МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

Лекція 13

Гібридізація мов програмування. Мультипарадигменна мова Ocaml.

Філософські роздуми про гібридизацію мов програмування

Нам потрібні гібридні мови програмування, що поєднують як функціональні, так і об'єктно-орієнтовані функції. І нам потрібно навчитися ефективно їх використовувати.

Нам потрібно навчитися ефективно використовувати мови з багатьма парадигмами, які підтримують функціональну, об’єктно-орієнтовану та процедурну парадигми.
Раніше світ програмування був розділений на функціональні мови, об’єктно-орієнтовані мови та все інше (переважно процедурні мови).

Одним з них «був» функціональний програміст (принаймні як хобі), який писав «Лісп», «Хаскелл» чи «Ерланг»; або хтось «був» програмістом ОО (принаймні професійно), писав код на Java або C ++. (Ніколи не називали себе "процедурним програмістом"; коли ці імена втекли з академічного середовища в 1990-х, називати себе "процедурним програмістом" було б подібним до того, щоб носити широкі краватки та джинси з дзвоном).

Але цей світ змінювався. За останні два десятиліття ми спостерігали зростання гібридних мов програмування, що поєднують як функціональні, так і об’єктно-орієнтовані функції. Деякі з цих мов (наприклад, Scala) з самого початку мали багатопарадигму. Інші, такі як Python (при переході від Python 2 до 3) або Java (із введенням Lambdas в Java 8) є об'єктно-орієнтованими або процедурними мовами, до яких додані функціональні можливості. Незважаючи на те, що ми розглядаємо C ++ як об'єктно-орієнтовану мову, вона також була мультипарадигмою з самого початку. Почалося з C, процедурної мови, та додані об’єктно-орієнтовані функції.

Пізніше, починаючи зі стандартної бібліотеки шаблонів, на C + + вплинуло багато ідей від Scheme, нащадка LISP. JavaScript також зазнав сильного впливу Scheme і популяризував ідею анонімних функцій та функцій як об'єктів першого класу. І JavaScript з самого початку був об’єктно-орієнтованим, з об’єктною моделлю та синтаксисом на основі прототипів (хоча і не семантикою), які поступово еволюціонували, щоб стати схожими на Java.

Ми також спостерігали зростання мов, що поєднують статичну та динамічну набір тексту (TypeScript у світі JavaScript; додавання необов'язкового натяку на тип у Python 3.5; Rust має деякі обмежені можливості динамічного набору тексту). Друк - це інший вимір у просторі парадигми. Динамічне введення тексту призводить до мов, які роблять програмування цікавим і де легко бути продуктивним, тоді як суворе введення значно спрощує створення, розуміння та налагодження великих систем. Завжди було легко знайти людей, які вихваляють динамічні мови, але, за винятком кількох років наприкінці 00-х, динамічно-статична парадигматика не привертала стільки уваги.

Чому ми все ще бачимо священні війни між прихильниками функціонального та об'єктно-орієнтованого програмування? Це здається мені величезною втраченою можливістю. Що може означати «програмування з декількома парадигмами»? Що означало б відкинути чистоту та використовувати будь-який набір функцій, що забезпечує найкраще рішення в будь-якому даному контексті? Найважливіше програмне забезпечення є досить значним, щоб воно, безумовно, мало компоненти, де об’єктно-орієнтована парадигма має більший сенс, і компоненти, де функціональна парадигма вища.

Наприклад, подивіться на “функціональну” функцію, як рекурсія. Є, звичайно, алгоритми, які мають набагато більше сенсу рекурсивно (Вежі Ханоя або друк відсортованого двійкового дерева по порядку); є алгоритми, де не має великої різниці, використовуєте ви цикли чи рекурсію (коли оптимізація рекурсії хвоста спрацює); і є, звичайно, випадки, коли рекурсія буде повільною та потребуватиме пам’яті. Скільки програмістів знає, яке рішення найкраще в будь-якій ситуації?

Ось такі запитання нам потрібно розпочати. Шаблони проектування з самого початку були пов’язані з об’єктно-орієнтованим програмуванням. Які види дизайнерських зразків мають сенс у світі з багатьма парадигмами? Пам’ятайте, що шаблони дизайну не «винайдені»; за ними спостерігають, вони вирішують проблеми, які з’являються знову і знову, і це має стати частиною вашого репертуару.

