МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

## Лекція 14

# Принципи мультипарадигменого програмування в мові OCaml.

# Змінні і функції

Змінні і функції є фундаментальними ідеями практично у всіх мовах програмування. Однак в OCaml використовується дещо інший підхід до реалізації цих понять, ніж в більшості інших мов, з якими ви, ймовірно, знайомі. Тому в даній лекції ми ближче розглянемо особливості функцій і змінних в мові OCaml, починаючи з найпростішого - як визначити змінну - і закінчуючи такими складними особливостями функцій, як аргументи з мітками і необов'язкові аргументи.

У найпростішому розумінні *змінна* - це ідентифікатор, пов'язаний з деяким значенням. У мові OCaml такі зв'язки часто створюються за допомогою ключового слова let, в загальному випадку має наступний синтаксис (зверніть увагу, що ім'я змінної повинно починатися з літери нижнього регістру або символу підкреслення):

```
let <variable> = <expr>
```

Кожна прив'язка до змінної має певну область видимості - область програми, де можна послатися на цю прив'язку. В інтерактивній оболонці `utop` область видимості будь `let`-прив'язки верхнього рівня поширюється на весь код, наступний нижче, який знадобиться майже сеансу. Коли прив'язка здійснюється всередині модуля, її дія продовжується до кінця цього модуля.

Наприклад.

```
# let x = 3;;
val x : int = 3
# let y = 4;;
val y : int = 4
# let z = x + y;;
val z : int = 7
```

Ключове слово **let** можна також використовувати для створення пов'язаних змінних, область видимості яких обмежується певним виразом, з використанням наступного синтаксису:

**let <variable> = <expr1> in <expr2>**

Тут спочатку буде обчислено вираз **expr1**, а потім вираз **expr2**, всередині якого буде доступна змінна **variable**, пов'язана зі значенням, виробленим виразом **expr1**. Ось як це виглядає у вигляді рядків програми:

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
let language_list = String.split languages
~on:',' in
String.concat ~sep:"-" language_list
;;
val dashed_languages : string = "Ocaml-Perl-C++-
C"
```

Зверніть увагу, що область видимості зміною **`language_list`** обмежується виразом **`String.concat ~ sep: "-" language_list`**, і вона недоступна на верхньому рівні, у чому можна переконатися, спробувавши отримати її значення:

```
# language_list;;
Characters -1-13:
Error: Unbound value language_list
```

(Помилка: Несв’язне значення language_list)

**let**-прив'язки у вкладеній області видимості можуть затінювати, або приховувати, визначення у зовнішній області видимості. Так, ми могли б записати приклад **dashed_languages** по іншому:

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
let languages = String.split languages ~on:','
in
String.concat ~sep:"-" languages
;;
val dashed_languages: string="OCaml-Perl-C++-C"
```

На цей раз у внутрішній області видимості ми зв'язали список рядків з ім'ям **`languages`** замість **`language_list`**, приховавши тим самим оригінальне визначення **`languages`**. А

жами визначення **`dashed_languages`** дію внутрішньої області видимості закінчується, і знову починає діяти початкове визначення **`languages`**:

```
# languages;;
- : string = "Ocaml,Perl,C++,C"
```

Для виконання складних обчислень часто прийнято використовувати послідовність вкладених виразів **let / in.**

Наприклад, можна записати такий вираз:

```
# let area_of_ring inner_radius outer_radius =
let pi = acos (-1.) in
let area_of_circle r = pi *. r *. r in
area_of_circle outer_radius -. area_of_circle
inner_radius
;;
val area_of_ring : float -> float -> float =
<fun>
# area_of_ring 1. 3.;;
- : float = 25.1327412287
```

Чому змінні не можуть змінюватися?
Факт незмінності змінних є одним з джерел непорозуміння для тих, хто тільки починає освоювати OCaml. Це здається дивним навіть з точки зору термінології. Хіба суть змінної не в тому, щоб змінюватися?
Відповідь на це питання проста: змінні в OCaml (і взагалі в функціональних мовах) в дійсності більше нагадують змінні в рівняннях, ніж змінні в імперативних мовах.

Уявіть собі математичне тотожність **`x (y + z) = xy + xz.`**
Змінні **`x`**, **`y`** і **`z`** в ньому не можна змінити. Вони змінюються в тому сенсі, що ви можете підставити в це тотожність різні числа, для яких воно буде виконуватися.
Те ж справедливо і для змінних в функціональних мовах. Функціям можна передати різні вхідні значення, внаслідок чого змінні в них будуть набувати різних значень без всякого зміни.

