Dépôt BLIH: PYjour04
Répertoire de rendu : racine du dépôt.
docs.python.org est ton ami.

Cnorm, les étudiants de lionel en mode dépression

Installer le cnorm : pip install cnorm

Lire le tuto du cnorm :

> http://pythonhosted.org/cnorm/tutorial.html

L'AST complet du C se trouve décrit dans **cnorm.nodes** et peut se voir plus simplement ici :

> http://pythonhosted.org/cnorm/nodes.html

Des instanciations à la main sont présentées dans **tests/internal_cnorm.py** sur:

> https://github.com/LionelAuroux/cnorm/blob/master/tests/internal_cnorm.py

Une vision d'un générateur de code est présents dans **passes/to_c.py**. On ne peut pas s'éviter de connaître en partie la structure de l'AST et de la grammaire pour travailler dessus (notamment pour le KOOC). Le but de ces exercices est de vous faire manipuler, et donc apprendre, l'AST et la grammaire dans les grandes largeur.

Marvin, le bot dépressif

Faire un fichier nommé '**marvin.py**' qui utilise le cnorm pour décoder un fichier C en entrée et affiche un résumé peu engageant sur les déclarations faites dans ce fichier.

Nous envoyons directement une Decl à la fonction **marvin**. Ensuite à vous de décomposer l'AST en français. Toutefois pour faire vos tests, il peut être nécessaire de vous aider des tutoriaux.

Cette opération longue et fastidieuse va se concentrer uniquement sur certaines constructions déclaratives du C (principalement présenté ici).

Nous allons par exemple ne pas décomposer les paramètres d'une fonction ou les champs d'une structure. Il s'agit surtout d'un processus récursive qui pose juste des problèmes de mise en forme et compréhension du C.

Voici le prototype de la fonction à écrire.

def marvin(ast : Decl) → str

Voici des exemples de constructions en C qui après parsing vous serons envoyé via le paramètre '**ast**'. Suis ce que vous devez retourner (il y a un cat -e pour voir les retours à la ligne):
Chaque déclaration C constitue un appel a marvin et donc je write (pas print) le retour de la fonction pour chacune de ces lignes.

```
const int   a;$
extern int  bite : 8;$
register float    d = 42;$
auto double b;$
unsigned short zz1;$
unsigned long zz2;$
unsigned long long zz3;$
int   *pi;$
static int  aaa;$
extern short      bbbb;$
void **buf;$
unsigned buffer[sizeof(struct s) * 512];$
float bigbuffer[42][32][41];$
char * const toto;$
volatile char     *volatile*const* theStr;$
$
int  (*callback)(int, char **);$
```

```
void  tutu(void *, ...);$
int  (*const*volatile*tululu[23])(int, char **);$
$
typedef double GG;$
GG    boulon;$
typedef int  (*entry_point)(int, char **);$
enum E {A,B = 5 * sizeof(int),C = 4,D} F; $
struct _list { void *data; int bla : 4; struct _list *next;};$
union A {int a; double d;};$
union W {int a; double d;} V;$
inline      int   max(int x, int y) {return (x < y)? y : x;}$
$
struct {int x; double y;} *carnage(enum e, double*[]);$
$
int concat_fopen (char *s1, char *s2, char *mode);$
```

Liste des retours de la fonction marvin :

```
je definie! a est un entier constant$
je declare! bite est un champs de bit$
je definie! d est un floattant stocker dans un registre qui est initialise a une
certaine valeur mais ca me saoule$
je definie! b est un floattant double precision$
je definie! zz1 est un entier court positif ou nul$
je definie! zz2 est un entier a la jack positif ou nul$
je definie! zz3 est un entier a la super-jack positif ou nul$
je definie! pi est un pointeur sur un entier$
je definie! aaa est un entier definie localement$
je declare! bbbb est un entier court$
je definie! buf est un pointeur de pointeur sur rien$
je definie! buffer est un tableau dont la taille depend d'une expression relou a
calculer ou chaque case contient un entier positif ou nul$
je definie! bigbuffer est un tableau de taille 42 de tableau de taille 32 de
tableau de taille 41 ou chaque case contient un floattant$
je definie! toto est un pointeur constant sur un caractere$
je definie! theStr est un pointeur de pointeur constant de pointeur toujours mis
a jour lorsqu'on y accede sur un caractere toujours mis a jour lorsqu'on y
accede$
je definie! callback est un pointeur sur une fonction qui retourne un entier et
qui prends des parametres qui me saoulent$
je declare! tutu est une fonction qui retourne rien et qui prends des parametres
qui me saoulent$
je definie! tululu est un tableau de taille 23 ou chaque case contient un
pointeur de pointeur toujours mis a jour lorsqu'on y accede de pointeur constant
sur une fonction qui retourne un entier et qui prends des parametres qui me
saoulent$
je declare! GG est un type sur un floattant double precision$
je definie! boulon est de type utilisateur GG$
je declare! entry_point est un type sur un pointeur sur une fonction qui
retourne un entier et qui prends des parametres qui me saoulent$
je definie! F est un enumere E qui a 4 valeurs possibles$
je declare! tout seul dans son coin, une structure _list qui contient des champs
qui me saoulent$
je declare! tout seul dans son coin, une union A qui contient des champs qui me
saoulent$
je definie! V est une union W qui contient des champs qui me saoulent$
je definie! max est une fonction qui retourne un entier et dont le code est
integrer a l'appelant et qui prends des parametres qui me saoulent$
je declare! carnage est une fonction qui retourne un pointeur sur une structure
qui contient des champs qui me saoulent et qui prends des parametres qui me
saoulent$
je declare! concat_fopen est une fonction qui retourne un entier et qui prends
des parametres qui me saoulent$
```

Attention les espaces, les retour chariots et les fautes d'orthographe sont à respecter.
Le fichier '.c' en exemple contient tous les cas que vous devez gérer, ni plus ni moins. La note sera en fonction du nombre de cas gérés.

