Wykład 7

Abstrakcyjne typy danych.

Moduły i funktory w języku OCaml.

Komponenty programowe
Abstrakcyjne typy danych
Algebry abstrakcyjne
Algebraiczna specyfikacja stosu
Stos jako zbiór funkcji
Moduły w języku OCaml
Wykorzystanie sygnatur (interfejsów) do ukrywania reprezentacji
Moduły jako jednostki kompilacji - OCaml
Funktory w języku OCaml
Przykład funktora – słownik jako binarne drzewo poszukiwań

Komponenty programowe

• Większe programy dzielimy na składowe lub komponenty programowe (ang. software components).

• Komponenty mogą być różnego rodzaju, np. moduły (ang. modules), klasy (ang. classes), pakiety (ang. packages), biblioteki (ang. libraries), procesy (ang. processes), usługi sieciowe (ang. web services). Ich rozmiary mogą się wahać od kilku wierszy kodu do setek lub tysięcy wierszy.

• Każdy komponent posiada dwie części – *interfejs* (ang. interface) oraz *implementację* (ang. implementation). Poza komponentem widoczny jest wyłącznie interfejs. Komponent może używać innych komponentów jako części swojej implementacji.

• Program jest w takim przypadku grafem skierowanym (zorientowanym) komponentów. Krawędź od jednostki A do jednostki B oznacza, że A potrzebuje B w swojej implementacji.

• Komponenty można łączyć za pomocą wielu mechanizmów, np. agregacja (ang. aggregation), parametryzacja (ang. parameterization), dziedziczenie (ang. inheritance), zdalne wywołanie (ang. remote invocation), przekazywanie komunikatów (ang. message passing).

Abstrakcyjny typ danych

Nieformalnie abstrakcja danych umożliwia korzystanie z danych w sposób abstrakcyjny, czyli bez zajmowania się ich implementacją.

Abstrakcyjny typ danych (ang. abstract data type = ADT) składa się z dobrze wyspecyfikowanego *zbioru elementów* oraz *zbioru operacji*, które mogą być wykonywane na tym zbiorze elementów. Przykłady ATD: stos, kolejka, graf, a także liczby całkowite, liczby rzeczywiste, wartości logiczne.

Specyfikacja ADT **nie** opisuje wewnętrznej reprezentacji zbioru elementów ani sposobu implementacji operacji, która powinna być ukryta (*hermetyzacja*, ang. encapsulation).

Abstrakcyjne typy danych i moduły

Podstawę abstrakcyjnych typów danych stanowi *rozdzielenie* (publicznego) *interfejsu i* (ukrytej) *implementacji*.

Systemy wspierające stosowanie abstrakcji (w szczególności ADT) powinny posiadać trzy własności:

- *Hermetyzacja* (ang. encapsulation) umożliwia ukrycie wnętrza części systemu.
- *Kompozycyjność* (ang. compositionality) umożliwia tworzenie nowych elementów systemu przez łączenie istniejących części.
- *Konkretyzacja/wywołanie* (ang. instantiation/invocation) umożliwia tworzenie wielu egzemplarzy elementu systemu w oparciu o tę samą definicję.

ADT można wygodnie implementować, np. jako moduły lub klasy.

Można wyróżnić dwa aspekty programowania, związane z modułami:

- *fizyczna* dekompozycja programu na pliki, które mogą być oddzielnie kompilowane i wielokrotnie używane;
- *logiczna* dekompozycja programu, ułatwiająca projektowanie i zrozumienie programu (wyższy poziom abstrakcji).

W języku OCaml można programować używając modułów i klas, co pozwala na porównanie tych mechanizmów i związanych z nimi technik programowania.

Algebra abstrakcyjna (uniwersalna)

Wiele abstrakcyjnych typów danych można wyspecyfikować jako algebry abstrakcyjne.

Algebrą abstrakcyjną nazywamy *zbiór elementów S* (nośnik, dziedzina lub uniwersum algebry), na którym są zdefiniowane pewne *operacje*, posiadające własności zadane przez *aksjomaty* równościowe.

Przykłady algebr abstrakcyjnych (homogenicznych)

- *Półgrupa* $\langle S, \bullet \rangle$ $\bullet : S \times S \to S$

  $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ (łączność)

- *Monoid* $\langle S, \bullet, 1 \rangle$ jest półgrupą z obustronną jednością (elementem neutralnym) $1 : S$

  $a \bullet 1 = a \qquad 1 \bullet a = a$

- *Grupa* $\langle S, \bullet, \bar{\ }, 1 \rangle$ jest monoidem, w którym każdy element posiada element odwrotny względem binarnej operacji monoidu $\bar{\ } : S \to S$

  $a \bullet \bar{a} = 1 \qquad \bar{a} \bullet a = 1$

Algebraiczna specyfikacja stosu (heterogeniczna)

Sygnatura:

```
empty   : -> Stack
push    : Elem * Stack -> Stack
top     : Stack -> Elem
pop     : Stack -> Stack
isEmpty : Stack -> bool
```

Aksjomaty równościowe:

∀ s:Stack, e: Elem

```
isEmpty (push (e,s)) = false
isEmpty (empty)      = true
pop (push(e,s))      = s
pop (empty)          = empty
top (push(e,s))      = e
top (empty)          = ERROR
```

Stos jako zbiór funkcji - OCaml

Stos można zaimplementować w języku OCaml jako typ algebraiczny z dwoma konstruktorami, przepisując prawie dosłownie powyższą specyfikację algebraiczną.

```
# type 'a stack = EmptyStack | Push of 'a * 'a stack;;
type 'a stack = EmptyStack | Push of 'a * 'a stack

# exception Empty of string;;
exception Empty of string

# let empty() = EmptyStack;;
val empty : unit -> 'a stack = <fun>

# let push(e,s) = Push(e,s);;
val push : 'a * 'a stack -> 'a stack = <fun>

# let isEmpty s = function
    EmptyStack -> true
  | Push _       -> false;;
val isEmpty : 'a stack -> bool = <fun>
```

Stos jako zbiór funkcji - OCaml

```
# let pop = function
    Push(_,s)     -> s
  | EmptyStack  -> EmptyStack;;
val pop : 'a stack -> 'a stack = <fun>

# let top = function
    Push(e,_)     -> e
  | EmptyStack  ->  raise (Empty "Stack: top");;
val top : 'a stack -> 'a = <fun>

# let s = Push(3,Push(2,Push(1,EmptyStack)));; (* można użyć reprezentacji wewnętrznej zamiast
                                                  funkcji empty i push – niedobrze! *)
val s : int stack = Push (3, Push (2, Push (1, EmptyStack)))

# pop s;;
-: int stack = Push (2, Push (1, EmptyStack))

# top s;;
-     : int = 3
```

Ta implementacja w sposób oczywisty spełnia specyfikację algebraiczną, ale udostępnia klientom reprezentację wewnętrzną stosu i stanowi zbiór nie związanych funkcji, znajdujących się w globalnej przestrzeni nazw.

