, nous verrons en détail cette notion somme toutes complexe.

Un **programme fonctionnel** est dans l'idée **une énorme fonction** (un calcul) mappant un état initial (entrée) en un état de sortie (output). Il n'y a **pas de changements d'états (immuabilité)**. Le programme décrit un calcul qui est vu comme une **évaluation de fonctions**.

Dans un langage **fonctionnel** on dit à l'ordinateur **comment faire (how to do)**.

```
#lang racket

;; Nombre de colonnes (x l'espace) et de lignes (t le temps)
(define nb-x 30)
(define nb-t 10)

;; Fonction pour afficher la grille (mode texte)
(define (print-grid grid)
  (for-each
   (lambda (row)
     (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
     (newline))
   (vector->list grid)))

;; Fonction pour créer une matrice 2D initialisée à zéro
(define (create-matrix rows cols)
  (make-vector rows (make-vector cols 0)))

;; Initialiser la grille avec des valeurs spécifiques
(define (init-grid)
  (let ((grid (create-matrix nb-t nb-x)))
    (vector-set! (vector-ref grid 0) 19 1)
    (vector-set! (vector-ref grid 0) 9 1)
    grid))

;; Calcul de l'indice modulo
(define (indice i n)
  (modulo (+ i n) n))


;; Fonction pure pour calculer la prochaine ligne
;; en fonction de la règle 30
(define (evol-regle30 xt-t1)
  (define (rule30 i)
    (let ((left (vector-ref xt-t1 (indice (- i 1) nb-x)))
          (center (vector-ref xt-t1 i))
          (right (vector-ref xt-t1 (indice (+ i 1) nb-x))))
      (cond
        ((and (= center 1) (= left 1)) 0)
        ((= center 1) 1)
        ((= (+ left right) 1) 1)
        (else 0))))
  (list->vector (map rule30 (build-list nb-x values))))

;; Fonction pour calculer toutes les lignes
(define (calcule-toutes-lignes grid)
  (define (evolve grid t)
    (let ((new-line (evol-regle30 (vector-ref grid (- t 1)))))
      (vector-set! grid t new-line)
      grid))
  (foldl (lambda (t g) (evolve g t))
         grid (build-list (- nb-t 1) (lambda (x) (+ x 1)))))

;; Programme principal
(let ((x-t (init-grid)))
  (define result-grid (calcule-toutes-lignes x-t))
  (print-grid result-grid))
```

Figure I.3: Exemple de programme fonctionnel en Racket (une forme de Lisp) ...

```
(define (print-grid grid)
  (for-each
   (lambda (row)
     (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
     (newline))
   (vector->list grid)))

;; Programme principal
(let ((x-t (init-grid)))
  (define result-grid (calcule-toutes-lignes x-t))
  (print-grid result-grid))
```

```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1 1 0 0 0 0 0
0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0
0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 0
0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 0
1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 1 0
>
```

Figure I.4: ... et le résultat de son execution (affichage console seulement).

Programmation Descriptive

Il existe de très nombreux langages de description. Les langages de description comme JSON ou XML permettent de décrire des informations, basiquement des données qu'on souhaite stocker. Mais d'autres sont spécialisés dans la description de transformations (qui sont des informations) comme XSLT (qui est un langage XML) ou encore le format USD (Universal Scene Description) de Pixar.

```
from pxr import Usd, UsdGeom
stage = Usd.Stage.Open('HelloWorld.usda')
hello = stage.GetPrimAtPath('/hello')
stage.SetDefaultPrim(hello)
UsdGeom.XformCommonAPI(hello).SetTranslate((4, 5, 6))
print stage.GetRootLayer().ExportToString()
stage.GetRootLayer().Save()
refStage = Usd.Stage.CreateNew('RefExample.usda')
refSphere = refStage.OverridePrim('/refSphere')
print refStage.GetRootLayer().ExportToString()
refSphere.GetReferences().AddReference('./HelloWorld.usda')
print refStage.GetRootLayer().ExportToString()
refStage.GetRootLayer().Save()
```

Figure I.5: Exemple de description de scène en langage USD (Pixar), tiré de la documentation officielle du format USD