Прикро, що функціональні програмісти, як правило, не говорять про шаблони дизайну; коли ви усвідомлюєте, що закономірності є спостережуваними рішеннями, твердження типу „шаблони не потрібні у функціональних мовах” перестають мати сенс. Функціональні програмісти, безсумнівно, вирішують проблеми, і, звичайно, бачать, що одні й ті ж рішення з’являються неодноразово. Не слід очікувати, що ці проблеми та рішення будуть такими самими проблемами та рішеннями, які спостерігають програмісти ОО. Які схеми дають найкращі результати з обох парадигм? Які закономірності можуть допомогти визначити, який підхід є найбільш підходящим у тій чи іншій ситуації?

Мови програмування представляють способи мислення над проблемами. З роками парадигми множились разом із проблемами, які ми зацікавлені у вирішенні. Зараз ми говоримо про програмування на основі подій, і багато програмних систем керуються подіями, принаймні на передній панелі. Метапрограмування було популяризовано JUnit, першим широко використовуваним інструментом, який спирався на цю функцію, яка частіше асоціюється з функціональними мовами; відтоді кілька кардинально різних версій метапрограмування зробили можливим нові речі в Java, Ruby та інших мовах.

Ми ніколи не розглядали проблему того, як змусити ці парадигми добре грати разом; до цього часу мови, які підтримують кілька парадигм, залишали програмістам зрозуміти, як ними користуватися. Але просто змішати парадигми спеціально, мабуть, не є ідеальним способом побудови великих систем - і ми зараз розробляємо програмне забезпечення на таких масштабах і швидкостях, які важко було уявити лише кілька років тому. Наші інструменти вдосконалились; тепер нам потрібно навчитися добре ними користуватися. І це неминуче призведе до змішування парадигм, які ми давно розглядаємо як окремі або навіть конфліктні.

Введення до мови програмування Ocaml

Як будь-який інша мова функціонального програмування, Objective CAML це мова виразів, що складається в основному з створення функцій і їх застосування. Результатом обчислення одного з таких виразів є значення даної мови (value in the language) і виконання програми полягає в обчисленні всіх виразів з яких вона складається.
У Objective CAML визначені наступні типи: цілі числа, числа з плаваючою комою, символьний, рядковий і логічний.

Розрізняють цілі **`int`** і числа з плаваючою комою **`float`**. Якщо результат цілочисельної операції виходить за інтервал значень типу **`int`**, то це не призведе до помилки. Підсумок буде перебувати в інтервалі цілих чисел, тобто всі дії над цілими обмежені операцією **`modulo`** з межами інтервалу. Значення різних типів, таких як **`float`** і **`int`**, не можуть порівнюватися між собою безпосередньо. Для цього існує функції перекладу одного типу в інший (**`float_of_int`** і **`int_of_float`**).

```
# 1 ;;
-: int = 1
# 1 + 2 ;;
-: int = 3
# 9 / 2 ;;
-: int = 4
# 11 mod 3 ;;
-: int = 2
(* Ліміт подання *)
(* Цілих чисел *)
# 2147483650 ;;
-: int = 2
```

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
-: float = 3.3
# 9.1 /. 2.2 ;;
-: float =
4.13636363636
# 1. /. 0. ;;
-: float = Inf
(* Ліміт подання *)
(* З плав. комою *)
# 222222222222.11111 ;;
-: float = 222222222222
```

Символи, тип **`char,`** відповідають цілим числам в інтервалі від 0 до 255, перші 128 значень відповідають кодам ASCII. Функція **`char_of_int i int_of_char`** перетворять один тип в інший. Рядки, тип **`string`** це послідовність символів певної довжини. Оператором об'єднання рядків (конкатенації) є шапка "**`^`**". Наступні функції необхідні для перетворення типів **`int_of_string, string_of_int, string_of_float i float_of_string`**.

Якщо рядок складається з цифр, то ми не зможемо використовувати її в кількісних операціях, не виконавши явного перетворення.

Значення типу **`boolean`** належить множині що складається з двох елементів: **`true`** і **`false`**.

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

Оператори порівняння і рівності - це поліморфні оператори, тобто вони можуть бути застосовані, як для порівняння двох цілих, так і двох рядків. Єдине обмеження це те, що операнди повинні бути одного типу.

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0))
;;
- : bool = true
# "un" < "deux" ;;
- : bool = false
```

Структурна рівність, при перевірці двох змінних, порівнює значення полів структури, тоді як фізична рівність перевіряє займають ці змінні одне і те ж місце в пам'яті. Обидва оператора повертають однаковий результат для простих типів: **`boolean, char, int`** і константні конструктори.

