Wykład 2

Podstawy programowania funkcyjnego cd.

Typy bazowe w języku OCaml i Scala
Reguły wartościowania jako reguły dedukcji – OCaml
Środowisko i domknięcie
Wyrażenia funkcyjne i definicje funkcji
Polimorfizm parametryczny
Funkcjonały; postać zwinięta i rozwinięta funkcji
Organizacja pamięci programu
Rekursja ogonowa
Wartościowanie gorliwe
Dopasowanie do wzorca w definicjach zmiennych
Wzorzec z wieloznacznikiem
Wyrażenie „match”
Współdzielenie danych; wzorce warstwowe
Dodatek: Arytmetyka liczb całkowitych i zmiennopozycyjnych w języku Java

Typy bazowe w języku OCaml

typ `unit` – istnieje tylko jedna wartość tego typu
`():unit`

typ `bool`
`false true not && or` lub `||`
`= <> == != < > <= >=`

Operator równości strukturalnej (`=`) porównuje strukturę operandów, podczas gdy operator równości fizycznej (`==`) sprawdza, czy operandy zajmują w pamięci to samo miejsce. Operator `==` jest przydatny w programowaniu imperatywnym. Operator `<>` jest negacją operatora `=`, a `!=` jest negacją `==`.

typ `char`

typ `string` Patrz moduł `String`
`^` konkatenacja napisów

typ `int` $[-2^{30},2^{30}-1] = [-1073741824,1073741823]$
`+ - * / mod`

typ `float` (liczby zmiennoprzecinkowe podwójnej precyzji)
`+. -. *. /. **`
`ceil floor sqrt exp log log10 sin cos tan acos asin atan`

Funkcje koercji:
val float_of_int : int -> float
val int_of_float : float -> int (obcina część ułamkową = truncate)