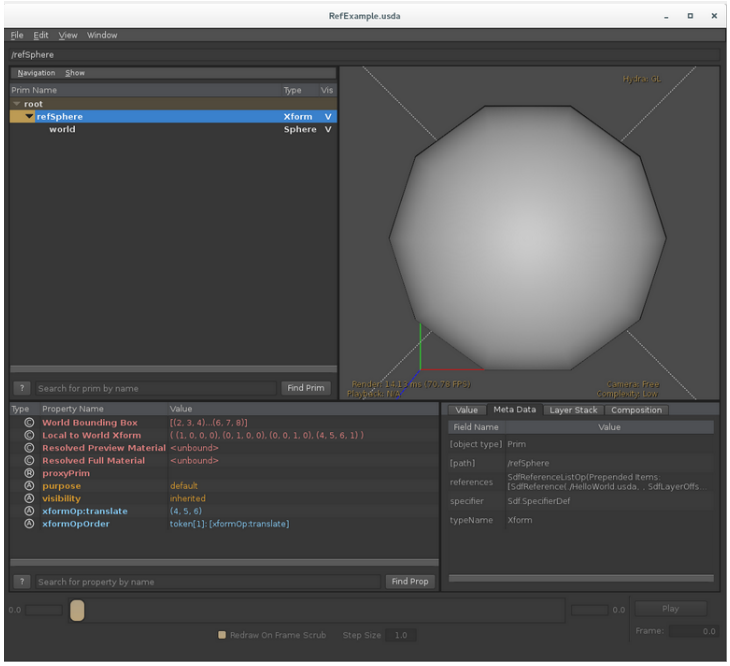

Figure I.6: ... et le résultat de son interprétation par l'API OpenUSD en Python.

Ces langages peuvent être typés (comme les instances de documents XML qui peuvent être typée par un Schema XML) ou non (comme les documents JSON qui ne sont pas contraints par un typage). Le typage de documents permet de spécifier précisément des types, d'imposer des contraintes sur les informations qui seront stockées, ne laissant pas à l'utilisateur d'ambiguité dans ce qui peut être stocké ; Cela impose également au programmeur de penser finement la façon dont il souhaite représenter les informations, donc de les formaliser au préalable. C'est ce que nous verrons dans les cours de XML qui suit ce chapitre.

2.b Caractéristiques des langages informatiques

Tout comme les langages naturels, les langages informatiques vérifient systématiquement les caractéristiques suivantes:

l'Alphabet

L'alphabet est l'ensemble des caractères qui peuvent être utilisées dans un langage (écrit ou parlé par des humains en général). Par exemple

- en Anglais : a, b, c, ..., z.
- en Français: idem qu'en Anglais, avec en plus ç, à, ê, etc.
- en Langage Machine: 0 ou 1
- en en C, C++, Visual Basic: alphabet ASCII
- en Java, Python, C#: Unicode UTF-8

L'alphabet ASCII est un code qui donne une valeur numérique à chaque lettre par exemple A vaut 65, B vaut 66 etc.L'alphabet ASCII ne peut coder que 128 caractères c'est pourquoi on peut pas représenter les lettres accentuées ou autres caractères spéciaux dans cet alphabet. Pour en savoir plus sur l'alphabet ASCII, reportez-vous à la page Wikipédia dédiée[1].

L'alphabet UTF-8 aussi appelé iso 859-1 est un alphabet qui permet d'encoder plus de caractères, notamment les caractères accentués. Reportez-vous à la page Wikipédia dédiée[2].

[1] http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange
[2] http://fr.wikipedia.org/wiki/UTF-8

le Vocabulaire

Le vocabulaire est l'ensemble des mots prédéfinis d'un langage (humain). On trouve généralement la définition de l'ensemble des mots d'un langage c'est-à-dire de son vocabulaire dans le dictionnaire. Par exemple:

- en Anglais: Oxford Dictionnary
- en Français: Le Petit Robert, Le Petit Larousse, etc.
- en Langage Machine: Assembleur (dépend du processeur)
- en Java: 48 mots prédéfinis (ex: `for`, `class`, etc.)
- en C# : 78 mots prédéfinis (cf. Mots clefs du C#)

la Syntaxe

La syntaxe est l'ensemble des règles qui indiquent comment se font les phrases. Par exemple:

- en Anglais ou en Français : *Sujet Verbe Complément.* ou *Verbe Sujet Complément ?*
- en Langage Machine: *mot + données = instruction*
- en C, C++, Java, C# ... : *mot + opérateur (=, +, -) ponctuation (;)*

la Sémantique

La sémantique est l'ensemble des règles qui indiquent quelles sont les phrases qui ont du sens. Par exemple si l'on considère les quatre phrases suivantes, elles sont toutes correctes sur le plan du vocabulaire et de la syntaxe mais deux d'entre elles n'ont aucun sens.

- Pascal lance un caillou. ==> OK
- Un caillou lance Pascal. ==> NON OK
- Il faut changer l'ampoule. ==> OK
- Je faux changer l'ampoule. ==> NON OK

La sémantique doit également être respectée dans les langages informatiques. Par exemple en C, si l'on écrit:

- `x = 5;` cela signifie que l'on met la valeur `5` dans la variable `x`.
- `5 = x;` ne veut rien dire, puisque l'on ne peut pas mettre de valeur dans le chiffre `5`.

l'organisation des phrases

Dans les langage naturels, les phrases sont organisées en paragraphes, les paragraphes sont organisés en sous-sections, sections, chapitres, tomes, volumes, etc. En langage objet comme C# ou Java par exemple les instructions sont organisées en classes, méthodes, et blocs d'instructions.