Analogicznie stos można reprezentować jako listę.

Moduły w języku OCaml

Język OCaml posiada wygodne wsparcie lingwistyczne dla modułów.

module *Nazwa* = **struct** *definicje typów i wartości* **end**

Nazwa modułu musi zaczynać się z wielkiej litery.
Moduły (struktury), podobnie jak funkcje mogą być anonimowe.

struct *definicje typów i wartości* **end**

Kompilator sam tworzy domyślny interfejs (sygnaturę) struktury, umieszczając w niej definicje typów i typy wartości.

Dostęp do typów i wartości zdefiniowanych w module można uzyskać używając notacji kropkowej:

NazwaModułu.nazwaSkładowej

Identyfikatory użyte w module są lokalne. Moduł tworzy własną przestrzeń nazw.

Można też moduł otworzyć: **open** *Nazwa*
Grozi to jednak przesłonięciem własnych identyfikatorów. Lepiej moduł otwierać lokalnie:
let open *Nazwa* **in** *wyrażenie* lub równoważnie *Nazwa*.(*wyrażenie*)
Np. let open List in tl [1;2;3;4];; lub List.(tl [1;2;3;4]);;

Stos jako moduł - OCaml

```
# module Stack' =
struct
  type 'a t = EmptyStack  |  Push of  'a * 'a t
  exception Empty of string
  let create() = EmptyStack
  let push(e,s) = Push(e,s)
  let top = function   Push(e,_) -> e  | EmptyStack -> raise (Empty "module Stack': top")
  let pop = function   Push(_,s) -> s  | EmptyStack -> EmptyStack
  let isEmpty s =  s = EmptyStack
end;;

module Stack' :
  sig                                          (* To jest wygenerowany przez kompilator interfejs (sygnatura) modułu (struktury)  *)
    type 'a t = EmptyStack | Push of 'a * 'a t   (* W sygnaturze widoczny jest typ reprezentujący stos, w konsekwencji
                                                    na zewnątrz widoczna będzie reprezentacja stosu  *)
    exception Empty of string
    val create :  unit -> 'a t
    val push : 'a * 'a t -> 'a t
    val top : 'a t -> 'a
    val pop : 'a t -> 'a t
    val isEmpty : 'a t -> bool
  end
# let s = Stack'.push(2,Stack'.push(1,Stack'.create()));;
val s : int Stack'.t = Stack'.Push (2, Stack'.Push (1, Stack'.EmptyStack))    (* Widoczna jest reprezentacja stosu  *)
```

Sygnatury (interfejsy) w języku OCaml

Definicja sygnatury (interfejsu):

module type *NAZWA* =
 sig
  składowe sygnatury
 end

Specyfikacja składowych wartości sygnatury: **val** *nazwa* **:** *typ*
Nazwa sygnatury jest dowolna, ale w OCamlu na mocy konwencji używane są wyłącznie duże litery.
Sygnatury mogą być też anonimowe.
sig *składowe sygnatury* **end**

Definicja struktury (modułu), spełniającego zadaną sygnaturę:

module *NazwaModułu: SYGNATURA* =
 struct
  definicje typów i wartości
 end

lub

module *Nazwa* = **(** *struktura* : *sygnatura* **)**

Taki moduł w pełni hermetyzuje swoją zawartość. Na zewnątrz widać tylko to, co pokazuje sygnatura.

Sygnatura stosu - OCaml

```
# module type STACK_FUN =
sig
  type 'a t
  exception Empty of string
  val create: unit -> 'a t
  val push: 'a * 'a t -> 'a t
  val top: 'a t -> 'a
  val pop: 'a t -> 'a t
  val isEmpty: 'a t -> bool
end;;

module type STACK_FUN =  (* To jest odpowiedź kompilatora, potwierdzająca poprawność *)
  sig                    (* definicji sygnatury                                      *)
    type 'a t
    exception Empty of string
    val create : unit -> 'a t
    val push : 'a * 'a t -> 'a t
    val top : 'a t -> 'a
    val pop : 'a t -> 'a t
    val isEmpty : 'a t -> bool
  end
```

Implementacja stosu (moduł) - OCaml

```
# module Stack : STACK_FUN =
struct
  type 'a t = EmptyStack | Push of 'a * 'a t
  exception Empty of string

  let create() = EmptyStack
  let push(e,s) = Push(e,s)

  let top = function
      Push(e,_) -> e
    | EmptyStack -> raise (Empty "module Stack: top")

  let pop = function
      Push(_,s) -> s
    | EmptyStack -> EmptyStack

  let isEmpty s = s = EmptyStack
end;;

module Stack : STACK_FUN     (* To jest odpowiedź kompilatora *)
```