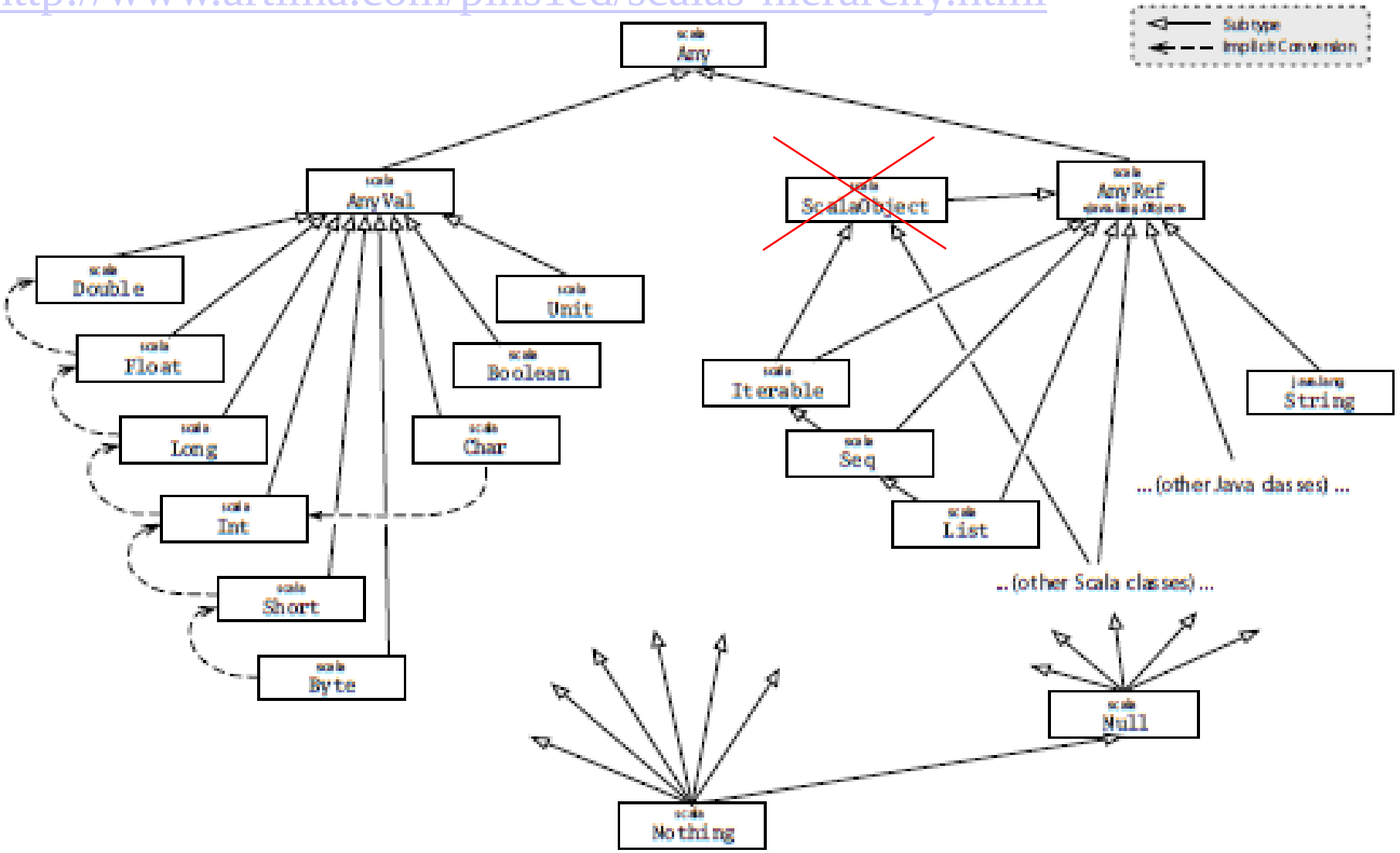
Typy bazowe w języku Scala

Typ	**Zakres wartości**
Byte	8-bitowe liczby całkowite ze znakiem [-128,127]
Short	16-bitowe liczby całkowite ze znakiem [-32 768, 32 767]
Int	32-bitowe liczby całkowite ze znakiem [-2 147 483 648, 2 147 483 647]
Long	64-bitowe liczby całkowite ze znakiem [-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]
Float	32-bitowe liczby zmiennopozycyjne ze znakiem [-3.4028235e+38, -1.40e-45] ∪[1.40e-45, 3.4028235e+38] oraz -0.0f i +0.0f
Double	64-bitowe liczby zmiennopozycyjne ze znakiem -0.0 i +0.0 oraz [-1.7976931348623157e+308, -4.9e-324] ∪ [4.9e-324, 1.7976931348623157e+308]
Char	16-bitowe znaki w kodzie Unicode, umożliwia zakodowanie 2^{16} = 65 536 znaków
String	(skończona) sekwencja znaków (typu Char)
Boolean	wartości logiczne; literały true i false
Unit	istnieje tylko jedna wartość tego typu: ()
AnyVal	nadtyp każdego typu wartości
Null	podtyp każdego typu referencyjnego, pusta referencja; literał null
AnyRef	nadtyp każdego typu referencyjnego
Nothing	podtyp każdego typu; nie ma żadnych wartości
Any	nadtyp każdego typu; każdy obiekt jest typu Any

W Scali typy proste i klasy opakowujące nie są rozróżniane.

Hierarchia klas w języku Scala

http://www.artima.com/pins1ed/scalas-hierarchy.html



Wartościowanie gorliwe w języku OCaml

Wartość jest wyrażeniem, wartościowanym do siebie samego.

- W celu obliczenia wartości krotki, oblicz jej składniki (od prawej do lewej).
- W celu obliczenia wartości wyrażenia **`if`** b **`then`** e_1 **`else`** e_2 oblicz wartość b. Jeśli otrzymasz wartość `true`, to wartością całego wyrażenia jest wartość wyrażenia e_1. Jeśli otrzymasz wartość `false`, to wartością całego wyrażenia jest wartość wyrażenia e_2.
- Wyrażenie funkcyjne (**`function`** x $\rightarrow$ e) jest wartością.
- W aplikacji e_1 e_2 wartościowany jest argument e_2 (otrzymujemy wartość v), a potem wyrażenie e_1 (otrzymujemy wartość **`function`** x $\rightarrow$ e). Ewaluacja następuje po poprawnej kompilacji (w szczególności została już sprawdzona zgodność typów, stąd wiadomo, że e_1 musi być funkcją). Wynik aplikacji jest wartością wyrażenia $e[x \leftarrow v]$.
- W celu obliczenia wartości wyrażenia **`let`** x=e_1 **`in`** e_2 wartościujemy e_1, co daje wartość v_1, a następnie wartościujemy wyrażenie $e_2[x \leftarrow v_1]$.