Jdr, le lionel féstif:)

Dans un sous répertoire **jdr**

modules standard autorisé : tous

Avec le sujet, vous est donné le fichier **jdr.py** qu'il va falloir compléter. Notre objectif est de simuler des dés de jeu de rôle, et notamment pour gérer le jet de poignée de dés bizarre (de 4 à 20 faces). Pour comprendre ce que l'on vous donne, il faudra regarder le module python **random**, la **surcharge d'opérateur**, la surcharge de la fonction **repr** et les fonctions génératrices(statement **yield**) et enfin les **lambda**.

1) Vous devez compléter la classe Pool, et les classes dés (en s'inspirant du D4 fournit dans le jdr.py de l'intra), D6,D8,D10,D12,D20. Ainsi que les instances standard d4, d6, d8, ...d20 pour tout type de dés.

Complétez le fichier jdr.py tel que (hé oui, un bout de code est aussi une spécification) :

```
from jdr import *

p1 = D6() + d6 + 1
print(repr(p1))

p2 = d6 * 4 + 12 + 4 * d8
print(repr(p2))

AbstractResult.seed(XXXXX)  # XXXXX sera donné par la correction
print(p1.throw())
print(p2.throw())
print(p1.show())
print(p2.show())

if type(p1.roll()).__name__ != 'generator':
      raise TypeError('la methode roll doit retourner un generateur')

for t in (D6() + D4()).roll():
        print(t)
```

Produira le résultat suivant :

```
D6 + D6 + 1
4D6 + 12 + 4D8
8
42
6, 1, 1
1, 6, 3, 2, 12, 4 , 1, 5, 8
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
(6, 1)
(6, 2)
(6, 3)
(6, 4)
```

- Voir surcharge des opérateurs (3.3.7. Emulating numeric types de The Python Language Reference).
- Une expression comprenant des dés est un Pool. Ex : D4 * 5 + 3
- Les opérateurs binaires demandé sont +,*, -.

  Toutefois on ne multiplie pas les dés entre eux. D6 * D10 n'a pas de sens.

  Quand on écrit 5 * D6, cela signifie juste qu'on lance 5 dés à 6 faces.
- FrozenDice est utilisé pour les valeurs constantes dans un pool de dés. Ex : D10 + 5.
- Quand on multiplie un dés par un entier c'est pour faire l'équivalent de la notation nDx des jeux de rôles, et ce, quelque soit le sens.

  Par exemple : 2 * D6()+6 s'affiche via repr par « 2D6 + 6 ».

  Ainsi D6() * 2 + 6, s'affichera de la même manière.
- On ne 'show' qu'un Pool de dés qui a été lancé (throw). Chaque dés est lancé logiquement de manière pseudo-aléatoire. La méthode show retourne la somme des dés.
- AbstractResult.seed permet de contrôler l'aléat généré par les fonctions random. Pour une seed X connu, les jets de dés seront identique entre 2 run du module (comportement standard des fonctions aléatoires).

2)

Codez la méthode `def success(self, lamb) → float` dans la classe Pool de tel manière que l'on puisse calculer la probabilité de chance qu'un throw soit un succès. La lambda lamb définit ce qu'on entends par succès. La lambda reçoit en unique paramètre un tuple représentant un jet de dés (chaque élément du tuple donne la valeur de chaque dés). La lambda sera évidement appelé par success pour toutes les solutions possibles pour ce jet de dés.

Par exemple, dans le RPG StarWars un succès c'est quand avec la somme d'un certain nombre de D6 ont dépasse un seuil. 6D6 étant une valeur de compétence correct, et un niveau de difficulté de 30 un acte héroïque...

```
proba = (D6() * 6).success(lambda x: sum(x) >= 30)
print("check %s%%" % proba)

=> check 1.9675925925925926%
```

Avec 2 % de chance de réussir on comprend mieux l'héroïsme.

3)

Dans le système World Of Darkness, on compte le nombre de fois qu'un D10 dépasse une certaine valeur par dés. Si le nombre de dés ainsi calculé dépasses un certain objectif c'est un succès.

Vous devez donc fournir une lambda stocké dans une variable de la classe Pool et nommé **darkness** qui compte le nombre de dés ≥ 5 et si le nombre total de dés remplissant ce contrat est ≥ 4 alors c'est un succès.

```
import jdr

proba = (jdr.D10() * 5).success(jdr.Pool.darkness)
print("check %s%%" % proba)

=> check 33.696%
```