Іншим корисним властивістю **let**-прив'язки є підтримка зразків (**patterns**) з лівого боку. Погляньте на наступний код, який використовує **List.unzip**, функцію для перетворення списку пар в пару списків:

```
# let (ints,strings) = List.unzip [(1,"one");
(2,"two"); (3,"three")];;
val ints : int list = [1; 2; 3]
val strings : string list = ["one"; "two";
"three"]
```

Застосування зразка в `let`-прив'язці має більше сенсу для безперечних зразків, тобто коли будь-яке значення розглянутого типу гарантовано буде відповідати зразку. Зразки кортежів і записів є безперечними, а зразки списків - немає. Погляньте на наступний приклад, який реалізує функцію для перекладу в верхній регістр першого елемента списку значень, розділених комами:

```
# let upcase_first_entry line =
let (first :: rest) = String.split ~on:',' line
in   String.concat   ~sep:","   (String.uppercase
first :: rest)
;;
Characters 40-53:
Warning   8:   this   pattern-matching   is   not
exhaustive.
Here   is   an   example   of   a   value   that   is   not
matched:
[]
val   upcase_first_entry   :   string   ->   string   =
<fun>
```

На практиці дана ситуація неможлива, тому що **`String.split`** завжди повертає список як мінімум з одним елементом. Але компілятор не знає про це і тому вивів попередження:

```
(Попередження 8: це порівняння не є вичерпним.
Нижче слідує приклад значення, яке не відповідає
співставлення:
[])
```

## Функції

З огляду на, що OCaml є функціональним мовою, не дивно, що функції в ньому грають важливу роль. Фактично функції присутні практично у всіх прикладах, які ми бачили. Зараз дамо більш докладну інформацію про функції та розповімо про те, як діють функції в мові OCaml. Як буде показано далі, функції в OCaml мають безліч відмінностей від функцій в основних мовах програмування.

## Анонімні функції

Почнемо зі знайомства з найбільш типовим способом оголошення функцій в мові OCaml: *анонімними функціями*. Анонімна функція - це функція, оголошена без імені. Такі функції можуть оголошуватися за допомогою ключового слова **fun**, як показано нижче:

```
# (fun x -> x + 1);;
- : int -> int = <fun>
```

Анонімні функції діють практично так само, як іменовані. Наприклад, анонімну функцію можна застосувати до аргументу:

```
# (fun x -> x + 1) 7;;
- : int = 8
```

Або передати іншій функції. Передача функцій ітеративним функцій, таких як **`List.map`**, є, мабуть, найбільш типовим випадком використання анонімних функцій:

```
# List.map ~f:(fun x -> x + 1) [1;2;3];;
- : int list = [2; 3; 4]
```

Ними можна навіть наповнювати структури даних:

```
# let increments = [ (fun x -> x + 1); (fun x ->
x + 2) ] ;;
val increments : (int -> int) list = [<fun>;
<fun>]
# List.map ~f:(fun g -> g 5) increments;;
- : int list = [6; 7]
```

Важливо зрозуміти, що функції в мові OCaml є самими звичайними значеннями і з ними можна виконувати будь-які операції, допустимі для звичайних значень, включаючи передачу іншим функціям і повернення з функцій, а також збереження в структурах даних. Навіть імена присвоюються функцій точно так же, як будь-яким іншим значенням, - за допомогою **let**-прив'язки:

```
# let plusone = (fun x -> x + 1);;
val plusone : int -> int = <fun>
# plusone 3;;
- : int = 4
```

Визначення іменованих функцій настільки типово, що для цього навіть є *синтаксичний цукор* (доповнення синтаксису мови програмування, які не додають нових можливостей, а роблять використання мови програмування зручнішим для людини). Наприклад, таке визначення функції **`plusone`** еквівалентно попередньому:

```
# let plusone x = x + 1;;
val plusone : int -> int = <fun>
```

Це - найбільш часто використовуваний і зручний спосіб оголошення функцій, але, крім синтаксичних тонкощів, ці два стилі визначення функцій абсолютно еквівалентні.

