I53: Compilation et théorie des langages
TP 1

Semestre 1 2019-20

1 Bytecode Python et le module `dis`

Nous avons vu en cours que le langage Python dans ça version `CPython` n'est pas tout à fait un langage interprété puisque le code source est compilé en `bytecode`, c'est à dire en un code intermédiaire qui est ensuite exécuté sur une machine virtuelle.

Le module `dis` va nous permettre d'étudier la forme que prend ce `bytecode`. Le code suivant permet d'afficher d'afficher le `bytecode` de la fonction `hello`:

```
import dis

def hello():
   print('Hello World!')
```

```
>>> dis.dis(hello)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_CONST               1 ('Hello wolrd!')
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP
             10 LOAD_CONST               0 (None)
             13 RETURN_VALUE
```

On peut également directement accéder au `bytecode` d'une instruction comme suit:

```
>>> dis.dis("x = 3+ int('2')")
  1           0 LOAD_CONST               0 (3)
              3 LOAD_NAME                0 (int)
              6 LOAD_CONST               1 ('2')
              9 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             12 BINARY_ADD
             13 STORE_NAME               1 (x)
             16 LOAD_CONST               2 (None)
```

La machine virtuelle de Python est une machine à pile. Cela signifie que les différentes instructions consistent à empiler et dépiler des données lors de l'évaluation du programme.

1. Quels sont les noms des opérations arithmétiques élémentaires?
2. Comparer les instructions de type `if-elif-else` avec des structures sémantiquement équivalentes utilisant seulement des `if-else` imbriquées.
3. Comparer les temps d'exécution des boucles `for` et `while`. Comment justifier cette différence?
4. Étudier le fonctionnement de la compréhension de liste en Python et le comparer à une initialisation classique.

2 Compilation avec gcc

On considère le programme C suivant:

```c
#define CARRE(X) ((X)*(X))
#define CUBE(X) (CARRE(X)*(X))
#define DIX 10
/*
  Commentaire sur
  plusieurs
  lignes
*/
int main(int argc, char *argv[])
{
  int a=2,b=1,c,i;

  //Commentaire sur une seule ligne

  for (i=0; i<DIX; i++) a++;
  a = a+1;
  b = a+b;
  c = CARRE(a+b) + CUBE(a-b);

  return c;
}
```

1. Tester l'effet de l'option `-E` de `gcc`. Ajouter la bibliothèque d'entrées/sorties pour voir la différence.
2. Tester la commande `-S` (elle produit un fichier `.s`). Ajouter l'option `-O` et observer la différence.
3. Idem avec l'option `-c`.

3 Makefile

L'archive `crible.tar` contient les fichiers sources de projet d'étude de la fonction $\pi(x)$ qui compte le nombre de nombres premiers inférieurs à x. Le but de d'écrire un `makefile` permettant de compiler le compte rendu du projet.

1. Écrire une cible permettant de compiler une exécutable `crible.exe` à partir du fichier `crible.c`.
2. Écrire une cible qui crée un fichier `data.dat` contenant sur deux colonnes les cordonnées des points du graphes de la fonction π obtenus après avoir exécuter la commande `./crible.exe -p100 -l10`.
3. Écrire une cible qui crée le fichier `pi.png`.
4. Le fichier `timing.png` est obtenu en traçant une courbe à partir des résultats de la commande `./crible.exe -t -l <n>` où `n` doit varier de 10 à 28.
5. Le fichier `timingOx.png` est obtenu en traçant 3 courbes correspondant à l'expérimentation précédente mais en compilant l'exécutable avec les option `-O1`,`-O2` et `-O3`.
6. Le compte rendu final est compilé à l'aide de la commande `pdflatex compte_rendu.tex`.