Wykorzystanie modułu - OCaml

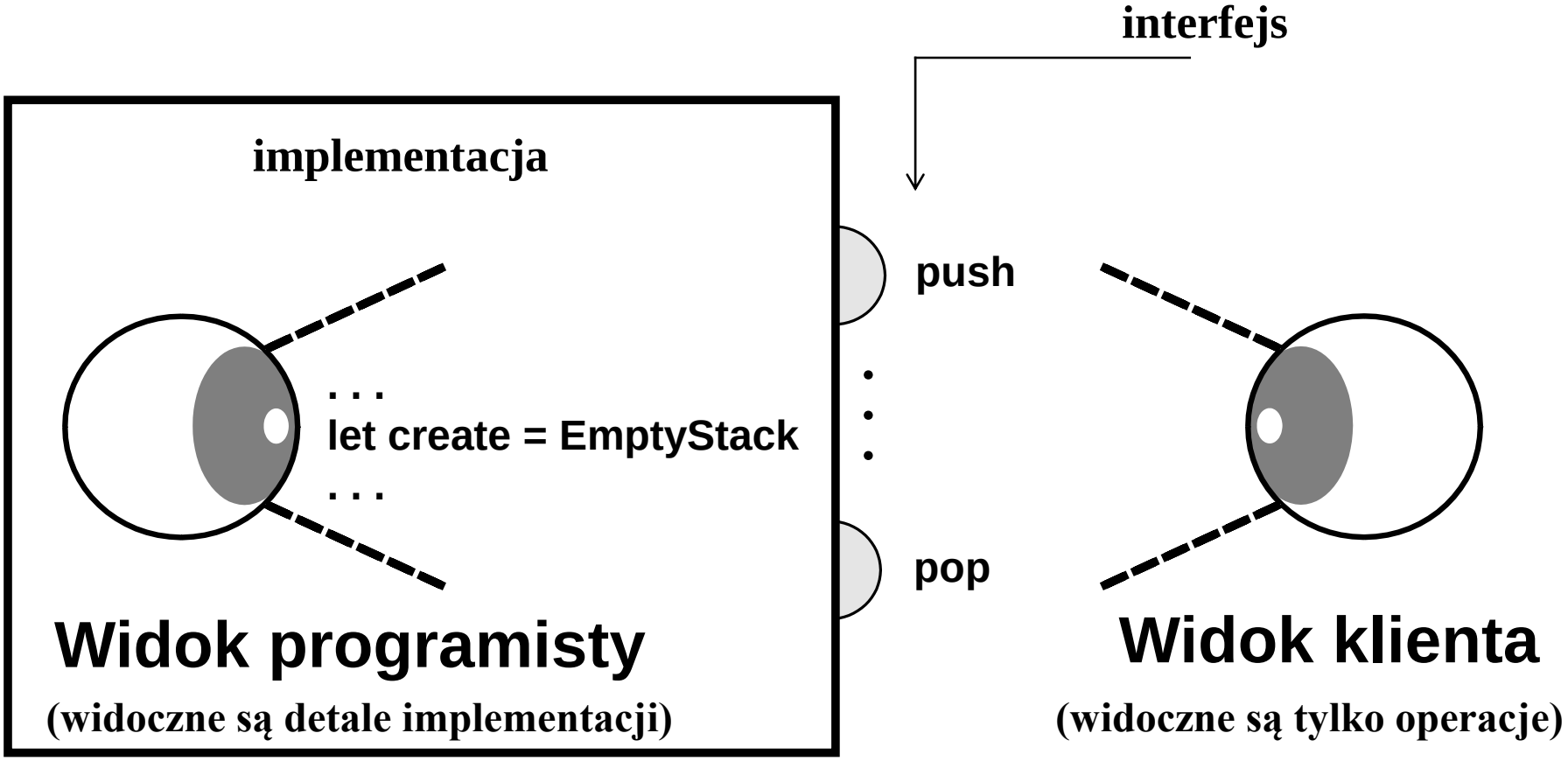
```
# let s = let open Stack in push(2,push(1,create()));;
(* lub
let s = Stack.(push(2,push(1,create())));;  lub
let s = Stack.push(2,Stack.push(1,Stack.create()));;  *)
val s : int Stack.t = <abstr> (* Reprezentacja stosu jest niewidoczna *)
# Stack.top s;;
- : int = 2
# Stack.top (Stack.pop(Stack.pop s));;
Exception: Stack.Empty "module Stack: top".
```

Użycie sygnatury umożliwiło ukrycie reprezentacji wewnętrznej stosu i sposobu implementacji operacji (hermetyzacja).

Zwykle moduły są komponentami statycznymi, istotnymi z punktu widzenia organizacji (struktury) programu, a nie samych obliczeń. W języku OCaml moduł może być spakowany jako *wartość pierwszej kategorii*, która może dynamicznie zostać rozpakowana jako moduł (nie będzie o tym mowy na wykładzie).

Dwa widoki abstrakcyjnego typu danych
(na przykładzie stosu)

ATD Stack



Bezpieczne ADT w języku OCaml

OCaml wykorzystuje statyczną typizację do hermetyzacji i pełnej ochrony wewnętrznej reprezentacji wartości ADT, np. wartości stosu, przed nieupoważnionymi działaniami. Zauważmy, że sygnatura jest „typem" modułu.

język podstawowy		moduły
typ	~	sygnatura (typ modułu, interfejs)
wartość	~	struktura (moduł, implementacja)
funkcja	~	funktor

Sygnatura stosu modyfikowalnego

```
# module type STACK_MUT =
sig
  type 'a t
  exception Empty of string
  val create: unit -> 'a t
  val push: 'a * 'a t -> unit
  val top: 'a t -> 'a
  val pop: 'a t -> unit
  val isEmpty: 'a t -> bool
end;;
module type STACK_MUT =
  sig
    type 'a t
    exception Empty of string
    val create : unit -> 'a t
    val push : 'a * 'a t -> unit
    val top : 'a t -> 'a
    val pop : 'a t -> unit
    val isEmpty : 'a t -> bool
  end
```

Implementacja stosu modyfikowalnego na liście

```
# module StackMutList =
struct
  type 'a t = { mutable l : 'a list }
  exception Empty of string

  let create() = { l = [] }
  let push(e,s) = s.l <- e :: s.l

  let top s =
    match s.l with
      hd::_ -> hd
    | []    -> raise (Empty "module StackMutList: top")

  let pop s =
    match s.l with
      hd::tl -> s.l <- tl
    | []     -> ()

  let isEmpty s = s.l = []

end;;
```

Stos modyfikowalny na liście - odpowiedź systemu

```
module StackMutList :
  sig
    type 'a t = { mutable l : 'a list; }
    exception Empty of string
    val create : unit -> 'a t
    val push : 'a * 'a t -> unit
    val top : 'a t -> 'a
    val pop : 'a t -> unit
    val isEmpty : 'a t -> bool
  end
# let s = StackMutList.create();;
val s : '_a StackMutList.t = {StackMutList.l = []}
# StackMutList.push(1,s);;
- : unit = ()
# StackMutList.push(2,s);;
- : unit = ()
# StackMutList.top s;;
- : int = 2
# StackMutList.pop s;;
- : unit = ()
# StackMutList.pop s;;
- : unit = ()
# StackMutList.top s;;
Exception: StackMutList.Empty "module StackMutList: top".
```

Implementacja stosu modyfikowalnego na tablicy

```
# module StackMutAr =
struct
  type 'a t = { mutable n : int; mutable a : 'a option array }
  exception Empty of string
  let size = 5
  let create() = { n=0 ; a = Array.make size None }
  let increase s = s.a <- Array.append s.a (Array.make size None)
  let push(e,s) = begin if s.n = Array.length s.a then increase s;
                        s.a.(s.n) <- Some e ;
                        s.n <- succ s.n
                  end
  let top s = if s.n=0 then raise (Empty "module StackMutAr: top")
              else match s.a.(s.n-1) with
                      Some e -> e
                    | None -> failwith
                        "module StackMutAr: top (implementation error!!!)"
  let pop s = if s.n=0 then () else s.n <- pred s.n
  let isEmpty s = s.n=0
end;;
```