## let і fun

Функції та **`let`**-прив'язки мають безпосереднє відношення один до одного. Параметри функції можна розглядати як змінні, які будуть пов'язані зі значеннями, переданими викликає програмою. Наприклад, наступні два вирази майже еквівалентні:

```
# (fun x -> x + 1) 7;;
- : int = 8
# let x = 7 in x + 1;;
- : int = 8
```

## Функції декількох аргументів

Звичайно ж, OCaml підтримує функції декількох аргументів, такі як:

```
# let abs_diff x y = abs (x - y);;
val abs_diff : int -> int -> int = <fun>
# abs_diff 3 4;;
- : int = 1
```

Сигнатура функції **abs_diff** з усіма її стрілками може здатися трохи складною. Щоб зрозуміти, чому вона вийшла такою, давайте перепишемо **abs_diff**, представивши її в еквівалентній формі за допомогою ключового слова **fun**:

```
# let abs_diff =
(fun x -> (fun y -> abs (x - y)));;
val abs_diff : int -> int -> int = <fun>
```

Така форма запису більш явно показує, що **`abs_diff`** фактично є функцією одного аргументу, що повертає іншу функцію одного аргументу, яка, в свою чергу, повертає кінцевий результат. Так як функції вкладені одна в одну, внутрішнє вираження **`abs (x - y)`** має доступ до верхньому значенню - до значення **`x`**, пов'язаного застосуванням зовнішньої функції, і до значення **`y`**, пов'язаному застосуванням внутрішньої функції.

Функції такого виду називають *каріованими функціями (curried function)*. (Назва *«каррінг»* було прийнято на честь математика і логіка *Хаскела Каррі (Haskell Curry)*, який справив великий вплив на теорію і архітектуру мов програмування.)

Для правильної інтерпретації сигнатури типу карірованої функції важливо пам'ятати, що стрілка **->** є право асоціативною. Відповідно, в сигнатуру **abs_diff** можна додати круглі дужки, як показано нижче:

```
val abs_diff : int -> (int -> int)
```

Дужки не змінюють суті сигнатури, але з їх допомогою легше побачити карінг.

Карінг - це набагато більше, ніж теоретичний курйоз. Карінг можна використовувати для спеціалізації функції передачею їй деяких з аргументів. Наприклад, в наступному фрагменті створюється спеціалізована версія функції **abs_diff**, що обчислює відстань до заданого числа від числа 3:

```
# let dist_from_3 = abs_diff 3;;
val dist_from_3 : int -> int = <fun>
# dist_from_3 8;;
- : int = 5
# dist_from_3 (-1);;
- : int = 4
```

Прийом застосування деяких аргументів карірованої функції, щоб отримати нову функцію, називається *частковим застосуванням (partial application)*. Зверніть увагу, що ключове слово fun підтримує карінг, надаючи власний синтаксис, тому наступне визначення функції **abs_diff** еквівалентно попереднього.

```
# let abs_diff = (fun x y -> abs (x - y));;
val abs_diff : int -> int -> int = <fun>
```

Карінг - не єдиний спосіб визначення функцій декількох аргументів на OCaml. У цій мові допускається також використовувати в ролі аргументів різні елементи кортежу. Тобто ми цілком могли б написати таке визначення функції:

```
# let abs_diff (x,y) = abs (x - y);;
val abs_diff : int * int -> int = <fun>
# abs_diff (3,4);;
- : int = 1
```

При використанні цієї форми визначення функцій компілятор OCaml виконує деякі оптимізації. Зокрема, він не створює в пам'яті новий кортеж з єдиною метою передати аргументи функції. Але ви не зможете використовувати прийом часткового застосування з функціями, оголошеними таким чином.

Кожен з цих стилів має свої переваги, але в загальному випадку слід дотримуватися каріровання, тому що цей стиль є стилем за замовчуванням в світі Ocaml.

## Рекурсивні функції

Функція вважається рекурсивної, якщо вона посилається на саму себе. Рекурсія - важливий прийом в будь-якій мові програмування, але в функціональних мовах він має особливу значущість, тому що дає можливість створювати циклічні конструкції.

