МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

Лекція 15

Принципи мультипарадигменого програмування в мові Ocaml (завершення)

ВАРІАНТИ

Варіантні типи є однією з найбільш корисних можливостей мови OCaml, а також однією з найбільш незвичайних. Вони дозволяють представляти дані, які можуть приймати самі різні форми, де кожна форма явно позначається явним тегом. Як буде показано в цьому розділі, при об'єднанні з зіставленням зі зразком варіанти забезпечують зручний спосіб представлення комплексних даних і організації їх аналізу.

Нижче наводиться базовий синтаксис оголошення варіантного типу:

```
type <вариант> =
    | <Тег> [ of <тип> [* <тип>]... ]
    | <Тег> [ of <тип> [* <тип>]... ]
    | ...
```

Кожен рядок описує одну з форм представлення *варіантного типу*. Для кожної форми визначається тег і додатково може визначатися послідовність полів, де кожне поле має зазначений тип.

Розглянемо конкретний приклад використання варіантів. Практично всі термінали підтримують вісім основних кольорів, і ми можемо організувати їх представлення за допомогою варіанту. Всі кольори оголошуються як прості теги і розділяються символом вертикальної риски (|). Зверніть увагу, що теги варіантів повинні починатися з великої літери:

```
# type basic_color =
    | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;
type basic_color =
    Black
  | Red
  | Green
  | Yellow
  | Blue
  | Magenta
  | Cyan
  | White
# Cyan ;;
- : basic_color = Cyan
# [Blue; Magenta; Red] ;;
- : basic_color list = [Blue; Magenta; Red]
```

Наступна функція використовує зіставлення зі зразком для перетворення **`basic_color`** в відповідне ціле число. Перевірка повноти вираження зіставлення зі зразком, що виконується компілятором, гарантує висновок попередження, якщо ми пропустимо якийсь колір:

```
# let basic_color_to_int = function
  | Black -> 0 | Red     -> 1 | Green -> 2 | Yellow -> 3
  | Blue  -> 4 | Magenta -> 5 | Cyan  -> 6 | White  -> 7 ;;
val basic_color_to_int : basic_color -> int = <fun>
# List.map ~f:basic_color_to_int [Blue;Red];;
- : int list = [4; 1]
```

За допомогою цієї функції ми зможемо згенерувати відповідні екрановані послідовності (escape-послідовності) для зміни кольору рядка при відображенні її в терміналі:

```
# let color_by_number number text =
    sprintf "\027[38;5;%dm%s\027[0m" number text;;
val color_by_number : int -> string -> string = <fun>
# let blue = color_by_number (basic_color_to_int Blue) "Blue";;
val blue : string = "\027[38;5;4mBlue\027[0m"
# printf "Hello %s World!\n" blue;;
Hello Blue World!
```

У більшості терміналів слово **«Blue»** буде виведено синім кольором.

У цьому прикладі форми варіанти є простими тегами, з якими не пов'язані ніякі дані. У такому вигляді варіантний тип діє подібно перерахуванням в мові C і Java. Але, як ми побачимо нижче, варіанти здатні на більше, ніж представляти прості перерахування. Взагалі кажучи, простого перерахування недостатньо для ефективного опису повного безлічі квітів, підтримуваного сучасними терміналами. Багато термінали, включаючи поважний xterm, підтримують 256 різних кольорів, розбиваючи їх на наступні групи:

- вісім базових квітів для простого і жирного шрифтів;
- колірної куб RGB розміром $6 \times 6 \times 6$;
- 24-рівнева палітра відтінків сірого кольору.

