4_Control flow_fr

May 2, 2023

1 Contrôle de flot d'exéction

Les constructions les plus communes pour le contrôle du flot d'excéution en Rust sont les expression **if** et **loop**.

1.1 Expressions *if*

[ ]:
```
fn main() {
    let number = 3;

    if number < 5 { // les conditions sont parfois appelées 'branches' et␣
 ↪doivent être booléennes
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
main();
```

[ ]:
```
fn main() {
    let number = 3;

    if number { // la condition doit être booléenne !
        println!("number was three");
    }
}
main();
```

A l'inverse d'autres langages comme Ruby et Javascript, Rust ne vas pas automatiquement convertir en booléen les expressions non-booléennes utilisées en condition.

1.2 Conditions multiples avec *else if*

[ ]:
```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
```

```
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
main();
```

Notez que bien que 6 soit également divisible par 2, ce résultat n'est pas affiché, pas plus que le dernier message que le nombre n'est pas divisible pas 4, 3 ou 2. En Rust, seul le premier block dont la condition est vraie est exécuté. Une fois celui-ci trouvé, les autres conditions ne sont pas évaluées.

1.3 Utilisation de *if* avec *let*

[ ]:
```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("La valeur du nombre est: {}", number);
}
main();
```

[ ]:
```
fn main() {
    let condition = true;

    let number = if condition {
        5   // le résultat est un nombre entier
    } else {
        "six" // mauvais type
    };

    println!("La valeur du nombre est: {}", number);
}
main();
```

Rust a besoin de connaitre à la compilation le type des variables; en particulier quel est le type de la variable `number`dans cet exemple. Rust ne peut pas faire cela à l'exécution car le compilateur ne pourrait alors pas fournir les mêmes garanties s'il devait prendre en compte de multiples hypothèses sur le typage éventuel des variables.

2 Boucles

2.1 Répétition avec *loop*

[ ]:
```
fn main() {
    loop {  // type ^C to stop the loop
        println!("again!");
    }
}
main();

// ATTENTION, n'exécutez pas cette cellule qui est une boucle infinie.
// si vous le faites, il est nécessaire de redémarrer le kernel Rust (menu␣
 ↪Kernel/restart)
```

Une des utilisations des boucles est de ré-essayer une opération que l'on sait pouvoir échouer, par exemple vérifier qu'un thread a bien terminé son travail. Cependant, il est parfois possible de passer le résultat de cette opération au reste du code. Si vous ajoutez la valeur à retourner de la boucle après le mot clé `break`, elle correspondra à la valeur d'évaluation de la boucle et pourra ainsi être utilisée.

[ ]:
```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("Le résultat est {}", result);
}
main();
```

2.2 Boucles conditionnelles avec *while*

[ ]:
```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!");
```

```
}
main();
```

[ ]:
```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("la valeur est: {}", a[index]);

        index = index + 1;
    }
}
main();
```

Cette manière d'écrire est très susceptible de créer des erreurs à l'excution si la valeur de `index` est incorrecte. Elle est aussi assez lente parce que le compilateur ajoute du code de vérification des bornes, exécuté à chaque itération de la boucle. Une approche plus concise et idiomatique consiste à écrire une boucle pour chaque élément de la collection en utilisant un itérateur de la façon suivante :

[ ]:
```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("La valeur est: {}", element);
    }
}
main();
```

[ ]:
```
fn main() {
    for number in (1..4).rev() { // Notez la définition d'une séquence avec␣
 ↪l'opérateur ..
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
main();
```

3 Exercises

Exercise 1: Implémentez le jeu fizz-buzz qui compte jusqu'à 20.

Pour chaque nombre, affichez : * `Fizz` si le nombre est divisible par 3 * `Buzz` si le nombre est divisible par 5 * `Fizz Buzz` si le nombre est divisible par 15 * Le nombre lui-même dans tous les autres cas

[ ]:
```
fn main() {
    // TO DO Implement the fizz buzz game

}
main();
```

Solution:

[ ]:
```
fn main() {
    for number in 1..21 {
        if number % 3 == 0 {
            if number % 5 == 0 {
                println!("Fizz Buzz");
            } else {
                println!("Fizz");
            }
        } else if number % 5 == 0 {
            println!("Buzz");
        }
        else {
            println!("{}", number);
        }
    }
}
main();
```

Exercice 2 : Fibonacci * Implémenter une version récursive d'une fonction culculant la valeur du rang n de suite de Fibonacci * Implémenter une version itérative * Calculer un tableau avec les 50 premiers éléments de la suite

[ ]:
```
fn main() {

}
main();
```

Solution:

[ ]:
```
fn fibonacci_recursive(n: u32) -> u32 {
    if n == 0 {
        0
    } else if n == 1 {
        1
    } else {
        fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
    }
}

fn fibonacci_iterative(n: u32) -> u32 {
```

```
    if n == 0 {
        return 0;
    }
    let mut n0 = 0;
    let mut n1 = 1;
    for _ in 2..(n+1) {
        let next = n0 + n1;
        n0 = n1;
        n1 = next;
    }
    n1
}

fn fibonacci_firsts() -> [u32 ; 10] {
    let mut result = [0 ; 10];
    result[1] = 1;
    for i in 2..10 {
        result[i] = result[i-2] + result[i-1];
    }
    result
}

fn main() {
    println!("fibonacci_recursive(10) = {}", fibonacci_recursive(10));
    println!("fibonacci_iterative(10) = {}", fibonacci_iterative(10));
    println!("fibonacci_firsts = {:?}", fibonacci_firsts());
}

main();
```