Щоб визначити рекурсивну функцію, необхідно позначити **`let`**-прив'язку як рекурсивну за допомогою ключового слова **`rec`**, як показано в наступному прикладі визначення функції, що виконує пошук першого повторюваного елемента в списку:

```
# let rec find_first_stutter list =
match list with
| [] | [_] ->
(* нуль або один елемент – нема повторюваних
елементів *)
None
| x :: y :: tl ->
if x = y then Some x else find_first_stutter
(y::tl)
;;
val find_first_stutter : 'a list -> 'a option =
<fun>
```

Можна також визначити безліч взаємно рекурсивних значень, використовуючи **let rec** в поєднанні з ключовим словом **and**. Приклад наводиться нижче:

```
# let rec is_even x =
if x = 0 then true else is_odd (x - 1)
and is_odd x =
if x = 0 then false else is_even (x - 1)
;;
val is_even : int -> bool = <fun>
val is_odd : int -> bool = <fun>
# List.map ~f:is_even [0;1;2;3;4;5];;
-: bool list=[true;false;true;false;true; false]
# List.map ~f:is_odd [0;1;2;3;4;5];;
-: bool list=[false;true;false;true;false; true]
```

## Оголошення функцій за допомогою ключового слова function

Інший спосіб визначити функцію - скористатися ключовим словом **`function`**.

Замість синтаксичного підтримки оголошення функцій декількох аргументів (карінг) ключове слово **`function`** пропонує вбудовану підтримку зіставлення зі зразком.

Наприклад:

```
# let some_or_zero = function
| Some x -> x
| None -> 0
;;
val some_or_zero : int option -> int = <fun>
# List.map ~f:some_or_zero [Some 3; None; Some
4];;
- : int list = [3; 0; 4]
```

Різні стилі оголошення функцій можуть комбінуватися без будь-яких обмежень, як показано в наступному прикладі, де визначається функція двох аргументів (карірована), другий аргумент якої аналізується за допомогою виразу зіставлення зі зразком:

```
# let some_or_default default = function
| Some x -> x
| None -> default
;;
val some_or_default : 'a -> 'a option -> 'a =
<fun>
# some_or_default 3 (Some 5);;
- : int = 5
```

```
# List.map ~f:(some_or_default 100) [Some 3; None; Some 4];;
- : int list = [3; 100; 4]
```

Зверніть також увагу на використання прийому часткового застосування для створення функції, яка потім передається в виклик **`List.map`**. Іншими словами, **`some_or_default 100`** - це функція, створена передачею єдиного, перший аргумент функції **`some_or_default`**.

## Аргументи з мітками

До сих пір всі наші функції розрізняли свої аргументи по їх позиціях, тобто порядок передачі аргументів функцій відігравав важливу роль. Однак мова OCaml підтримує також аргументи з мітками (labeled arguments), дозволяючи функцій ідентифікувати аргументи по іменах. Насправді ми вже зустрічали функції з бібліотеки `Core`, такі як `List.map`, що використовують аргументи з мітками. Визначення таких аргументів починаються зі знака тильди (`~`), за яким слід мітка (завершується двокрапкою), яка в тілі функції буде служити ім'ям аргументу.

Наприклад:

```
# let ratio ~num ~denom = float num /. float denom;;
val ratio : num:int -> denom:int -> float = <fun>
```

Передавати аргументи таких функцій можна, використовуючи схожу угоду. Як показано нижче, аргументи з мітками допускається передавати в довільному порядку:

```
#  ratio ~num:3 ~denom:10;;
- : float = 0.3
# ratio ~denom:10 ~num:3;;
- : float = 0.3
```

Мова OCaml підтримує також *ущільнення міток (label punning)*, тобто, якщо ім'я мітки і ім'я змінної збігаються, текст після двокрапки (:) можна відкинути. Фактично ми вже використали цей прийом в оголошенні функції **`ratio`**. Наступний приклад демонструє, як прийом ущільнення міток можна використовувати у виклику функції:

main.topscript

```
# let num = 3 in
let denom = 4 in
ratio ~num ~denom;;
- : float = 0.75
```

Аргументи з мітками з успіхом можна використовувати в різних ситуаціях.

- Коли визначається функція з великим числом аргументів. Вище певного порогу аргументи простіше запам'ятовувати по іменах, ніж по позиціях.
- Коли призначення того чи іншого аргументу неочевидно з сигнатури. Уявіть функцію, яка створює хеш-таблицю, в першому аргументі якої передається початковий розмір масиву, що лежить в основі хеш-таблиці, а в другому - логічний прапор, який вказує, чи повинен скорочуватися обсяг масиву при видаленні елементів:

```
val create_hashtable : int -> bool -> ('a,'b) Hashtable.t
```

За однією тільки сигнатурою складно зрозуміти призначення аргументів, але, додавши мітки, можна зробити наші наміри більш очевидними:

```
val create_hashtable :
init_size:int -> allow_shrinking:bool -> ('a,'b)
Hashtable.t
```

Особливо важливу роль відіграє вибір імен міток щодо логічних значень, оскільки часом важко зрозуміти сенс значення **`true`** - «дозволити» або «заборонити» ту чи іншу особливість.

