Séance 9b: Spécifications et variables globales

Université Paris-Diderot

Objectifs:

- S'assurer de la validité d'une spécification pour un programme donné.
- Apprendre à utiliser les variables globales.

Dans une première partie on se familiarise avec la notion de spécification formelle. Dans une seconde on apprend à se servir des variables globales.

Exercice 1 (Spécification erronées, $\star$)

Vous trouverez ci-dessous les spécifications des fonctions présentes dans le fichier `Specification.py`*. Malheureusement, les implémentations de ces fonctions sont erronées... Pour chacune d'elles, trouvez une entrée pour laquelle la fonction ne respecte la spécification, puis corrigez ladite fonction.*

1. *Fonction* `charAtPosition` *:*

   Spécification:
   - *Entrées : Une chaîne de caractères* `s` *et un entier* `i`*.*
   - *Sortie : Une chaîne contenant uniquement le caractère à l'indice* `i` *si* `i` *est un indice valide de* `s`*, la chaîne vide sinon.*

2. *Fonction* `minList` *:*

   Spécification:
   - *Entrée : Une liste d'entiers.*
   - *Sortie : Le plus petit élément de la liste si la liste est non vide,* `0` *sinon.*

3. *Fonction* `initList` *:*

   Spécification:
   - *Entrée : Un entier* `i`*.*
   - *Sortie : Une liste contenant tous les entiers strictement compris entre* `0` *et* `i`*.*

4. *Fonction* `dichotomicSearch` *:*

   Spécification:
   - *Entrées : Une liste d'entiers triée* `lis` *et un entier* `i`*.*
   - *Sortie : L'indice auquel se trouve l'entier* `i` *dans* `lis` *si* `i` *est présent dans* `lis`*,* `-1` *sinon.*

5. ($\star\star$) *Fonction* `forallNotEmpty` *:*

   Spécification:

— *Entrée : Une liste de chaînes de caractères* `lis`.
— *Sortie : Un booléen assurant que toutes les chaînes de caractères présentes dans* `lis` *sont bien non vides.*

6. (⋆⋆) *Fonction* `existsPositiveLine` *:*

Spécification:
— *Entrée : Une liste de listes d'entiers* `lis`.
— *Sortie : Un booléen indiquant s'il existe une ligne non vide qui ne contient que des entiers positifs.*

□

Exercice 2 (Listes de listes, ⋆⋆)

Implémentez les fonctions respectant les spécifications suivantes.

1. *Fonction* `existsUnderscore` *:*

Spécification:
— *Entrée : Une liste de chaînes de caractères* `lis`.
— *Sortie : Un booléen indiquant s'il existe une chaîne de caractères dans* `lis` *contenant un tiret bas.*

2. *Fonction* `forallContainsZero` *:*

Spécification:
— *Entrée : Une liste de listes d'entiers* `lis`.
— *Sortie : Un booléen assurant que toutes les lignes de* `lis` *contiennent chacune au moins un* `0`.

3. (⋆⋆) *Fonction* `padMatrix` *:*

Spécification:
— *Entrées : Une liste de listes d'entiers* `lis` *de taille* `n` *et un entier* `x`.
— *Sortie : Une matrice d'entiers* `mat` *de taille* $\texttt{n} \times \texttt{m}$, *où* `m` *est la taille du plus grand sous tableau de* `lis`, *et où pour tout* `i` *et* `j`, `mat[i][j] == lis[i][j]` *si* `lis[i][j]` *est bien défini,* `mat[i][j] == x` *sinon.*

□

Exercice 3 (Variables globales, ⋆⋆)

De même, implémentez les fonctions respectant les spécifications suivantes, en utilisant une ou des variables globales.

1. *Fonction* `iterNum` *:*

Spécification:
— *Entrée : aucune.*
— *Sortie : un entier, augmentant de* 1 *à chaque appel de la fonction.*
— *Effet : incrémente une variable globale servant de compteur.*

Au premier appel, `iterNum` *doit retourner* `0`, *au second appel* `1`, *...*

2. *Procédure* `reset` *:*

Spécification:
— *Entrée : un entier* `x`.
— *Effet : Le compteur utilisé par* `iterNum` *est réinitialisé à* `x`. *Le prochain appel à* `iterNum` *devra donc renvoyer* `x`.

3. *De manière similaire à* `iterNum`, *on veut écrire une fonction qui retourne un à un les éléments d'une liste fournie par l'utilisateur. Commencer par écrire une procédure* `initWithList` *satisfaisant la spécification suivante :*

Spécification:

— *Entrée : une liste d'entiers* `lis`.