Stos modyfikowalny na tablicy – odpowiedź systemu

```
 module StackMutAr :
  sig
    type 'a t = { mutable n : int; mutable a : 'a option array; }
    exception Empty of string
    val size : int
    val create : unit -> 'a t
    val increase : 'a t -> unit
    val push : 'a * 'a t -> unit
    val top : 'a t -> 'a
    val pop : 'a t -> unit
    val isEmpty : 'a t -> bool
  end
# let s = StackMutAr.create();;
val s : '_a StackMutAr.t =
  {StackMutAr.n = 0; StackMutAr.a = [|None; None; None; None; None|]}
# StackMutAr.push(1,s);;
- : unit = ()
# StackMutAr.top s;;
- : int = 1
# StackMutAr.pop s;;
- : unit = ()
# StackMutAr.top s;;
Exception: StackMutAr.Empty "module StackMutAr: top".
```

Dwa moduły dla stosów (1)

Moduły reprezentują typ *t* z sygnatury w różny sposób.

```
# StackMutList.create();;
- : '_a StackMutList.t = {StackMutList.l = []}
# StackMutAr.create();;
- : '_a StackMutAr.t ={StackMutAr.n = 0; StackMutAr.a = [|None; None; None;
                                                          None; None|]}
```

Można ukryć reprezentację dla abstrakcyjnego typu danych wykorzystując sygnaturę.

```
# module SML : STACK_MUT = StackMutList;;
module SML : STACK_MUT
# module SMA : STACK_MUT = StackMutAr;;
module SMA : STACK_MUT
# let sl = SML.create();;
val sl : '_a SML.t = <abstr>
# let sa = SMA.create();;
val sa : '_a SMA.t = <abstr>
```

Przykład: dwa moduły dla stosów (2)

Oba moduły implementują ten sam interfejs, ale typy reprezentacji są różne.

```
# SMA.isEmpty sl;;

Characters 12-14:
  SMA.isEmpty sl;;
              ^^
Error: This expression has type 'a SML.t
       but an expression was expected of type 'b SMA.t
```

Nawet gdyby typy reprezentacji były takie same, to użycie sygnatury spowodowało ukrycie tej reprezentacji.

```
# module SL1 = (StackMutList : STACK_MUT);;
module SL1 : STACK_MUT

# module SL2 = (StackMutList : STACK_MUT);;
module SL2 : STACK_MUT

# let s = SL1.create();;
val s : '_a SL1.t = <abstr>

# SL2.isEmpty s;;
Characters 12-13:
  SL2.isEmpty s;;
              ^
Error: This expression has type 'a SL1.t
       but an expression was expected of  type 'b SL2.t
```

Różne widoki modułów (1)

```
#module M =
( struct
    type buffer = int ref
    let create() = ref 0
    let add x = incr x
    let get x = if !x>0 then (decr x; 1) else failwith "Empty"
  end
: sig
    type buffer
    val create : unit -> buffer
    val add : buffer -> unit
    val get : buffer -> int
  end
) ;;
module M :
  sig
    type buffer
    val create : unit -> buffer
    val add : buffer -> unit
    val get : buffer -> int
  end
```

Różne widoki modułów (2)

```
# module type PRODUCER =
sig
  type buffer
  val create : unit -> buffer
  val add : buffer -> unit
end ;;
module type PRODUCER = sig type buffer val create : unit -> buffer
                           val add : buffer -> unit end
# module type CONSUMER =
sig
  type buffer
  val get : buffer -> int
end ;;
module type CONSUMER = sig type buffer val get : buffer -> int end
# module Producer = (M:PRODUCER) ;;
module Producer : PRODUCER
# module Consumer = (M:CONSUMER) ;;
module Consumer : CONSUMER
```

Niestety, moduły Producer i Consumer nie mogą ze soba współpracować!

```
#let buf = Producer.create() in Producer.add buf; Consumer.get buf;;
                                                                ^^^
Error: This expression has type Producer.buffer
       but an expression was expected of type Consumer.buffer
```

Współdzielenie typów w modułach

W celu utożsamienia typów `Producer.t` i `Consumer.t` trzeba użyć poniższej konstrukcji językowej:

NAZWA **with type** *t1* **=** *t2* **and ...**

```
# module Producer = (M:PRODUCER with type buffer = M.buffer) ;;

module Producer : sig
                    type buffer = M.buffer
                    val create : unit -> buffer
                    val add : buffer -> unit

                  end

# module Consumer = (M:CONSUMER with type buffer = M.buffer) ;;

module Consumer : sig type buffer = M.buffer val get : buffer -> int end

# let buf = Producer.create();;
val buf: Producer.buffer = <abstr>

# Producer.add buf; Producer.add buf;;
- : unit = ()

# Consumer.get buf;;
- : int = 1

# Consumer.get buf;;
- : int = 1

# Consumer.get buf;;
Exception: Failure "Empty".
```

Współdzielenie typów i moduły wewnętrzne (1)

```
# module M1 =
( struct
        type buffer = int ref
        module M_hide =
        struct
           let create() = ref 0
           let add x = incr x
           let get x = if !x>0 then (decr x; 1)
                               else failwith"Empty"
        end
    module Producer = M_hide
    module Consumer = M_hide
end
:
sig
  type buffer
  module Producer : sig val create : unit -> buffer
                        val add : buffer -> unit end
  module Consumer : sig val get : buffer -> int end
end
) ;;
```

Współdzielenie typów i moduły wewnętrzne (2)

```
module M1 :
 sig
    type buffer
    module Producer : sig val create : unit -> buffer
                          val add : buffer -> unit
                      end
    module Consumer : sig val get : buffer -> int end
  end
```

Można teraz osiągnąć ten sam rezultat, który osiągnęliśmy za pomocą konstrukcji **with type**, chociaż dostęp do funkcji modułów Producer i Consumer odbywa się za pośrednictwem modułu M1 (można go jednak otworzyć):

```
 # let buf = M1.Producer.create();;
val buf : M1.buffer = <abstr>
# M1.Producer.add buf;
- : unit = ()
# M1.Consumer.get buf;;
- : int = 1
# M1.Consumer.get buf;;
Exception: Failure "Empty".
```

Moduły i oddzielna kompilacja

Notacja wprowadzona dla sygnatur i modułów odnosiła się do programów monolitycznych, które mogą fizycznie być podzielone na pliki, ale są kompilowane jako całość.

Jednostka kompilacji *K* składa się z dwóch plików:

- pliku z implementacją *K*`.ml`, który jest ciągiem definicji znajdujących się w programach monolitycznych między słowami kluczowymi `struct` ... `end` (ale bez tych słów);
- pliku z interfejsem *K*`.mli` (opcjonalnie), który jest ciągiem specyfikacji znajdujących się w programach monolitycznych między słowami kluczowymi `sig` ... `end` (ale bez tych słów).

Inna jednostka kompilacji *L* może się odwoływać do *K* jak do struktury w programie monolitycznym, używając notacji kropkowej *K.x*.