Коли визначається функція з декількома аргументами, які легко сплутати один з одним. Ця проблема особливо характерна для аргументів одного типу. Наприклад, погляньте на сигнатуру функції, що витягає подстроку:

```
val substring: string -> int -> int -> string
```

Тут є два цілочисельних аргументи, що визначають початкову позицію підрядка в рядку і довжину підрядка, відповідно. Ми можемо позначити їх більш очевидним чином, додавши мітки в сигнатуру:

```
val substring: string -> pos:int -> len:int -> string
```

Дотримання цього правила покращує читаність сигнатур і клієнтського коду і знижує ймовірність плутанини позиції і довжини підрядка.

Там, де необхідно більш висока гнучкість в порядку проходження аргументів при виклику функції. Уявіть собі функцію, таку як `List.iter`, яка приймає два аргументи: функцію і список елементів, до якої застосовуватиметься ця функція.

На практиці часто використовується прийом часткового застосування **List.iter** за рахунок передачі їй одній тільки функції, як показано в наступному прикладі:

```
# String.split ~on:':' path
|> List.dedup ~compare:String.compare
|> List.iter ~f:print_endline
;;
/bin
/sbin
/usr/bin
/usr/local/bin
- : unit = ()
```

При такому підході аргумент з функцією повинен слідувати в оголошенні першим. Однак іноді зручніше, коли аргумент з функцією передається другим. Одна з типових причин зміни аргументів місцями - читабельність коду. Зокрема, коли функція декількох аргументів передається іншій функції, код простіше читати, якщо ця функція передається останньої.

## Функції вищого порядку і мітки

Аргументи з мітками мають один прикрий недолік: коли викликається функція, що має аргументи з мітками, порядок їх слідування дійсно не має значення, але він має значення в контексті вищого порядку, наприклад коли функція з такими аргументами передається іншій функції. Наприклад:

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Тут визначається функція **apply_to_tuple**, приймаюча в першому аргументі функцію двох аргументів з мітками **first** і **second**, наступними саме в такому порядку. Ми могли б визначити **apply_to_tuple** інакше, змінивши порядок проходження аргументів з мітками:

```
# let divide ~first ~second = first / second;;
val divide : first:int -> second:int -> int = <fun>
```

можна виявити, що її не можна передати функції **apply_to_tuple_2**.

```
# apply_to_tuple_2 divide (3,4);;
Characters 17-23:
Error: This expression has type first:int ->
second:int -> int
but an expression was expected of type second:'a
-> first:'b -> 'c
(Помилка: Цей вираз має тип first:int ->
second:int -> int,
тоді як очікувався вираз типу second:’a ->
first:’b -> ‘c)
```

Але її можна передати первісної версії **apply_to_tuple**:

```
# let apply_to_tuple f (first,second) = f ~first
~second;;
val apply_to_tuple : (first:'a -> second:'b ->
'c) -> 'a * 'b -> 'c = <fun>
# apply_to_tuple divide (3,4);;
- : int = 0
```

## Необов'язкові аргументи

Необов'язковий аргумент чимось нагадує аргумент з міткою. Зухвалий код може за вибором передавати або не передавати такий аргумент в виклик функції. Необов'язкові аргументи передаються із застосуванням того ж синтаксису, що й аргументи з мітками, і, подібно до аргументів з мітками, можуть зазначатися в будь-якому порядку.

Нижче наводиться приклад функції, що виконує конкатенацію рядків з необов'язковим роздільником. Вона використовує оператор **^** для попарного об'єднання рядків:

```
# let concat ?sep x y =
let sep = match sep with None -> "" | Some x ->
x in
x ^ sep ^ y
;;
val concat : ?sep:string -> string -> string ->
string = <fun>
# concat "foo" "bar"
 (* без необов'язкового аргументу *);;
- : string = "foobar"
```

```
# concat ~sep:":" "foo" "bar" (* з необов'язковим аргументом *);;
- : string = "foo:bar"
```

Тут знак питання (**?**) у визначенні функції використовується, щоб позначити аргумент **sep** як необов'язковий. Якщо викликає код передасть в аргументі **sep** значення типу **string**, всередині функції аргумент **sep** буде доступний як значення типу **string option**, в іншому випадку він буде доступний як значення **None**.