Тип **`unit`** визначає множина з усього одного елемента, значення якого: **`( )`**

```
# ( );;
: Unit = ( )
```

Значення різних типів можуть бути згруповані в кортежі. Значення з яких складається кортеж розділяються комою. Для конструкції кортежу, використовується символ `*`.
`int*string` є кортеж, в якому перший елемент ціле число і другий рядок.

```
# (12, "octobre");;
: Int * string = 12, "octobre"
```

Існує різниця між парою і кодонів: тип `int*int*int` відмінний від `(int*int)*int` і `int*(int*int)`. Методи доступу до елементів триплета (і інших кортежів) не визначені в стандартній бібліотеці. У разі необхідності будемо використовувати зіставлення зі зразком.

Значення одного і того ж типу можуть бути об'єднані в списки. Список може бути або порожнім, або містити однотипні елементи.

```
#  [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
```

Для того щоб додати елемент в початок списку існує наступна функція у вигляді інфіксного оператора `::` - аналог **`cons`** в Caml.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Для об'єднання (конкатенації) списків існує інфіксной оператор: `@` (вухо, собака).

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

Решта функцій маніпуляції списками визначені в бібліотеці **`List`**. Функції **`hd`** і **`tl`** дають доступ до першого і останнього елемента списку.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.tl [ 1 ; 2 ; 3 ] ;;
- : int = 3
```

Поки що ми розглядали структури даних, такі як списки і кортежі, зумовлені в самій мові. Але OCaml дозволяє також визначати нові типи даних. Нижче наводиться іграшковий приклад типу даних, що представляє точку в двомірному просторі:

```
# type point2d = { x : float; y : float };;
type point2d = { x : float; y : float; }
```

`point2d` - це тип запису, який можна розглядати, як кортеж з іменованими елементами. Записи конструюються дуже просто:

```
# let p = { x = 3.; y = -4. };;
val p : point2d = {x = 3.; y = -4.}
```

Видаляти з даних цього типу можна за допомогою зіставлення зі зразком:

```
# let magnitude { x = x_pos; y = y_pos } =
sqrt (x_pos ** 2. +. y_pos ** 2.);;
val magnitude : point2d -> float = <fun>
```

Зіставлення зі зразком тут пов'язує змінну **`x_pos`** зі значенням в полі **`x`**, а змінну **`y_pos`** - багатозначно в поле **`y`**.

Цей код можна скоротити, скориставшись прийомом ущільнення полів (**field punning**). Зокрема, якщо ім'я поля і ім'я пов'язують з ним змінної збігаються, ім'я поля можна опустити. Тобто нашу функцію **magnitude** можна переписати так:

```
# let magnitude {x; y} = sqrt (x**2. +. y**2.);;
val magnitude : point2d -> float = <fun>
```

Для звернення до полів записів можна також використовувати точкову нотацію:

```
# let distance v1 v2 =
magnitude { x = v1.x -.v2.x; y = v1.y -.v2.y };;
val distance : point2d -> point2d -> float =
<fun>
```

І, звичайно ж, нові типи можна використовувати в визначеннях більших типів. Нижче наводяться деякі типи, що моделюють різні геометричні об'єкти, що містять значення типу point2d:

```
# type circle_desc = { center: point2d; radius:
float }
type rect_desc
 = { lower_left: point2d; width: float; height:
float }
type segment_desc = { endpoint1: point2d;
endpoint2: point2d } ;;
type circle_desc ={center:point2d;radius:float;}
type rect_desc = { lower_left : point2d; width :
float; height : float; }
```

```
type segment_desc = { endpoint1 : point2d;
endpoint2 : point2d; }
```

Тепер уявіть, що вам захотілося об'єднати кілька об'єктів цих типів разом, щоб описати сцену з декількома об'єктами. Для цього необхідний деякий універсальний спосіб представлення цих різнотипних об'єктів у вигляді єдиного типу. Один з таких способів полягає в використанні вариантного типу (variant type):