Przykład (wykonywać w oknie komend).

Pliki źródłowe (są w folderze „oddzielnie”): stack.mli, stack.ml, stackTest.ml.

Kroki kompilacji:

```
ocamlc -c stack.mli                                  (tworzy plik stack.cmi)
ocamlc -c stack.ml                                   (tworzy plik stack.cmo)
ocamlc -c stackTest.ml                               (tworzy plik stackTest.cmo)
ocamlc -o stackTest stack.cmo stackTest.cmo          (łączenie plików obiektowych,
                                                      kolejność jest istotna!)
```

Uruchamianie powstałego programu (w kodzie pośrednim, ang. bytecode):

```
ocamlrun stackTest
```

Możliwa jest też kompilacja do kodu rodzimego (ang. native code), patrz „OCaml manual”.

Funktory - składnia

Funktory można definiować podobnie jak funkcje:

functor (*Nazwa* **:** *sygnatura***) ->** *struktura*

```
# module Para = functor (El : sig type t end) ->
                                struct type para = El.t * El.t end;;
module Para :
  functor (El : sig type t end) -> sig type para = El.t * El.t end
```

Podobnie jak dla funkcji można użyć skrótu notacyjnego:

module *Nazwa1* **(***Nazwa2* **:** *sygnatura***) =** *struktura*

```
# module Para(El : sig type t end) =
                                struct type para = El.t * El.t end;;
module Para :
  functor (El : sig type t end) -> sig type para = El.t * El.t end
```

Funktory - składnia

Funktor może mieć dowolną liczbę parametrów:

functor (*Nazwa1* **:** *sygnatura1***) ->** ... **functor (***Nazwan* **:** *sygnaturan***) ->** *struktura*

Tu również można użyć skrótu:

module *Nazwa* **(***Nazwa1* **:** *sygnatura1***)** ... **(***Nazwan* **:** *sygnaturan***) =** *struktura*

Aplikacja funktora do argumentów jest zapisywana zgodnie z poniższą składnią (każdy argument musi być umieszczony w nawiasach):

module *Nazwa* **=** *funktor* **(***struktura1***)** ... **(***strukturan***)**

Funktory zapisujemy zawsze w postaci rozwiniętej. Nie ma odpowiednika postaci zwiniętej dla funkcji.

Słowniki

Słownikiem (ang. dictionary) nazywamy abstrakcyjny typ danych z operacjami wstawiania elementu do zbioru (insert), usuwania elementu ze zbioru (delete), oraz wyszukiwania elementu w zbiorze (lookup, search). Często przyjmuje się założenie, że klucze słownika należą do zbioru liniowo uporządkowanego. Słownik można reprezentować jako listę lub tablicę asocjacyjną. Efektywnymi strukturami służącymi do reprezentowania słowników są tablice z haszowaniem.

Jako przykład napiszemy funktor dla słownika, reprezentowanego przez binarne drzewo poszukiwań. Binarne drzewa poszukiwań nie są polimorficzne względem typu klucza — na zbiorze kluczy musi być zdefiniowany porządek liniowy. W języku OCaml (lub SML) można to wyrazić formalnie, parametryzując słownik modułem dla klucza, spełniającego odpowiednią sygnaturę `ORDER`. Sygnatura zawiera funkcję `compare: t -> t -> ordering`, porównującą dwa klucze i zwracającą jedną z wartości: `LT | EQ | GT`.

```
module type ORDER =
sig
  type t
  val compare: t -> t -> ordering
end;;
```

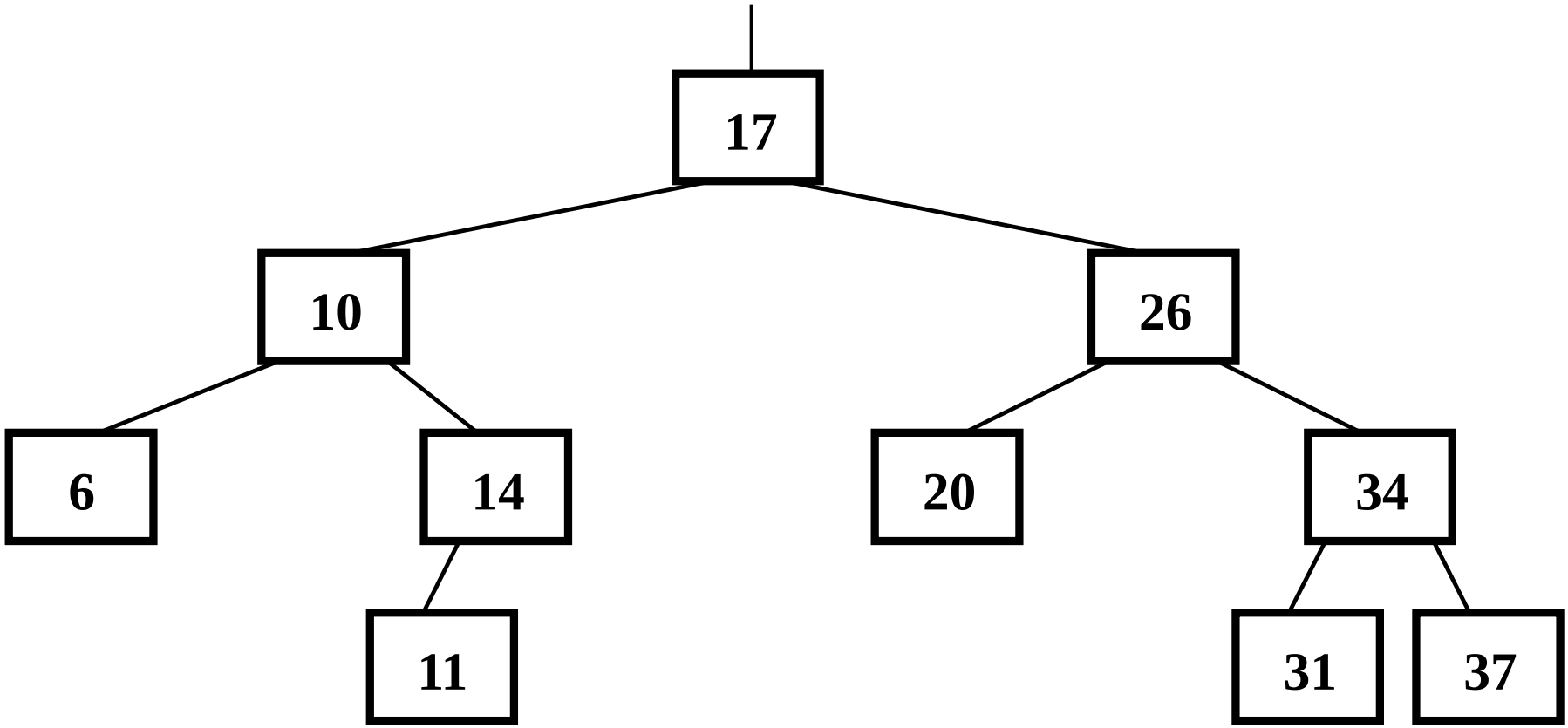
Definicje pomocnicze

```
# type ordering = LT | EQ | GT;;
(** Linearly ordered types **)
# module type ORDER =
sig
  type t
  val compare: t -> t -> ordering
end;;
module type ORDER = sig type t val compare : t->t->ordering end
# module StringOrder: ORDER with type t = string =
struct
  type t = string
  let compare s1 s2 = if s1<s2 then LT else
                      if s1>s2 then GT else EQ
end;;
module StringOrder : sig type t = string
                         val compare : t -> t -> ordering end
```