Ми також реалізуємо подання цього, більш складного колірного простору в вигляді варіантного типу, але на цей раз різні теги будуть забезпечуватися аргументами, що описують дані. Майте на увазі, що варіанти можуть мати безліч аргументів, розділених зірочками (*):

```
# type weight = Regular | Bold
  type color =
  | Basic of basic_color * weight (* базовые цвета, обычный и жирный шрифты *)
  | RGB of int * int * int        (* цветовой куб 6x6x6 *)
  | Gray of int                   (* 24-уровневая палитра оттенков серого *)
;;
type weight = Regular | Bold
type color =
    Basic of basic_color * weight
  | RGB of int * int * int
  | Gray of int
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

Як і раніше, для перетворення кольору в відповідне ціле число буде використовуватися зіставлення зі зразком. Але в даному випадку зіставлення буде робити набагато більше, ніж просто визначати ту чи іншу форму варіантного типу, - з його допомогою ми будемо отримувати дані, асоційовані з тегами:

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
val color_to_int : color -> int = <fun>
```

Тепер ми маємо можливість виводити текст, використовуючи повну множину доступних кольорів:

```
# let color_print color s =
    printf "%s\n" (color_by_number (color_to_int color) s);;
val color_print : color -> string -> unit = <fun>
# color_print (Basic (Red,Bold)) "A bold red!";;
A bold red!
# color_print (Gray 4) "A muted gray...";;
A muted gray...
```

Універсальні зразки і рефакторинг

Система типів в мові OCaml може діяти як інструмент рефакторінга, попереджаючи про необхідність приведення фрагментів коду у відповідність зі зміненим інтерфейсом. Це особливо цінно щодо варіантних типів.

Погляньте, що трапиться, якщо змінити визначення **`color`**, як показано нижче:

```
# type color =
  | Basic of basic_color    (* базовые цвета *)
  | Bold of basic_color     (* базовые цвета, жирный шрифт *)
```

```
  | RGB of int * int * int (* цветовой куб 6x6x6 *)
  | Gray of int            (* 24-уровневая палитра оттенков серого *)
;;
type color =
    Basic of basic_color
  | Bold of basic_color
  | RGB of int * int * int
  | Gray of int
```

Ми, по суті, розбили базовий випадок **`Basic`** на два, **`Basic`** і **`Bold`**, і тепер варіант **`Basic`** має єдиний аргумент замість двох. Однак функція **`color_to_int`** все ще очікує отримати варіантний тип зі старою структурою, тому якщо спробувати скомпілювати її, компілятор помітить невідповідність:

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
Characters 34-60:
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type basic_color
```

(Помилка: Цьому зразком відповідають значення типу 'a *' b, тоді як очікується зразок, відповідний значенням типу basic_color)

Тут компілятор зауважив, що тег **`Basic`** використовується з невірним числом аргументів. Але варто виправити цю проблему, як компілятор відразу ж помітить іншу - відсутність зразка для нового тега **`Bold`**:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
Characters 19-154:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Bold _
(Предупреждение 8: это сопоставление не является исчерпывающим.
Ниже следует пример значения, не соответствующего сопоставлению:
Bold _ )
val color_to_int : color -> int = <fun>
```

Виправивши цю проблему, ми отримаємо правильну реалізацію:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color

    | Bold basic_color -> 8 + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
val color_to_int : color -> int = <fun>
```

Для опису колекції типів, що включає варіанти, записи і кортежі, часто використовується термін алгебраїчні типи даних. Алгебраїчні типи даних діють як особливо зручний і потужний мова опису даних. Основу їх зручності складають два різновиди типів: типи творів (product types), такі як кортежі і записи, які здатні об'єднувати різні типи і з математичної точки зору нагадують декартові твори; і сумарні типи (sum types), такі як варіанти, які дозволяють складати комбінації з безлічі різних варіантів в рамках одного типу і з математичної точки зору нагадують незв'язні об'єднання (disjoint unions).

