Seconde-session 2021-2022: Programmation avancée

Axel Viala <axel.viala@darnuria.eu>

2 mai 2022

Nom et Prénom : Numéro étudiant :

Objectifs : La clarté des réponses sera appréciée, veillez à écrire soigneusement. Les questions portent sur le langage Rust. Notes de cours autorisé. Réponse sur une copie a part encouragée.

1 Généralités

1. Par défaut les déclarations de variables sont :
   - ◯ immutables
   - ◯ mutables
2. Toute valeur par exemple `struct A` hors type de base est :
   - ◯ Pris par réfèrence
   - ◯ Déplacé
   - ◯ Copié
3. Decrivez brievement les différentes formes de passage pour les valeurs :
   - Par deplacement *move*
   - Par référence *borrow* (mut/immutable)
   - Par copie *copy*

4. Anatomie d'un code Rust : associez les termes suivants au code suivant :

   - Opérateur d'addition
   - Nom de variable
   - Mot clef de déclaration de variable
   - Argument de fonction
   - Type
   - Mot clef de déclaration de fonction
   - Opérateur de d'enchaînement d'instruction
   - Bloc du corps de la fonction
   - Nom de fonction
   - Appel de fonction associée à un type
   - Argument de fonction passé en appel

```rust
fn foo(a: i32, b: i32) -> i32 {
    let t = a.min(b);
    t + a + b
}
```

5. Donnez une signature de fonction polymorphique/générique en Rust, expliquez succinctement l'intéret du polymorphisme.

6. Affichage et return, dans le code ci dessous, qu'afficherait le programme s'il appellait `mystere(1)` une fois ? Quelle est la valeur de retour de mystere avec cet appel ?

```rust
fn mystere(a: i32) -> i32 {
    println!("Mon mystère: {}", a);
    a + 42
}
```

7. Dans ce code, comment est passé `a`, comment est passé `b`. Ce code compile t'il ?

```rust
fn surprise(a: &mut i32, b: i32) {
    *a += b;
}
```

8. La fonction suivante peut-elle compiler ? Justifiez votre réponse.

```rust
fn mystere(a: i32) {
    match a {
        0 => 1,
        1 => 2 * a,
        n => a - a * 2,
    }
}
```

9. Rédigez le code néccessaire pour que `add_one` ait le comportement attendu dans sa documentation. Plusieurs solutions possibles. `unwrap`, `panic`, `unsafe` et `except` interdit.

```
/// Documentation de la fonction add_one
/// Ajoute la valeur derriere x, a un nombre contenu dans un Option, sinon None.
/// This function n'appelle pas `panic!()`.
fn add_one(a: Option<i32>, x: &i32) -> Option<i32> {









}
```

10. Implémentations de fonctions sur un type. On souhaite calculer la distance entre deux points via un `trait` .

```
struct Point {
    x: f32,
    y: f32,
}

trait Distance {
    fn distance(&self, other: &Self) -> f32;
}

// A vous de completer les _____

____ ___________  for __________ {

    // Completer les trou et ____.
    /// Calcule la distance entre deux `Points' via la formule
    /// `distance = sqrt((x - x') * (x - x'), (y - y') * (y - y'))`.
    /// Note: sqrt existe dans le module f32 et se nomme: sqrt()`
    fn ______(                              ) -> f32 {
        // A completer









    }
}
```

11. Écrire une fonction `simple_open` pour ouvrir un fichier sinon renvoyer une erreur, de signature :
`fn simple_open(path: &str) -> Result<File, SimpleError>`
On dispose de `pub fn open<P: AsRef<Path>>(path: P) -> Result<File>`. `path` qui peut être une chaine de caractères.
On souhaite en cas d'erreur une Erreur generique `Result::Err(SimpleError)` ou renvoyer `Ok(File)`.
indice : `match` ou `map_err`. `SimpleError` sera une unit struct.

2 Implémentation d'une machine virtuelle

Dans cette section il est proposé de réaliser une machine virtuelle a pile, aura une mémoire d'instructions representée par un `Vec<Instruction>` et une pile de travail representée par un `Vec<i16>`.

On ne **réalisera pas** la transformation du texte vers notre evaluateur. Pour cet exercice tout sera fondé sur votre énumeration `Instruction`.

La machine aura les instructions suivantes :

— Agissant sur les deux entiers au sommet de pile : Addition, Soustraction
— Pousse : Ajoute un `i16` sur la pile
— Duplique : Duplique le sommet de pile
— Supprime : Retire le sommet de pile

Votre machine virtuelle et son implémentation pourrons executer : `1 1 + duplique +` ce qui donne en décomposé : `1 1 +` soit `2` sur la pile, puis duplique `2` donc `2 2` en pile puis `+` soit `4` sur le sommet de pile.

`ExecError::PileInsuffisante` est retourné en cas d'absence de suffisament d'élements sur la pile pour une instruction. La gestion des erreurs est valorisé a la notation.

12. Réaliser une enumération `Instruction` representant les instructions décrite ci dessus.
13. Écrire une structure `VirtualMachine` qui contient notre tableau d'`Instructions`, la pile. Indice : utilisez des `Vec`, et tout champ que vous jugerez pertinent pour votre implémentation.
14. Écrire une enumération `ExecError` qui contient un unique variant `PileInsuffisante`.
15. Écrire les fonctions `new`, `eval` associées à la structure `VirtualMachine`.
    — `new` permet de construire une machine virtuelle avec une pile d'instructions passée par déplacement. La pile de travail sera initialisée vide.
    — `eval` fonction qui execute les instructions dans la pile d'instructions. Cette fonction renvoie un `Result<(), ExecError>`.