У попередньому прикладі було потрібно писати шаблонний код для вибору роздільник за замовчуванням, коли викликає код не передає аргумент **sep**. Це настільки поширений шаблон програмування, що для нього було додано явний синтаксис визначення значення за замовчуванням, що дозволяє записати реалізацію **concat** більш компактно:

```
# let concat ?(sep="") x y = x ^ sep ^ y ;;
val concat : ?sep:string -> string -> string ->
string = <fun>
```

Необов'язкові аргументи дуже зручні, але не слід зловживати ними. Головна перевага необов'язкових аргументів полягає в тому, що вони дозволяють писати функції декількох аргументів, які користувачі можуть просто ігнорувати в більшості ситуацій і згадувати про них, тільки коли потрібно передати значення, відмінні від значень за замовчуванням. Вони також дозволяють доповнювати API новими функціональними можливостями, не змінюючи використовує його коду.

Недолік же полягає в тому, що користувач може не дізнатися про можливості вибору і через це мимоволі (і помилково) використовувати поведінка за умовчанням. Взагалі кажучи, використовувати необов'язкові аргументи має сенс, тільки коли перевага компактності коду переважує недолік відсутності явності.

# СПИСКИ ТА ЗРАЗКИ

## Списки

Списки в мові OCaml є незмінними, кінцевими послідовностями елементів одного типу. Як ви вже знаєте, списки в OCaml можна створювати за допомогою квадратних дужок і крапок з комою:

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

З використанням еквівалентної нотації `::` :

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

Як бачите, оператор `::` є право асоціативним, а це означає, що списки можна конструювати без застосування круглих дужок. Порожній список `[]` використовується тут для позначення кінця списку.

Відзначимо, що порожній список є поліморфним, тобто може використовуватися з елементами будь-якого типу, наприклад:

```
# let empty = [];;
val empty : 'a list = []
# 3 :: empty;;
- : int list = [3]
# "three" :: empty;;
- : string list = ["three"]
```

## Використання зіставлення зі зразком для отримання даних зі списку

Читати дані зі списку можна за допомогою інструкції **match**. Нижче наводиться простий приклад рекурсивної функції, що визначає суму всіх елементів списку:

```
# let rec sum l =
match l with
| [] -> 0
| hd :: tl -> hd + sum tl
;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
```

```
# sum [];;
- : int = 0
```

Цей код слід домовленості про використання імені **`hd`** для подання першого елемента (або голови **`(head)`**) списку і імені **`tl`** для подання решти (або хвоста **`(tail)`**) списку.

Інструкція **`match`** в функції **`sum`** насправді виконує дві операції: по-перше, вона діє як інструмент вибору, виділяючи різні можливі варіанти, і, по-друге, дає можливість привласнити імена окремих елементах вихідної структури даних. В даному випадку змінні **`hd`** і **`tl`** зв'язуються другим зразком в інструкції **`match`**. Змінні, пов'язані таким способом, можна використовувати в вираженні праворуч від стрілки, наступного за поточним зразком.

Той факт, що інструкцію **`match`** можна використовувати для зв'язування нових змінних, може бути джерелом непорозумінь. Давайте подивимося, як.

Уявіть, що нам потрібно написати функцію, відфільтровує зі списку елементи, рівні деякому значенню. У вас напевно з'явиться бажання написати код, як показано нижче, але при спробі скомпілювати його компілятор відразу ж видасть попередження:

```
# let rec drop_value l to_drop =
match l with
| [] -> []
| to_drop :: tl -> drop_value tl to_drop
| hd :: tl -> hd :: drop_value tl to_drop
;;
Characters 114-122:
Warning 11: this match case is unused.
```

```
val drop_value : 'a list -> 'a -> 'a list =
<fun>
(Попередження 11: цей зразок не використовується
val drop_value : 'a list -> 'a -> 'a list =
<fun>)
```

Більш того, функція буде працювати неправильно, фільтруючи все елементи, а не тільки ті, що рівні зазначеному значенню:

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

Тут важливо розуміти, що присутність **`to_drop`** в другому зразку не має на увазі порівняння першого елемента списку зі значенням аргументу **`to_drop`**, переданого функції **`drop_value`**. Замість цього просто створюється нова змінна **`to_drop`**. Вона буде пов'язана з першим елементом списку і приховає колишнє визначення **`to_drop`**. Третій зразок виявиться невживаних, тому що фактично той же самий шаблон використовується у другому зразку.

**На наступній лекції ми розглянемо записи, варіанти, функтори, модулі та елементи ООП в мові OCaml.**