Sygnatura dla słownika

```
# module type DICTIONARY =
sig
  type key                                (* type of keys *)
  type 'a t                               (* type of dictionaries *)
  exception DuplicatedKey of key         (* error in insert *)
  val empty: unit -> 'a t                 (* empty dictionary *)
  val lookup: 'a t -> key -> 'a option
  val insert: 'a t -> key * 'a -> 'a t
  val delete: 'a t -> key -> 'a t
  val update: 'a t -> key * 'a -> 'a t   (* not necessary *)
end;;
```

Drzewo poszukiwań binarnych



Słownik jako binarne drzewo poszukiwań (1)

```
module Dictionary (Key: ORDER) : DICTIONARY with type key = Key.t =
struct
  type key = Key.t
  type 'a t = Tip | Node of key * 'a * 'a t * 'a t
    exception DuplicatedKey of key
    let empty() = Tip

     let rec lookup tree key =
        match tree with
          Node(k,info,t1,t2) ->
            (match Key.compare key k with
               LT  -> lookup t1 key
             | EQ  -> Some info
             | GT  -> lookup t2 key
            )
        | Tip -> None
     ;;
```

Słownik jako binarne drzewo poszukiwań (2)

```
let rec insert tree (key, value) =
  match tree with
     Tip -> Node(key, value, Tip, Tip)
   | Node(k,info,t1,t2) ->
       (match Key.compare key k with
           LT -> Node(k, info, insert t1 (key,value), t2)
         | EQ   -> raise (DuplicatedKey key)
         | GT    -> Node(k, info, t1, insert t2 (key,value))
        )
;;
```

Drzewo nie jest w żaden sposób wyważane, ponieważ cel przykładu jest inny.

Słownik jako binarne drzewo poszukiwań (3)

```
(* deletemin T returns a triple consisting of the least
   element y in tree T, its associated value and the tree
   that results from deleting y from T.
*)
 let rec deletemin tree =
  match tree with
    Node(k,info,Tip,t2) -> (k,info,t2)
      (* This is the critical case. If the left subtree
         is empty, then the element at the current node
         is the min. *)
  | Node(k,info,t1,t2) ->
      let (key,value,l) = deletemin t1
      in (key,value,Node(k,info,l,t2))
  | Tip -> failwith "Dictionary: implementation error"
;;
```

Słownik jako binarne drzewo poszukiwań (4)

```
let rec delete tree key =
    match tree with
          Tip ->  Tip
        | Node(k,info,t1,t2) ->
            match Key.compare key k with
                LT  -> Node(k,info, delete t1 key, t2)
              | EQ  ->
                  ( match (t1, t2) with
                       (Tip, t2) -> t2
                     | (t1, Tip)  -> t1
                     | _  -> let
                                 (ki,inf,t_right) = deletemin t2
                             in Node(ki,inf,t1,t_right)

                  )
              | GT -> Node(k,info, t1, delete t2 key)
    ;;
```

Słownik jako binarne drzewo poszukiwań (5)

```
let rec update tree (key, value) =
   match tree with
       Tip -> Node(key, value, Tip, Tip)
     | Node(k,info,t1,t2) ->
          ( match Key.compare key k with
              LT  -> Node(k, info, update t1(key,value), t2)
            | EQ  -> Node(k, value, t1, t2)
            | GT  -> Node(k, info, t1, update t2(key,value))
           )
   ;;
 end;; (* Dictionary *)
 module Dictionary : functor (Key : ORDER) ->
   sig
     type key = Key.t
     and 'a t
     exception DuplicatedKey of key
     val empty : unit -> 'a t
     val lookup : 'a t -> key -> 'a option
     val insert : 'a t -> key * 'a -> 'a t
     val delete : 'a t -> key -> 'a t
     val update : 'a t -> key * 'a -> 'a t
   end
```

Wykorzystanie funktora Dictionary (1)

```
# module StringDict = Dictionary(StringOrder);;
module StringDict :
  sig
    type key = StringOrder.t
    and 'a t = 'a Dictionary(StringOrder).t
    exception DuplicatedKey of key
    val empty : unit -> 'a t
    val lookup : 'a t -> key -> 'a option
    val insert : 'a t -> key * 'a -> 'a t
    val delete : 'a t -> key -> 'a t
    val update : 'a t -> key * 'a -> 'a t
  end

# let ( <| ) d (k,x) = StringDict.update d (k,x);;
val ( <| ) : 'a StringDict.t -> StringDict.key * 'a
                              -> 'a StringDict.t = <fun>
```

Przypomnienie. W języku OCaml każdy operator infiksowy `op` można zamienić na funkcję w postaci rozwiniętej przez umieszczenie go w nawiasach: `(op)` . Można też definiować własne operatory infiksowe, jak w omawianym przykładzie.