```
# module Log_entry = struct
    type t =
      { session_id: string;
        time: Time.t;
        important: bool;
        message: string;
      }
  end
  ;;
module Log_entry :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      important : bool;
      message : string;
    }
  end
```

Цей тип запису об'єднує безліч різнотипних полів даних в єдине значення. Зокрема, єдине значення типу Log_entry.t зберігає ідентифікатор сеансу session_id, і час time, і прапор important, і повідомлення message. Простіше кажучи, типи записів можна вважати кон'юнкція (твором) типів. Варіанти, навпаки, є диз'юнкція (сумою) типів і дозволяють уявити безліч варіантів (форм) єдиного значення, як показано в наступному прикладі:

```
# type client_message = | Logon of Logon.t
                        | Heartbeat of Heartbeat.t
                        | Log_entry of Log_entry.t
  ;;
type client_message =
    Logon of Logon.t
  | Heartbeat of Heartbeat.t
  | Log_entry of Log_entry.t
```

Значним типу client_message може бути значення типу Logon, або Heartbeat, або Log_entry. Якщо буде потрібно написати узагальнений код обробки будь-яких повідомлень, нам якраз стане в нагоді тип client_message, що дозволяє маніпулювати різними допустимими формами (варіантами) повідомлень. У своєму коді ми зможемо виконати зіставлення для значення типу client_message, щоб визначити конкретний тип поточного оброблюваного повідомлення.

Ви можете підвищити точність своїх типів, використовуючи варіанти для представлення різних форм і записи для уявлення загальної структури. Погляньте на наступну функцію, приймаючу список значень типу client_message і повертає всі повідомлення, згенеровані зазначеним користувачем. Ця функція виконує операцію згортки списку повідомлень, де роль акумулятора грає пара:

- множина ідентифікаторів сеансів для даного користувача;
- множина повідомлень, згенерованих даними користувачем.

Ось конкретна реалізація:

```
# let messages_for_user user messages =
    let (user_messages,_) =
      List.fold messages ~init:([],String.Set.empty)
        ~f:(fun ((messages,user_sessions) as acc) message ->
          match message with
          | Logon m ->
            if m.Logon.user = user then
              (message::messages, Set.add user_sessions m.Logon.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            let session_id = match message with
          | Logon m -> m.Logon.session_id
          | Heartbeat m -> m.Heartbeat.session_id
          | Log_entry m -> m.Log_entry.session_id
        in
        if Set.mem user_sessions session_id then
          (message::messages,user_sessions)
        else acc
      )
    in
    List.rev user_messages
  ;;
val messages_for_user : string -> client_message list -> client_message list =
<fun>
```

Варіанти і рекурсивні структури даних

Інша типова застосування варіантів - подання деревовидних, рекурсивних структур даних. Продемонструємо це на прикладі реалізації простого мови логічних виразів. Така мова може стати в нагоді скрізь, де потрібно організувати фільтрацію, - від засобів аналізу пакетів до програм - клієнтів
електронної пошти.

Вираз на цій мові буде представлено варіантним типом expr, де для кожного елемента виразу визначається свій тег:

```
# type 'a expr =
    | Base of 'a
    | Const of bool
    | And of 'a expr list
    | Or of 'a expr list
    | Not of 'a expr
  ;;
type 'a expr =
    Base of 'a
  | Const of bool
  | And of 'a expr list
  | Or of 'a expr list
  | Not of 'a expr
```

Відзначте, що визначення типу expr є рекурсивним, в тому сенсі що значення типу expr може містити інші значення типу expr. Крім того, тип expr параметризрвані поліморфним типом 'a, використовуваним для визначення типу значення, що відповідає тегу Base.
Призначення кожного тега досить очевидно і без зайвих слів. Теги And, Or і Not - це оператори логічних виразів, а тег Const дозволяє вводити в вираження константи true і false.

Тег Base - це сполучна ланка між expr і вашим додатком, що дає можливість вказувати елементи деякого базового типу предиката, істинність або хибність якого визначаються додатком. Для мови фільтрів, що використовується в додатку обробки електронної пошти, наприклад, базові предикати могли б визначати перевірки електронних листів, як показано нижче:

```
# type mail_field = To | From | CC | Date | Subject
  type mail_predicate = { field: mail_field;
                          contains: string }
  ;;
type mail_field = To | From | CC | Date | Subject
type mail_predicate = { field : mail_field; contains : string; }
```

На основі визначень вище можна сконструювати простий вислів з **`mail_predicate`** в якості базового предиката:

```
# let test field contains = Base { field; contains };;
val test : mail_field -> string -> mail_predicate expr = <fun>
# And [ Or [ test To "doligez"; test CC "doligez" ];
        test Subject "runtime";
      ]
  ;;
- : mail predicate expr =
And
  [Or
    [Base {field = To; contains = "doligez"};
     Base {field = CC; contains = "doligez"}];
   Base {field = Subject; contains = "runtime"}]
```

Взагалі кажучи, застосування варіантів для створення рекурсивних структур даних є поширеною практикою, і приклади такого їх використання можна зустріти де завгодно, від реалізацій простих мов до конструювання складних структур даних.

Поліморфні варіанти

Крім простих варіантів, які ми бачили до сих пір, OCaml підтримує також так звані поліморфні варіанти (polymorphic variants). Як буде показано далі, поліморфні варіанти є більш гнучкими і більш легковажними, ніж звичайні варіанти, але додаткові можливості мають свою ціну.

Синтаксично поліморфні варіанти відрізняються від звичайних початковим зворотним штрихом. І, на відміну від звичайних варіантів, поліморфні варіанти можуть використовуватися без явного оголошення типу:

```
# let three = `Int 3;;
val three : [> `Int of int ] = `Int 3
# let four = `Float 4.;;
val four : [> `Float of float ] = `Float 4.
# let nan = `Not_a_number;;
val nan : [> `Not_a_number ] = `Not_a_number
# [three; four; nan];;
- : [> `Float of float | `Int of int | `Not_a_number ] list =
[`Int 3; `Float 4.; `Not_a_number]
```

Як бачите, типи поліморфних варіантів виводяться автоматично, і коли ми об'єднуємо варіанти з різними тегами, компілятор виводить новий тип, якому відомі всі ці теги. Зверніть увагу, що в цьому прикладі імена тегів (наприклад, `Int) збігаються з іменами типів (int). Це - загальноприйнята угода в Ocaml.

На перший погляд, поліморфні варіанти виглядають як покращена версія звичайних варіантів. З їх використанням можна робити все те ж саме, що і з застосуванням звичайних варіантів, плюс до всього цього ви отримуєте додаткову гнучкість і більш короткий синтаксис. Так що ж не так?

Насправді звичайні варіанти в більшості ситуацій являють собою більш прагматичний вибір, тому що гнучкість поліморфних варіантів змушує платити занадто високу ціну. Нижче перераховані деякі недоліки:

- Складність. Як ми мали можливість переконатися, правила типізації для поліморфних варіантів набагато складніше тих же правил для звичайних варіантів. Це означає, що широке використання поліморфних варіантів може змусити вас міцно чухати потилицю, коли ви будете намагатися зрозуміти, чому цей фрагмент коду компілюється або, навпаки, не компілюється. Це може також змусити вас проводити довгі години в розшифровуванні дивних повідомлень про помилки.

Фактично стисліть на рівні програмного коду часто купується ціною детального опису типів.

- Пошук помилок. Поліморфні варіанти також піддаються строгому контролю типів, але терпимість до друкарські помилки через високу гнучкості знижує шанси компілятора знайти помилки в програмі.
- Ефективність. Хоч це і не найважливіший фактор, але поліморфні варіанти вимагають більше обчислювальних ресурсів, ніж звичайні варіанти, і компілятор OCaml не здатний генерувати код зіставлення для поліморфних варіантів, такий же ефективний, як для звичайних варіантів.

І тим не менше, не дивлячись на все вищесказане, поліморфні варіанти як і раніше залишаються корисною і потужною особливістю мови, просто треба знати і розуміти їх обмеження і особливості ефективного їх використання.
Ймовірно, найкраще поліморфні варіанти використовувати там, де досить було б звичайних варіантів, але вони виявляються занадто важкими синтаксично. Наприклад, часто буває бажано визначити варіантний тип для кодування введення або виведення функції, і немає ніякого бажання оголошувати окремий тип. Поліморфні варіанти в цьому випадку дуже знадобляться вам. І поки ви будете використовувати анотації типів, щоб обмежити їх явно, все буде працювати прекрасно.

Функтори

До сих пір в нашому обговоренні модулі відігравали важливу, але досить обмежену роль. Зокрема, ми розглядали їх як механізм організації коду в одиниці з певними інтерфейсами. Але система модулів в OCaml здатна на більше і може служити потужним інструментом створення універсального коду і структурування великомасштабних систем. Значна частина можливостей зосереджена в функтором.

Функтори, якщо говорити в загальних рисах, - це функції, що відображають модулі на модулі. Вони можуть застосовуватися для вирішення різних проблем структуризації коду, включаючи наступні:

впровадження залежностей - забезпечує можливість заміни деяких компонентів системи. Це може стати в нагоді, наприклад, для підстановки імітацій компонентів системи під час тестування;

- авторозширення модулів - функтори дають можливість додавати в наявні модулі нові функціональні можливості стандартним способом. Наприклад, комусь може знадобитися додати безліч операторів порівняння, заснованих на єдиній базовій функції порівняння. Щоб зробити це вручну, може знадобитися написати масу повторюваного коду для кожного типу, а з функторами досить буде написати логіку тільки один раз і застосувати її до різних типів;
- створення екземплярів модулів із змінним станом - модулі можуть містити змінні значення, відповідно, може знадобитися створити кілька примірників певного модуля, кожен зі своїм змінним станом. Функтори дадуть можливість автоматизувати створення таких модулів.

Це лише деякі з областей застосування функторів. Ми не будемо намагатися осягнути неосяжне і привести всі можливі приклади використання функторів. Замість цього в даній лекції буде зроблена спроба представити приклади, що демонструють мовні особливості і шаблони проектування, які бажано знати, щоб ефективно застосовувати функтори.

Давайте створимо функтор, що приймає модуль з єдиною цілочисельною змінною **x** і повертає новий модуль зі значенням **x** на одиницю більше.

Наша мета - пройтися по основних механізмів функторів, незважаючи на те що практична цінність цього прикладу невисока.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
```

Тепер можна визначити функтор. Ми будемо використовувати тип **X_int** і для визначення аргументу функтора, і для визначення модуля, що повертається функтором:

```
# module Increment (M : X_int) : X_int = struct
let x = M.x + 1
end;;
module Increment : functor (M : X_int) -> X_int
```

Перше, що кидається в очі, - функтори синтаксично більш великовагових, ніж звичайні функції. З одного боку, функтори вимагають явного використання анотацій типів, тоді як для функцій це не є обов'язковим. Технічно обов'язковим є тільки тип вхідного значення, однак на практиці зазвичай бажано також закріплювати тип модуля, що повертається функтором, а також використовувати файл `.mli`, навіть при тому що це не є обов'язковим.

Нижче показано, що трапиться, якщо опустити тип модуля, що повертається функтором:

```
# module Increment (M : X_int) = struct
let x = M.x + 1
end;;
module Increment : functor (M : X_int) -> sig
val x : int end
```

Як бачите, тип модуля виводиться системою типів буквально, а не як посилання на іменовану сигнатуру **`X_int`**.

