Що таке функціональне програмування?

Якщо ви запитаєте середньостатистичного програміста, що таке функціональне програмування, ви можете отримати будь-яку з таких відповідей:

- Це програмування за допомогою функцій;
- Функції — це елементи «першого класу»;
- Це програмування із прозорістю посилань;
- Це стиль програмування, заснований на лямбда-обчисленні.

Хоча ці твердження можуть виявитися правдивими, вони не є особливо корисними. Я думаю, що краща відповідь буде такою:

Це програмування без операторів присвоювання.

Можливо, це визначення не здається вам набагато кращим. Можливо, воно вас навіть лякає. Зрештою, яке відношення оператори присвоювання мають до функцій, і як узагалі можна програмувати без них?

Хороші запитання. Саме на них я маю намір відповісти в цьому розділі.

Розглянемо наступну просту програму на C:

```
int main(int ac, char** av) {
   while(!done())
      doSomething();
}
```

Ця програма є основним циклом практично кожної будь-коли написаної програми. Вона буквально говорить: «Роби щось, поки не закінчиш». Навіть більше, у цій програмі немає видимих операторів присвоювання. Чи є вона функціональною? І якщо так, то чи означає це, що кожна будь-коли написана програма є функціональною?

Змусимо цю функцію щось робити. Нехай вона обчислює суму квадратів перших десяти цілих чисел [1..10]:

```
int n=1;
int sum=0;
int done() {
   return n>10;
}
```

```
void doSomething() {
   sum+=n*n;
   ++n;
}

void sumFirstTenSquares() {
   while(!done())
      doSomething();
}
```

Ця програма не є функціональною, тому що вона використовує два оператори присвоювання у функції doSomething. Крім того, ці дві глобальні змінні виглядають просто потворно. Тож покращимо її:

```
int sumFirstTenSquares() {
   int sum=0;
   int i=1;
loop:
   if (i>10)
      return sum;
   sum+=i*i;
   i++;
   goto loop;
}
```

Це вже краще; дві глобальні змінні стали локальними. Але це все ще не функціонально. Можливо, вас турбує цей goto. На те є вагома причина. Потерпіть, поки ми розглядаємо цю невелику модифікацію, що використовує робочу функцію для перетворення локальних змінних на аргументи функції:

```
int sumFirstTenSquaresHelper(int sum, int i) {
loop:
   if (i>10)
      return sum;
   sum+=i*i;
   i++;
   goto loop;
}

int sumFirstTenSquares() {
   return sumFirstTenSquaresHelper(0, 1);
}
```

Ця програма ще не функціонує, але це важлива *віха*, до якої ми ще повернемося. Але зараз, з останньою зміною, відбувається щось магічне:

```
int sumFirstTenSquaresHelper(int sum, int i) {
   if (i>10)
      return sum;
   return sumFirstTenSquaresHelper(sum+i*i, i+1);
}

int sumFirstTenSquares() {
   return sumFirstTenSquaresHelper(0, 1);
}
```

Усі оператори присвоювання зникли, і програма працює. Вона також рекурсивна. І це не випадково. Якщо ви хочете позбутися операторів присвоювання, ви *повинні* використовувати рекурсію. Рекурсія дозволяє замінити присвоювання локальних змінних на *ініціалізацію* аргументів функції.

Це дійсно спалює багато місця на стеку. Однак є невеликий трюк, що допоможе розв'язати цю проблему.

Зверніть увагу, що останній виклик `sumFirstTenSquaresHelper` є також останнім використанням `sum` та `i` у цій функції. Тримати ці дві змінні у стеку після ініціалізації двох аргументів рекурсивного виклику безглуздо; вони ніколи не будуть використані. А що як замість того, щоби створювати новий кадр стеку *(stack frame)* для рекурсивного виклику, ми просто повторно використаємо поточний кадр стеку, повернувшись до початку функції за допомогою `goto`, як ми це зробили у *віховій* програмі?

Цей симпатичний маленький трюк називається *оптимізацією хвостових викликів (tail call optimization, TCO)*, і всі функціональні мови використовують його[1].

Зверніть увагу, що TCO фактично перетворює цю останню програму на програму *emany*. Останні три рядки `sumFirstTenSquaresHelper` у проміжній програмі є, по суті, рекурсивним викликом функції. Чи означає це, що *проміжна* програма також є функціональною? Ні, вона просто поводиться ідентично. На рівні вихідного коду ця програма не є функціональною, тому що в ній є оператори присвоювання. Але якщо ми зробимо крок назад і проігноруємо той факт, що локальні змінні змінилися, а не були відновлені в новому кадрі стеку, то програма поводиться як функціональна.

[1] Так чи інакше. Віртуальна машина Java (JVM) трохи ускладнює розрахунок TCO. C, звичайно ж, не робить TCO, і тому всі мої рекурсивні приклади на C будуть збільшувати стек.

Як ми дізнаємося з наступного розділу, це не є відмінністю без відмінності. А поки що просто пам'ятайте, що коли ви використовуєте рекурсію для усунення операторів присвоювання, ви не обов'язково витрачаєте багато місця у стеку. Мова, яку ви використовуєте, майже напевно використовує TCO.

Проблема із присвоюванням

Спочатку визначимо, що ми маємо на увазі під *присвоюванням*. Присвоювання значення змінній *змінює* початкове значення змінної на щойно присвоєне. Саме ця зміна і є присвоюванням.

У C ми ініціалізуємо змінну у такий спосіб:

```
int x=0;
```

Але присвоюємо змінну так:

```
x=1;
```

У першому випадку змінна x починає існувати зі значенням 0; до ініціалізації змінної x не існувало. У другому випадку значення x змінюється на 1. Це може здатися несуттєвим, але наслідки є глибокими.

У першому випадку ми не знаємо, чи є x насправді змінною. Це може бути константа. У другому випадку сумнівів немає. Ми змінюємо x, присвоюючи їй нове значення. Ось чому можна казати, що функціональне програмування — це програмування *без змінних*. Значення у функціональних програмах *не змінюються*.

Чому це бажано? Подумайте ось про що:

```
.
//Block A
.
x=1;
.
//Block B
.
```

Стан системи під час виконання `Block A` відрізняється від стану системи у `Block B`. Це означає, що `Block A` має виконуватися *перед* `Block B`. Якщо поміняти місцями ці два блоки, то система, найімовірніше, не буде виконуватися коректно.