Wykorzystanie funktora Dictionary (2)

```
# let dict = StringDict.empty();;
val dict : '_a StringDict.t = <abstr>

# let dict = dict <| ("kot","cat")
                  <| ("slon","elephant")
                  <| ("pies","dog")
                  <| ("ptak","bird")
;;
val dict : string StringDict.t = <abstr>
# StringDict.lookup dict "pies";;
- : string option = Some "dog"
# StringDict.lookup dict "papuga";;
- : string option = None
# let dict = dict <| ("papuga","parrot");;
val dict : string StringDict.t = <abstr>
# StringDict.lookup dict "papuga";;
- : string option = Some "parrot"
```

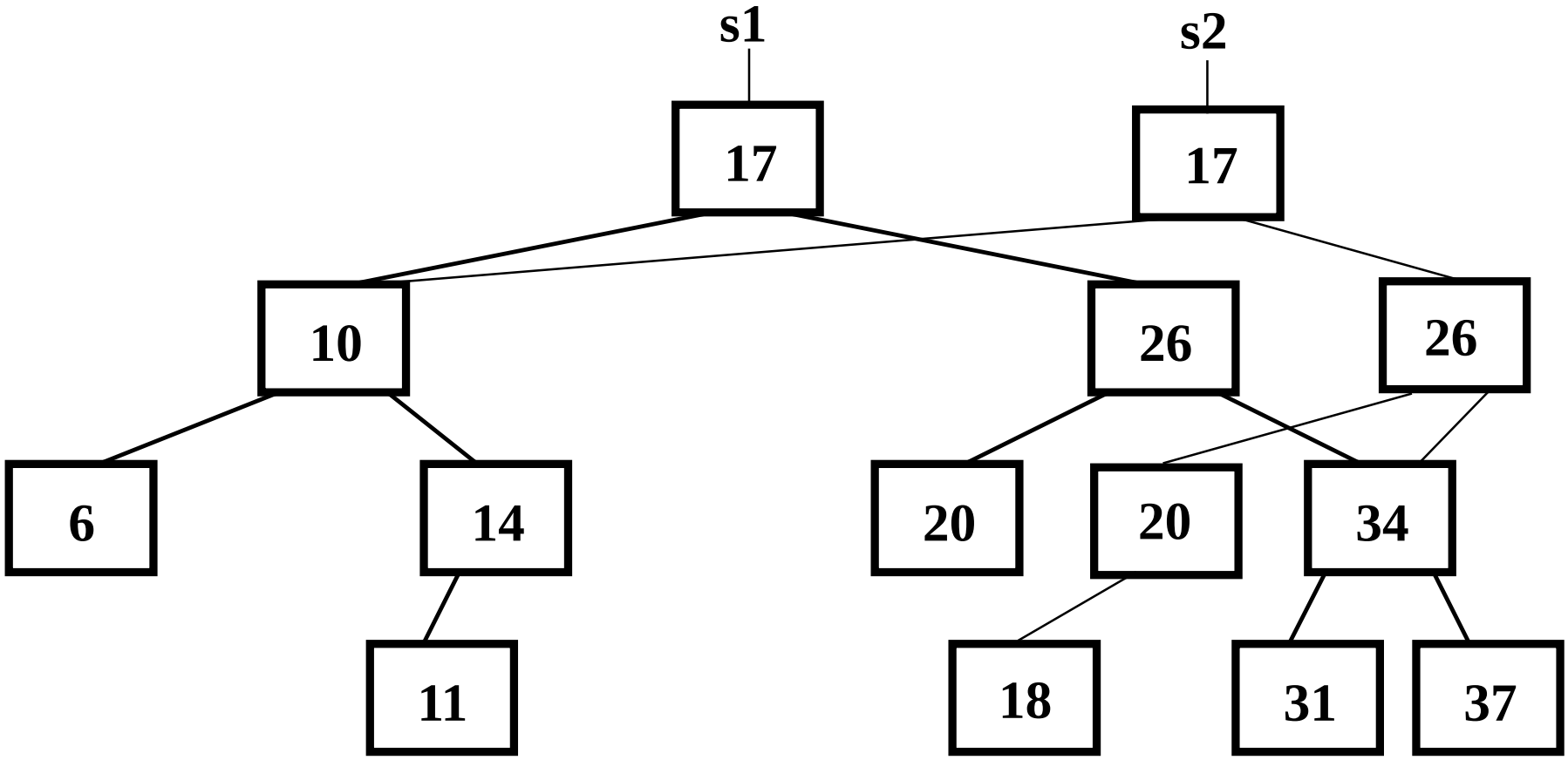
Słownik jako binarne drzewo poszukiwań (reprezentacja wewnętrzna)

Nawiązując do dyskusji dotyczącej dzielenia i kopiowania wartości z wykładu 6, można zadać sobie pytanie, jak wygląda reprezentacja wewnętrzna binarnego drzewa poszukiwań po dodaniu nowego elementu.

Języki funkcyjne (OCaml, SML, Haskell) wykorzystują tu kombinację współdzielenia i kopiowania (patrz następna strona). Kopiowane są wszystkie węzły znajdujące się na ścieżce poszukiwań, pozostałe węzły są współdzielone.

W naszej implementacji nie ma żadnych prób wyważania drzewa.

Słownik po wykonaniu operacji wstawiania
s2 = insert(s1,18,”eighteen”)



Dla uproszczenia rysunku pokazano tylko klucze elementów słownika. Kopiowana jest tylko część drzewa, położona na ścieżce wyszukiwania. Pozostała część jest współdzielona.

Zadania kontrolne

Algebraiczna specyfikacja kolejki nieskończonej

```
empty   : -> Queue
enqueue : Elem * Queue -> Queue
first   : Queue -> Elem
dequeue : Queue -> Queue
isEmpty : Queue -> bool

For all q:Queue, e1,e2: Elem
isEmpty (enqueue (e1,q))              = false
isEmpty (empty)                       = true
  dequeue (enqueue(e1,enqueue(e2,q))) =
                              enqueue(e1,dequeue(enqueue(e2,q)))
dequeue (enqueue(e1,empty))           = empty
dequeue (empty)                       = empty
first (enqueue(e1,enqueue(e2,q)))     = first(enqueue(e2,q))
first (enqueue(e1,empty))             = e1
first (empty)                         = ERROR
```

Zadania kontrolne

1. Dana jest następująca sygnatura dla kolejek niemodyfikowalnych (**czysto funkcyjnych**!; w pliku queue_fun_type.ml).

```
module type QUEUE_FUN =
sig
  (* This module implements queues (FIFOs)in a functional way. *)
  type 'a t
       (* The type of queues containing elements of type ['a]. *)
  exception Empty of string
       (* Raised when [first] is applied to an empty queue. *)
  val create: unit -> 'a t
       (* [create()] returns a new queue, initially empty. *)
  val enqueue: 'a * 'a t -> 'a t
       (* [enqueue (x,q)] adds the element [x] at the end of queue [q]. *)
  val dequeue: 'a t -> 'a t
       (* [dequeue q] removes the first element in queue [q] *)
  val first: 'a t -> 'a
       (* [first q] returns the first element in queue [q] without removing
           it from the queue, or raises [Empty] if the queue is empty.*)
  val isEmpty: 'a t -> bool
       (* [isEmpty q] returns [true] if queue [q] is empty,
           otherwise returns [false]. *)
end;;
```

Zadania kontrolne

a) Napisz strukturę, zgodna z powyższą sygnaturą, w której kolejka jest reprezentowana przez typ konkretny

```
type 'a t = EmptyQueue | Enqueue of 'a * 'a t
```

b) Napisz strukturę, zgodna z powyższą sygnaturą, w której kolejka jest reprezentowana przez listę.