Ми можемо використовувати функтор **`Increment`** для визначення нових модулів:

```
# module Three = struct let x = 3 end;;
module Three : sig val x : int end
# module Four = Increment(Three);;
module Four : sig val x : int end
# Four.x - Three.x;;
- : int = 1
```

В даному випадку ми застосували функтор **`Increment`** до модуля, сигнатура якого в точності еквівалентна типу **`X_int`**. Однак функтор **`Increment`** можна застосовувати до будь-яких модулів, що задовольняє інтерфейсу **`X_int`** в тих же термінах, в яких вміст файлів **`.ml`** задовольняє інтерфейсів в файлах **`.mli`**. Тобто в типі модуля може бути відсутнім деяка інформація, доступна в самому модулі, або шляхом виключення деяких полів, або шляхом перетворення їх в абстрактні поля. наприклад:

```
# module Three_and_more = struct
let x = 3
let y = "three"
end;;
module Three_and_more : sig val x : int val y :
string end
# module Four = Increment(Three_and_more);;
module Four : sig val x : int end
```

Правила визначення відповідності модуля зазначеної сигнатуре нагадують правила визначення відповідності об'єктів вказаною інтерфейсу в об'єктно-орієнтованих мовах. Як і в об'єктно-орієнтованому контексті, додаткова інформація, яка не відповідає зазначеній сигнатуре (в даному прикладі - змінна **`y`**), просто ігнорується.

Обчислення з використанням інтервалів

Розглянемо більш практичний приклад використання функторів: бібліотеку функцій для обчислень із застосуванням інтервалів. Інтервали часто використовуються в практиці програмування. Вони можуть мати різні типи і задіяні в різних контекстах. Наприклад, може знадобитися організувати обчислення із залученням інтервалів дійсних чисел, рядків або значень часу і в кожному з цих випадків виконувати однотипні операції: перевірку ширини інтервалу, перевірку попадання в інтервал і ін.

Давайте подивимося, як із застосуванням функторів можна реалізувати узагальнену бібліотеку підтримки інтервалів, яку можна було б використовувати з будь-якими типами, до яких застосовне поняття упорядкування.
Перш за все визначимо тип модуля, що описує інформацію про кінцевих точках інтервалу. Цей інтерфейс, який ми назвемо Comparable, містить два елементи: функцію порівняння і тип порівнюваних значень:

```
# module type Comparable = sig
type t
val compare : t -> t -> int
end ;;
module type Comparable = sig type t val
compare : t -> t -> int end
```

Функція порівняння реалізована із застосуванням стандартної ідіоми мови OCaml для таких функцій: вона повертає 0, якщо два елементи рівні; позитивне число, якщо перший елемент більше другого; і негативне число, якщо перший елемент менше другого. Тобто стандартні функції порівняння можна було б переписати на основі **compare**, як показано нижче.

```
compare x y < 0     (* x < y *)
compare x y = 0     (* x = y *)
compare x y > 0     (* x > y *)
```

Створення абстрактних функторів

Функтор **`Make_interval`** має одну проблему. Програмний код, який ми написали, залежить від вимоги, що верхня межа повинна бути більше нижньої, але ця вимога може бути порушено. Дана вимога встановлюється функцією **`create`**, але так як тип **`Interval.t`** не є абстрактним, ми можемо обійти функцію **`create`**:

```
# Int_interval.is_empty (* создание с помощью
create *)
(Int_interval.create 4 3) ;;
- : bool = true
# Int_interval.is_empty (* создание в обход
create *)
(Int_interval.Interval (4,3)) ;;
- : bool = false
```

Функтори насправді - це дуже потужний інструмент організації коду. Проблема лише в тому, що функтори виявляються більш важкими синтаксично, ніж інші конструкції мови, і для їх ефективного використання потрібно знати і розуміти, як вирішувати деякі складні проблеми за допомогою спільно використовуваних обмежень і деструктивної підстановки.