Uwaga. W języku OCaml kolejność wartościowania argumentów nie jest wyspecyfikowana. Jak dotąd wszystkie implementacje wartościowały argumenty od prawej do lewej, ale to może się zmienić (np. implementacje języka SML wartościują argumenty od lewej do prawej). To jest istotne *tylko* przy wykorzystywaniu efektów ubocznych.

Reguły wartościowania jako reguły dedukcji – OCaml

Niech relacja e $\Rightarrow$ v oznacza „wyrażenie e ma wartość v”.

$$\frac{przesłanki}{wniosek}\ (Nazwa)$$

$$\frac{\vdash e_n \Rightarrow v_n \quad \ldots \quad \vdash e_1 \Rightarrow v_1}{\vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)}\ (Krotka)$$

$$\frac{\vdash e_1 \Rightarrow \mathrm{true} \quad \vdash e_2 \Rightarrow v}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}\ (War1) \qquad \frac{\vdash e_1 \Rightarrow \mathrm{false} \quad \vdash e_3 \Rightarrow v}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}\ (War2)$$

$$\frac{}{\vdash (\texttt{function}\ x \rightarrow e) \Rightarrow (\texttt{function}\ x \rightarrow e)}\ (Fun1)$$

$$\frac{\vdash e_2 \Rightarrow v_2 \quad \vdash e_1 \Rightarrow (\texttt{function}\ x \rightarrow e) \quad \vdash e[x \leftarrow v_2] \Rightarrow v}{\vdash e_1\ e_2 \Rightarrow v}\ (Ap1)$$

$$\frac{\vdash e_1 \Rightarrow v_1 \quad \vdash e_2[x \leftarrow v_1] \Rightarrow v}{\vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow v}\ (Let)$$

Środowisko i domknięcie

Wyrażenia w języku OCaml (i w innych językach programowania) są wartościowane w *środowisku* (ang. environment) składającym się z listy par <identyfikator, wartość>. Z abstrakcyjnego punku widzenia środowisko jest *słownikiem*.

Wartością wyrażenia funkcyjnego jest *domknięcie* (ang. closure), które jest parą <środowisko, wyrażenie funkcyjne>.

Nazwa pochodzi od „domykania” literału funkcyjnego względem wszystkich zmiennych wolnych tego literału.

Powyżej prosta semantyka języka była oparta na zastępowaniu i przepisywaniu wyrażeń. Implementacja, wykorzystująca bezpośrednio ten model byłaby bardzo nieefektywna. W praktyce wartościowanie wykorzystuje środowiska i domknięcia.

Literały funkcyjne (funkcje anonimowe)

Scala

```
((x:Int) => x+x) (6)
res0: Int = 12
```

OCaml

```
# (function x -> x+x) 6;;
- : int = 12
```

W szczególności wyrażenie funkcyjne można związać z identyfikatorem.

```
val double = (x:Int) => x+x
double: Int => Int = <function1>
double(6)
res1: Int = 12
```

```
# let double = function x -> x+x;;
val double : int -> int = <fun>
# double 6;;
- : int = 12
```

Poniższy sposób definiowania funkcji jest wygodnym skrótem notacyjnym.

```
def double(x:Int) = x+x
double: (x: Int)Int  // w Scali to jest metoda
double(6)
res2: Int = 12
```

```
# let double  x = x+x;;
val double : int -> int = <fun>
# double 6;;
- : int = 12
```

Funkcjonały

Wynik funkcji może być funkcją.