```
# type scene_element =
| Circle of circle_desc
| Rect of rect_desc
| Segment of segment_desc
;;
type scene_element =
Circle of circle_desc
| Rect of rect_desc
| Segment of segment_desc
```

Символ | розділяє різні випадки варіантного значення (перший символ | можна опустити), а кожен випадок починається з імені, в якому перша буква - заголовна, наприклад **`Circle`**, **`Rect`** або **`Segment`**, щоб їх можна було відрізняти один від одного.

Нижче показано, як можна реалізувати функцію, що визначає, чи знаходиться задана точка в межах деякого елемента списку **`scene_elements`**:

```
# let is_inside_scene_element point
scene_element =
match scene_element with
| Circle { center; radius } ->
distance center point < radius
| Rect { lower_left; width; height } ->
point.x > lower_left.x && point.x < lower_left.x
+. width
&& point.y > lower_left.y && point.y <
lower_left.y +. height
| Segment { endpoint1; endpoint2 } -> false
;;
val is_inside_scene_element : point2d ->
scene_element -> bool = <fun>
```

```
# let is_inside_scene point scene =
List.exists scene
~f:(fun el -> is_inside_scene_element point el)
;;
val is_inside_scene : point2d -> scene_element
list -> bool = <fun>
# is_inside_scene {x=3.;y=7.}
[ Circle {center = {x=4.;y= 4.}; radius =
0.5 } ];;
- : bool = false
# is_inside_scene {x=3.;y=7.}
[ Circle {center = {x=4.;y= 4.}; radius =
5.0 } ];;
- : bool = true
```

Імперативне програмування в Ocaml

Код, який ми писали досі, є майже виключно функціональним. Це, грубо кажучи, означає, що код не змінює змінних або значень в процесі виконання. Насправді майже всі структури даних, що зустрічалися нам, відносяться до розряду незмінних (`immutable`), в тому сенсі, що в мові немає інструментів, які дозволяли б змінювати їх. такий стиль
в корені відрізняється від імперативного стилю програмування, коли обчислення будуються як послідовність інструкцій, що змінюють стан програми.

Функціональний стиль є основним при програмуванні на мові OCaml. При використанні цього стилю змінні і більшість структур даних залишаються незмінними. Але OCaml підтримує і імперативний стиль програмування, дозволяючи змінювати структури даних, такі як масиви і хеш-таблиці, і контролювати потік виконання за допомогою керуючих конструкцій, таких як цикли `for` і `while`.

Масив - важлива змінна структура даних, але не єдина. Записи, незмінні за замовчуванням, можуть містити поля, явно оголошені змінними. Нижче наводиться маленький приклад структури даних для накопичення статистичної інформації про колекції чисел.

```
# type running_sum =
{ mutable sum: float;
mutable sum_sq: float; (* сумма квадратов *)
mutable samples: int;
}
;;
type running_sum = {
mutable sum : float;
```

```
mutable sum_sq : float;
mutable samples : int;
}
```

Поля в **running_sum** оголошені так, що їх легко можна змінювати, накопичуючи в них інформацію. На їх основі можна обчислити середнє і стандартне відхилення, як показано в наступному прикладі. Зверніть увагу, що там використовуються дві **let**-прив'язки без двох точок з комою між ними. Це обумовлено тим, що дві крапки з комою потрібні, тільки щоб повідомити інтерактивної оболонці **utop**, що вона може приступити до обробки введення, але не для відділення двох оголошень:

```
# let mean rsum = rsum.sum /. float rsum.samples
let stdev rsum =
sqrt (rsum.sum_sq /. float rsum.samples
-. (rsum.sum /. float rsum.samples) ** 2.) ;;
val mean : running_sum -> float = <fun>
val stdev : running_sum -> float = <fun>
```

Тут ми використовували функцію **`float`**, яка є зручним еквівалентом функції **`Float.of_int`** з бібліотеки **`Pervasives.`**

Нам також потрібні функції для створення і зміни **`running_sums:`**

```
# let create () = { sum = 0.; sum_sq = 0.;
samples = 0 }
let update rsum x =
rsum.samples <- rsum.samples + 1;
rsum.sum <- rsum.sum +. x;
rsum.sum_sq <- rsum.sum_sq +. x *. x
;;
val create : unit -> running_sum = <fun>
val update : running_sum -> float -> unit =
<fun>
```

Функція **`create`** повертає запис **`running_sum,`** відповідну порожньому безлічі, а **`update rsum x`** змінює **`rsum`**, відображаючи додавання значення **`x`** в множину шляхом зміни числа елементів у множині, суми значень і суми квадратів значень.

Зверніть увагу, що операції в послідовності відокремлюються одна від одної одним символом крапки з комою. Коли ми використовували функціональний стиль, в цьому не було необхідності, але вимоги змінилися при переході до імперативного стилю.

Нижче наводиться приклад використання функцій **`create`** і **`update`**. Відзначте, що в цьому прикладі застосовується функція **`List.iter`**, яка викликає функцію **`~ f`** для кожного елемента зазначеного списку:

```
# let rsum = create ();;
val rsum : running_sum = {sum = 0.; sum_sq = 0.;
samples = 0}
# List.iter [1.;3.;2.;-7.;4.;5.] ~f:(fun x ->
update rsum x);;
- : unit = ()
# mean rsum;;
- : float = 1.33333333333
```

```
# stdev rsum;;
- : float = 3.94405318873
```

Одиночне змінюване значення можна створити за допомогою оголошення **ref**. Тип **ref** визначається в стандартній бібліотеці, але в ньому немає нічого особливого. Це простий тип запису з єдиним змінним полем **contents**:

```
# let x = { contents = 0 };;
val x : int ref = {contents = 0}
# x.contents <- x.contents + 1;;
- : unit = ()
# x;;
- : int ref = {contents = 1}
```

Існує кілька зручних функцій і операторів для роботи з посиланнями:

```
# let x = ref 0 (* создает ссылку, то есть
{ contents = 0 } *) ;;
val x : int ref = {contents = 0}
# !x (* возвращает содержимое ссылки, то есть
x.contents *) ;;
- : int = 0
# x := !x + 1 (* присваивание, то есть
x.contents <- ... *) ;;
- : unit = ()
# !x ;;
- : int = 1
```

У цих операторах немає нічого таємничого. Ви можете реалізувати ту ж саму функціональність всього в декількох рядках коду:

```
# type 'a ref = { mutable contents : 'a }
let ref x = { contents = x }
let (!) r = r.contents
let (:=) r x = r.contents <- x
;;
type 'a ref = { mutable contents : 'a; }
val ref : 'a -> 'a ref = <fun>
val ( ! ) : 'a ref -> 'a = <fun>
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

Комбінація символів `'a` перед `ref` вказує, що тип `ref` є поліморфним за аналогією зі списками, тобто має здатність зберігати значення будь-якого типу.

Круглі дужки навколо `!` і `: =` необхідні, так як це - оператори, а не звичайні функції.

Навіть при тому, що `ref` є всього лише типом записи, він відіграє важливу роль, тому що реалізує стандартний спосіб імітації поведінки традиційних змінюваних змінних, підтримуваних більшістю інших мов.

Наприклад, можна імперативно підсумувати елементи списку, передавши функції `List.iter` іншу, просту функцію, яка буде викликана для кожного елемента списку і збереже суму на за посиланням:

```
# let sum list =
let sum = ref 0 in
List.iter list ~f:(fun x -> sum := !sum + x);
!sum
;;
val sum : int list -> int = <fun>
```

Цикли for і while

Мова **OCaml** підтримує також традиційні імперативні керуючі конструкції, такі як цикли **for** і **while**. Нижче наводиться приклад реалізації випадкової перестановки елементів масиву з використанням циклу **for**. Як джерело випадковості ми використовували модуль **Random**. Модуль **Random** завжди починає обчислення з одного і того ж початкового значення за замовчуванням, але його можна змінити викликом **Random.self_init** для установки іншого начального значення:

```
# let permute array =
let length = Array.length array in
for i = 0 to length - 2 do
(* вибрати j для перестановки *)
let j = i + Random.int (length - i) in
(* Поміняти місцями i и j *)
let tmp = array.(i) in
array.(i) <- array.(j);
array.(j) <- tmp
done
;;
val permute : 'a array -> unit = <fun>
```

З точки зору синтаксису, зверніть увагу на ключові слова, що відрізняють цикл **for**: **for**, **to**, **do** і **done**.