c) Reprezentacja z punku a) i b) jest mało efektywna, ponieważ operacja wstawiania do kolejki ma złożoność liniową. W lepszej reprezentacji kolejka jest reprezentowana przez parę list. Para list ($[x_1; x_2; ...; x_m]$, $[y_1; y_2; ...; y_n]$) reprezentuje kolejkę $x_1\ x_2\ ...\ x_m\ y_n\ ...\ y_2\ y_1$. Pierwsza lista reprezentuje początek kolejki, a druga – koniec kolejki. Elementy w drugiej liście są zapamiętane w odwrotnej kolejności, żeby wstawianie było wykonywane w czasie stałym (na początek listy). *enqueue*(y, q) modyfikuje kolejkę następująco: (xl, $[y_1; y_2; ...; y_n]$) → (xl, $[y;y_1; y_2; ...; y_n]$). Elementy w pierwszej liście są pamiętane we właściwej kolejności, co umożliwia szybkie usuwanie pierwszego elementu.

dequeue(q) modyfikuje kolejkę następująco: ($[x_1; x_2; ...; x_m]$, yl) → ($[x_2; ...; x_m]$, yl). Kiedy pierwsza lista zostaje opróżniona, druga lista jest odwracana i wstawiana w miejsce pierwszej: ($[]$, $[y_1; y_2; ...; y_n]$) → ($[y_n; ...\ y_2; y_1]$, $[]$). Reprezentacja kolejki jest w postaci normalnej, jeśli **nie** wygląda tak: ($[]$, $[y_1; y_2; ...; y_n]$) dla n≥1. **Wszystkie operacje kolejki mają zwracać reprezentację w postaci normalnej**, dzięki czemu pobieranie wartości pierwszego elementu nie spowoduje odwracania listy. Odwracanie drugiej listy po opróżnieniu pierwszej też może się wydawać kosztowne. Jeśli jednak oszacujemy nie koszt pesymistyczny (oddzielnie dla każdej operacji kolejki), ale koszt zamortyzowany (uśredniony dla całego czasu istnienia kolejki), to okaże się, że koszt operacji wstawiania i usuwania z kolejki jest stały.

Napisz strukturę, zgodna z powyższą sygnaturą, w której kolejka jest reprezentowana w postaci pary list.

Zadania kontrolne

2. Dana jest następująca sygnatura dla kolejek modyfikowalnych (w pliku queue_mut.mli).

```
(* This module implements queues (FIFOs) with in-place modifications. *)
type 'a t  (* The type of queues containing elements of type ['a]. *)
exception Empty of string
      (* Raised when [first q] is applied to an empty queue [q]. *)
exception Full of string
      (* Raised when [enqueue (x,q)] is applied to a full queue [q]. *)
val create: int -> 'a t
      (* [create n] returns a new queue of length [n], initially empty. *)
val enqueue: 'a * 'a t -> unit
      (* [enqueue (x,q)] adds the element [x] at the end of a queue [q]. *)
val dequeue: 'a t -> unit
      (* [dequeue q] removes the first element in queue [q] *)
val first: 'a t -> 'a
      (* [first q] returns the first element in queue [q] without removing
         it from the queue, or raises [Empty] if the queue is empty.*)
val isEmpty: 'a t -> bool
      (* [isEmpty q] returns [true] if queue [q] is empty,
         otherwise returns [false]. *)
val isFull: 'a t -> bool
      (* [isFull q] returns [true] if queue [q] is full,
         otherwise returns [false]. *)
```

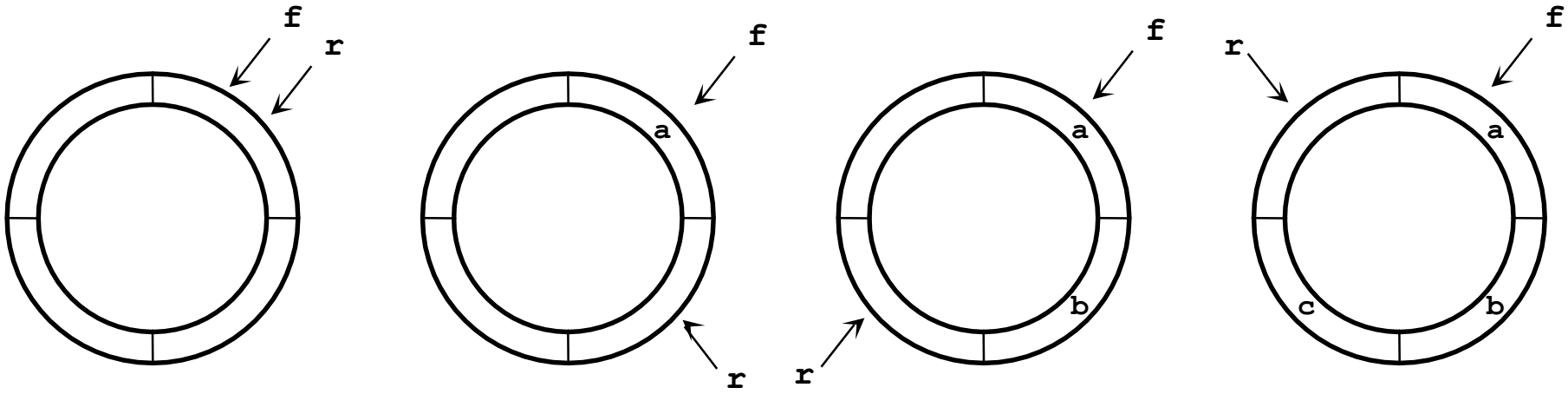
Zadania kontrolne

Napisz strukturę, zgodną z powyższą sygnaturą, w której kolejka jest reprezentowana przez tablicę cykliczną. **Wykorzystaj mechanizm oddzielnej kompilacji!** Nie zapomnij o testach! Program testowy powinien wypisać menu, zawierające wszystkie operacje kolejki i wykonać wybraną operację.

Kolejka reprezentowana przez tablicę cykliczną

kolejka pusta

kolejka pełna



Zadania kontrolne

3. Napisz program pokazujący, że operacja `insert` dla binarnego drzewa poszukiwań zwraca drzewo o strukturze pokazanej na stronie 45.
4. Napisz funktor dla słownika (czysto funkcyjnego) reprezentowanego przez listę skończoną, zgodny z sygnaturą podaną na wykładzie. Elementy w liście mają być uporządkowane niemalejąco według kluczy. Funkcjonalność słownika ma być dokładnie taka sama, jak słownika z wykładu. Wykorzystując ten funktor utwórz dwa konkretne słowniki z różnymi typami kluczy i przetestuj je.