Scala

```
val f = (y:Int) => (x:Int) => x*x+y

f: Int => (Int => Int) = <function1>


f(2)(5)   // Aplikacja wiąże w lewo  (f (2)) 5

res3: Int = 27

val f2 = f(2)

f2: Int => Int = <function1>
// f2 jest funkcją wyspecjalizowaną

f2 (5)

res4: Int = 27

val f5 = (z:Int) => f(z)(5)

f5: Int => Int = <function1>
// f5 jest funkcją wyspecjalizowaną

f5(2)

res5: Int = 27
```

OCaml

```
# let f = function y -> function x -> x*x+y;;

val f : int -> int -> int = <fun>
(* Strzałka  wiąże w prawo int -> (int -> int)  *)

# f 2 5;;    (* Aplikacja wiąże w lewo  (f 2) 5    *)

- : int = 27

# let f2 = f  2;;

val f2 : int -> int = <fun>
(* f2 jest funkcją wyspecjalizowaną  *)

# f2 5;;

- : int = 27

# let f5 = function z -> f z 5;;

val f5 : int -> int = <fun>
(* f5 jest funkcją wyspecjalizowaną *)

# f5 2;;

- : int = 27
```

Lukier syntaktyczny dla funkcji w języku OCaml

fun *arg1 arg2 ... argn -> wyr* jest skrótem dla

function *arg1* -> **function** *arg2* -> ... -> **function** *argn* -> *wyr*

let *identyfikator* = **function** *arg -> wyr*;; można skrócić do

let *identyfikator arg = wyr*;;

```
# let plus = function x -> function y -> x + y;;
val plus : int -> int -> int = <fun>
# let plus' x = function y -> x + y;;
val plus' : int -> int -> int = <fun>
# let plus'' x y = x + y;;
val plus'' : int -> int -> int = <fun>
# let plus3 = fun x y -> x + y;;
val plus3 : int -> int -> int = <fun>
```

Te cztery definicje funkcji są równoważne

```
# let d1=plus 2 3   let d2=plus' 2 3   let d3=plus'' 2 3   let d4=plus3 2 3;;
val d1 : int = 5      val d2 : int = 5      val d3 : int = 5      val d4 : int = 5
```

Polimorfizm parametryczny

Scala

```
def id[A](x:A) = x

id: [A](x: A)A

id(5)

res6: Int = 5

id (3+4, "siedem")

res7: (Int, String) = (7,siedem)

id[String=>String] (id) ("OK")

res5: String = OK

id _

res8: Nothing => Nothing = <function1>
```

OCaml

```
# let id x = x;;

val id : 'a -> 'a = <fun>

# id 5;;

- : int = 5

# id (3+4, "siedem");;

- : int * string = (7, "siedem")

# id id "OK";;

- : string = "OK"

# id;;

- : 'a -> 'a = <fun>
```

Funkcje w Scali są zawsze monomorficzne,
więc typ został skonkretyzowany.
To będzie wyjaśniowe później.

Postać zwinięta (ang. uncurried) i rozwinięta (ang. curried) funkcji

W językach funkcyjnych wszystkie funkcje są jednoargumentowe!

Scala

```
// postać zwinięta
def plus(x:Int, y:Int) = x+y
plus: (x: Int, y: Int)Int
plus (4,5)
res9: Int = 9

// postać rozwinięta
def add (x:Int)(y:Int) = x+y
add: (x: Int)(y: Int)Int
add(4)(5)
res10: Int = 9
```

OCaml

```
(* postać zwinięta *)
# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# plus (4,5);;
-     : int = 9

(* postać rozwinięta *)
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 4 5;;
- : int = 9
```

Funkcje wzajemnie rekurencyjne

Scala

```
def evenR(n: Int): Boolean =
    if (n==0) true else oddR(n-1)

def oddR(n: Int) :Boolean =
    if (n==0) false else evenR(n-1)

evenR: (n: Int)Boolean

oddR: (n: Int)Boolean

evenR(128)

res11: Boolean = true

oddR(128)

res12: Boolean = false
```

OCaml

```
# let rec evenR n = if n=0 then true else oddR(n-1)
      and oddR n =if n=0 then false else evenR(n-1);;


val evenR : int -> bool = <fun>

val oddR : int -> bool = <fun>

# evenR 128;;

- : bool = true

# oddR 128;;

: bool = false
```