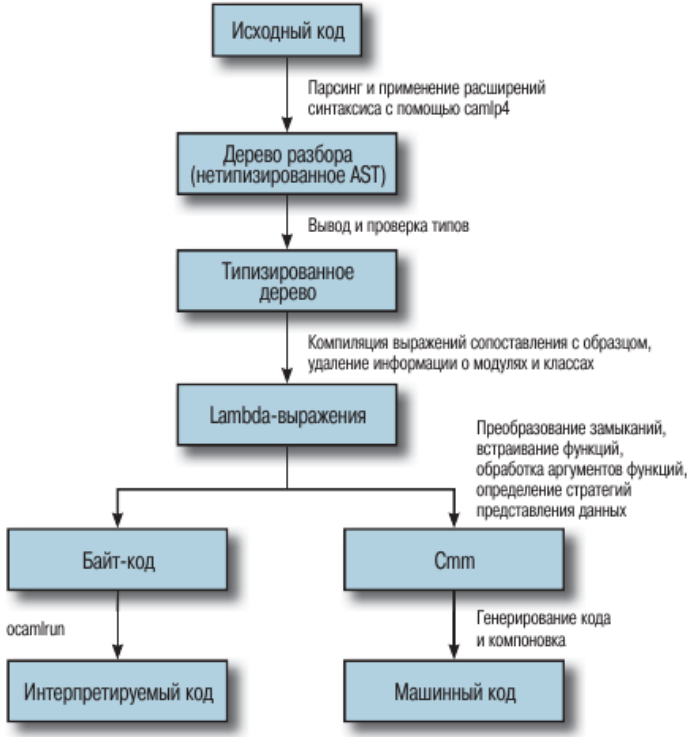
З вищесказаного випливає, що для невеликих і простих програм широке використання функторів напевно буде помилкою. Але в міру зростання складності ваших програм і затребуваності модульної організації коду функтори перетворюються в найцінніший інструмент.

Компіляція вихідного коду в виконувану програму здійснюється цілим комплексом бібліотек, компонувальник і ассемблеров. Важливо розуміти, як вони поєднуються один з одним, допомагаючи вам вирішувати повсякденні завдання по розробці, налагодженні і розгортання додатків.

Мова OCaml є строго універсальна мова і відкидає вихідний код, який не відповідає його вимогам. Досягається це за рахунок послідовності перевірок і трансформацій, які виконуються компілятором. На кожній стадії роботи компілятора вирішується своя задача (наприклад, перевірка типів, оптимізація або створення виконуваного коду) і відкидається деяка частка інформації, отриманої на попередній стадії. На виході виходить машинний код, нічого не підозрюючи про модулях або об'єктах OCaml, з яких починалася компіляція.

Зрозуміло, від вас не потрібно робити щось з перерахованого вручну. Досить запустити команди виклику компілятора (ocamlc і ocamlopt) з командного рядка, і вони автоматично виконають всю послідовність стадій. Однак іноді буває бажано зануритися в процес компіляції глибше звичайного, щоб відловити помилку або спробувати вирішити проблему продуктивності.

Кожен файл з вихідним кодом представляє одиницю компіляції (compilation unit) і компілюється незалежно від інших файлів. Компілятор генерує проміжні файли з різними розширеннями імен файлів для використання на наступних етапах компіляції. Компоновщик (linker) приймає колекцію
скомпільованих файлів і виробляє автономний виконуваний файл або бібліотеку, яку можна використовувати в інших додатках.


Исходный код
Парсинг и применение расширений синтаксиса с помощью camlp4
Дерево разбора (нетипизированное AST)
Вывод и проверка типов
Типизированное дерево
Компиляция выражений сопоставления с образцом, удаление информации о модулях и классах
Lambda-выражения
Преобразование замыканий, встраивание функций, обработка аргументов функций, определение стратегий представления данных
Байт-код
Cmm
ocamlrun
Генерирование кода и компоновка
Интерпретируемый код
Машинный код

На наступній лекції ми розглянемо мультипарадигмену мову програмування Oz.