Нижче наведемо приклад використання цього коду:

```
# let ar = Array.init 20 ~f:(fun i -> i);;
val ar : int array =
[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13;
14; 15; 16; 17; 18; 19|]
# permute ar;;
- : unit = ()
# ar;;
- : int array =
[|1; 2; 4; 6; 11; 7; 14; 9; 10; 0; 13; 16; 19;
12; 17; 5; 3; 18; 8; 15|]
```

Крім того, OCaml підтримує цикли **while,** як показано у визначенні наступної функції пошуку першого негативного елементу в масиві. Відзначимо, що **while** (як і **for**) є ключовим словом:

```
# let find_first_negative_entry array =
let pos = ref 0 in
while !pos < Array.length array && array.(!pos)
>= 0 do
pos := !pos + 1
done;
if !pos = Array.length array then None else Some
!pos
;;
```

```
val find_first_negative_entry : int array -> int
option = <fun>
# find_first_negative_entry [|1;2;0;3|];;
- : int option = None
# find_first_negative_entry [|1;-2;0;3|];;
- : int option = Some 1
```

Зауважимо також, що попередній код використовує той факт, що оператор **`&&`** в OCaml (виконує логічну операцію «І») обчислюється по короткій схемі.
Зокрема, в вираженні виду **`expr1 && expr2`** підвираз **`expr2`** обчислюється, тільки якщо підвираз **`expr1`** поверне **`true.`**

Якби це було не так, попередня функція генерувала б помилку виходу за межі масиву. Насправді ми могли б отримати цю помилку, переписавши функцію так, щоб виключити обчислення по короткій схемі:

```
# let find_first_negative_entry array =
let pos = ref 0 in
while
let pos_is_good = !pos < Array.length array in
let element_is_non_negative = array.(!pos) >= 0
in
pos_is_good && element_is_non_negative
do
pos := !pos + 1
```

```
done;
if !pos = Array.length array then None else Some
!pos
;;
val find_first_negative_entry : int array -> int
option = <fun>
# find_first_negative_entry [|1;2;0;3|];;
Exception: (Invalid_argument "index out of
bounds").
```

До сих пір ми знайомилися з самими основними особливостями мови, використовуючи для цього інтерактивну оболонку `utop`. Тепер ми покажемо вам, як створити просту самостійну програму. В даному розділі ми створимо програму, що підсумовує числа зі списку, прочитаного зі стандартного вводу.

Збережемо наступний код у файлі з ім'ям `sum.ml`. Звернемо увагу, що не потрібно завершувати вираження послідовністю `;;`, тому що цього не потрібно за межами інтерактивної оболонки:

```
open Core.Std
let rec read_and_accumulate accum =
let line = In_channel.input_line
In_channel.stdin in
match line with
| None -> accum
| Some x -> read_and_accumulate (accum +.
Float.of_string x)
let () =
printf "Total: %F\n" (read_and_accumulate 0.)
```

Це наш перший досвід використання підпрограм введення / виведення в мові OCaml. Функція **`read_and_accumulate`** є рекурсивної і використовує **`In_channel`**.
`input_line` для читання рядків, одну за однією, зі стандартного вводу, викликаючи саму себе в кожній ітерації і передаючи накопичену суму. Відзначте, що **`input_line`** повертає необов'язкове значення, де **`None`** є ознакою кінця вхідного потоку.
Після того як **`read_and_accumulate`** поверне накопичену суму, її необхідно вивести. Це робиться за допомогою команди **`printf`**, яка забезпечує підтримку рядків формату, подібних до тих, що можна зустріти в багатьох мовах.

Рядок формату аналізується компілятором і використовується для визначення кількості і типів інших аргументів. В даному випадку є всього одна директива форматування, `% F`, тому **`printf`** очікує отримати тільки один додатковий аргумент типу **`float`**.

Скомпілювати програму можна за допомогою **`corebuild`**, невеликий обгортки навколо **`ocamlbuild`**, інструменту збірки, розповсюджуваного разом з компілятором OCaml. Сценарій **`corebuild`** встановлюється разом з бібліотекою **`Core`** і служить для передачі компілятору прапорів, необхідних для складання програми разом з бібліотекою **`Core`**.

```
$ corebuild sum.native
```

Розширення файлу `.native` вказує, що ми створюємо виконуваний файл з машинним кодом. Після закінчення збірки можна запустити вийшов виконуваний файл, як будь-яку іншу утиліту командного рядка. Ми можемо передати програмі `sum.native` вихідні числа, вводячи їх по одному в рядку і натиснувши **Ctrl-D** після закінчення:

```
$ ./sum.native
1
2
3
94.5
Total: 100.5
```

Щоб створити більш зручну утиліту командного рядка, доведеться докласти додаткових зусиль і реалізувати належний інтерфейс командного рядка, а також передбачити обробку помилок.

На наступній лекції ми більш докладніше розглянемо принципи мультипарадигменного програмування в мові Ocaml.