— *Effet : stocke* `lis` *dans une variable globale.*

4. *Écrire une fonction* `iterList` *satisfaisant la spécification :*

Spécification:

— *Entrée : aucune.*

— *Sortie : retourne une valeur de la liste* `lis`. *précédemment fournie à* `iterList`, *en commençant par la valeur d'indice* `0`, *et à chaque appel retourne la valeur suivante. Si toutes les valeurs ont déjà été retournées, ou si* `initWithList` *n'a pas été appellé, retourne* `-1`

— *Effet : incrémente une variable globale servant de compteur.*

Contrat:

```
a = [3, 4]
initWithList(a)
print(iterList())
print(iterList())
print(iterList())
```

doit afficher

```
3
4
-1
```

□

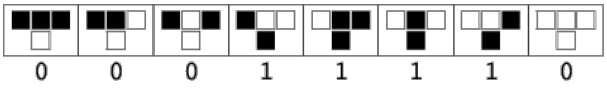
Exercice 4 (Automates cellulaires en 1D, ⋆⋆ - ⋆⋆⋆)

Un automate cellulaire est une collection de cellules "colorées" vivant sur une grille. La grille et ses cellules évoluent par étapes, où à chaque étape, l'état d'une case de la grille dépend de l'état de celle-ci et son voisinnage immédiat à l'étape précédente.

Dans cet exercice, les cellules sont noires ou blanches (leur "couleur" est donc représentée par un booléen), et vivent sur une grille uni-dimensionnelle. On se propose de réaliser un programme simulant l'évolution d'une grille, pour différentes règles d'évolution.

Dans le squelette de code fourni, on dispose de deux variables globales, `front` *et* `back`. *À chaque étape,* `front` *représente l'état courant de la grille, et le nouvel état est calculé dans* `back`. *À la fin du calcul on échange* `back` *et* `front` *; le but est d'éviter d'allouer des listes intermédiaires inutilement. La grille est représentée par une liste de booléens, indiquant à chaque case si la cellule est noire (booléen à* `True`*) ou blanche (booléen à* `False`*).*

1. *Écrire une procédure* `init`, *prenant en argument un entier* `n`, *et initialisant* `front` *et* `back` *en des listes de taille* `n`. *Toutes les cellules doivent être initialisées à* `False`, *sauf celle au milieu de la liste, qui doit être initialisée à* `True`.
2. *Écrire une procédure* `swap`, *ne prenant aucun argument, et échangeant les listes vers lesquelles pointent* `front` *et* `back` *(*`front` *devient* `back`, *et* `back` *devient* `front`*).*
3. *Écrire une procédure* `print`, *ne prenant aucun argument, et affichant sur une ligne l'état courant de la grille, stocké dans* `front`. *On utilisera les caractères* `black` *et* `white` *fournis dans le squelette.*
4. *Il s'agit maintenant d'implémenter le calcul de l'état suivant de la grille à partir de l'état courant. Pour cela, l'utilisateur fournit une règle d'évolution. Cette règle indique pour une cellule son nouvel état, pour chaque configuration possible de la cellule et de ses voisins immédiats.*
   Par exemple, la figure ci-après illustre la "règle 30". La ligne du haut correspond à un voisinage de trois cellules, et la ligne du bas indique pour chaque voisinage ce que devient la cellule centrale à l'étape suivante.

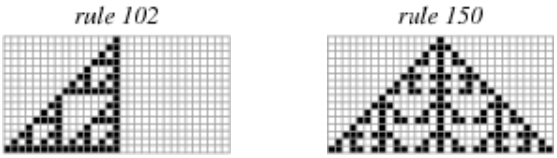


On spécifie une règle comme une liste de 8 booléens, indiquant dans l'ordre pour chaque configuration de voisinage ce que devient la cellule centrale. En fait, on remarque que par convention, (de droite à gauche) le voisinnage à l'indice i correspond à l'écriture binaire de i.
Écrire une fonction `computeIndex` *prenant en argument trois booléens, et retournant l'entier correspondant à l'écriture binaire constituée de ces trois booléens.*

> ***Contrat:***
> `computeIndex(True, True, False)` *doit retourner* `5`.

5. *Écrire une procédure* `step`*, prenant en argument une règle, sous la forme d'une liste de 8 booléens. Étant donné l'état courant de la grille dans* `front`*,* `step` *calcule le nouvel état de la grille et l'écrit dans* `back`*. On supposera que les cases en dehors de la grille sont égales à* `False`*, et on utilisera* `computeIndex`*.*
6. *Finalement, écrire une procédure* `run`*, prenant en argument un entier indiquant un nombre d'étapes, un entier indiquant la taille de la grille, et une liste de booléens indiquant une règle.* `run` *devra procéder à l'initialisation de* `front` *et* `back`*, puis effectuer autant d'étapes que demandé, en affichant l'état de la grille à chaque étape. On pensera à appeler* `swap` *après chaque appel à* `step`*.*

> ***Contrat:***
> ```
> r = [False, True, True, False, False, True, True, False]
> run(32, 64, r)
> ```
>
> *et*
>
> ```
> r = [False, True, True, False, True, False, False, True]
> run(32, 64, r)
> ```
>
> *doivent afficher respectivement des motifs similaires à :*



On pourra trouver d'autres exemples de règles amusantes sur http://mathworld.wolfram.com/ElementaryCellularAutomaton.html .

7. *(Bonus) On remarque qu'une règle n'est au final qu'une liste de booléens. On peut donc l'écrire comme l'entier dont l'écriture binaire correspond à ces 8 booléens, ce qui est plus compact. Écrire une fonction* `rule`*, prenant en argument un entier entre 0 et 255, et produisant la règle (une liste de 8 booléens) correspondante.*
8. *(Bonus) Modifier* `init` *pour pouvoir spécifier une configuration initiale quelconque.*

□