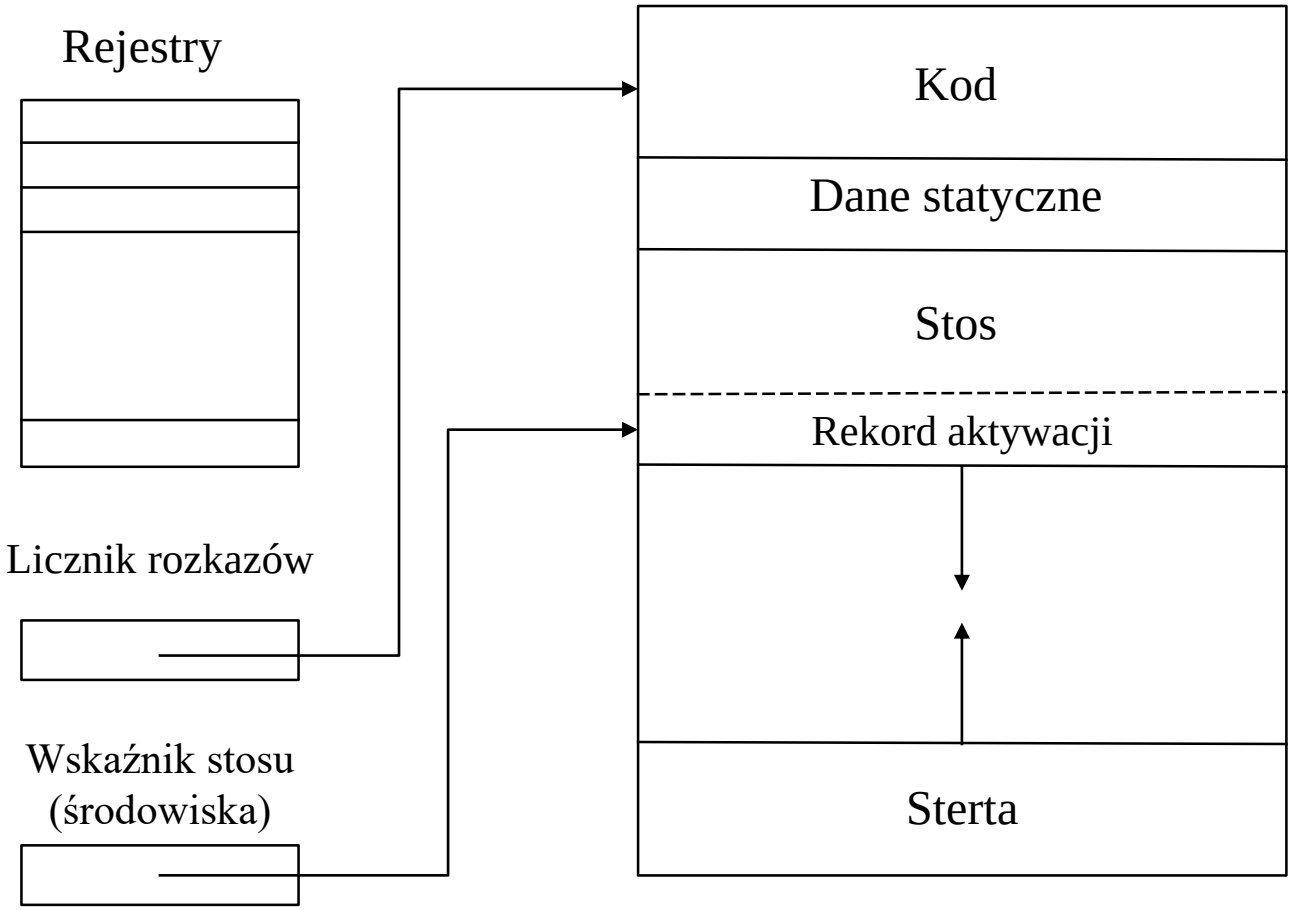
Oczywiście efektywne definicje funkcji even i odd powinny wyglądać tak:

```
def even(n: Int) = n % 2 == 0
even: (n: Int)Boolean

def odd(n: Int) = n % 2 != 0
odd: (n: Int)Boolean
```

```
# let even n = n mod 2 = 0;;
val even : int -> bool = <fun>

# let odd n = n mod 2 <> 0;;
val odd : int -> bool = <fun>
```

Organizacja pamięci programu

W czasie wykonania w obszarze pamięci programu przechowywany jest jego *kod* (ang. code) i *dane* (ang. data). Obszar danych jest podzielony na *dane statyczne* (ang. static data), *stos* (ang. stack) i *stertę* (ang. heap) z różną strategią rezerwacji pamięci.



Organizacja pamięci programu

- Obszar danych statycznych zawiera wartości zmiennych statycznych, dla których pamięć jest przydzielana w trakcie kompilacji (np. w C i C++ za pomocą słowa kluczowego static wewnątrz funkcji). Uwaga: modyfikator static w językach Java, C#, C++ w kontekście programowania obiektowego ma inną semantykę!
- Stos (pamięć automatyczna) składa się z *rekordów aktywacji* (ang. activation records), tworzonych automatycznie dla każdego bloku (funkcji) przy wejściu i zdejmowanych ze stosu przy wyjściu z bloku. Rekord aktywacji, nazywany też *ramką stosu* (ang. stack frame) zawiera m.in. zmienne lokalne, łącze dostępu, argumenty wywołania funkcji, wartość zwracaną (lub jej adres), adres powrotu (adres kolejnej instrukcji w kodzie programu po zakończeniu wykonywania bieżącej funkcji).
- Sterta (pamięć wolna, pamięć dynamiczna) zawiera dane, które mają istnieć dłużej niż wykonanie bieżącego bloku. Pamięć dla nich jest przydzielana dynamicznie np. za pomocą operatora new.

Wartością wyrażenia funkcyjnego jest *domknięcie* (ang. closure), które jest parą <środowisko, wyrażenie funkcyjne>. Domknięcie jest implementowane jako para składająca się ze wskaźnika do rekordu aktywacji i wskaźnika do kodu funkcji. Wskaźnik do rekordu aktywacji wystarczy do reprezentowania środowiska, ponieważ rekord aktywacji dla funkcji zawiera *łącze dostępu*, nazywane *też łączem statycznym* (ang. access link, static link), wskazujące na rekord aktywacji najbliższego bloku, w którym funkcja jest zdefiniowana.

Rekursja ogonowa - OCaml

Rekursja ogonowa lub terminalna (ang. tail recursion, terminal recursion) ma miejsce wtedy, kiedy w każdej klauzuli definiującej funkcję wartość funkcji jest obliczana bez wywołania rekurencyjnego lub jest *bezpośrednim* wynikiem tego wywołania.

```
# let rec suc n = if n=0 then 1 else 1 + suc(n-1);;
val suc : int -> int = <fun>
# suc 1000000;;
Stack overflow during evaluation (looping recursion?).

# let succ_tail n =
  let rec succ_iter(n,accum) =
      if n=0 then accum else succ_iter(n-1,accum+1)
  in succ_iter(n,1)
;;
val succ_tail : int -> int = <fun>
# succ_tail 1000000;;
- : int = 1000001
```

Gorliwe wartościowanie rekursji ogonowej - OCaml

```
let rec factIt (n, ak) =  if n=0 then ak else factIt(n-1, n*ak);;
```

factIt(4,1) ⇒ factIt(4-1,4*1)
⇒ factIt(3,4)
⇒ factIt(3-1,3*4)
⇒ factIt(2,12)
⇒ factIt(2-1,2*12)
⇒ factIt(1,24)
⇒ factIt(1-1,1*24)
⇒ factIt(0,24)
⇒ 24

Analogicznie przebiega obliczenie w języku Scala.

Dobre kompilatory wykrywają rekursję ogonową i wykonują ją efektywnie (iteracyjnie, bez tworzenia nowych rekordów aktywacji na stosie). Rezultat wywołania rekurencyjnego `factIt(n-1,n*ak)` nie podlega dalszym obliczeniom, lecz jest zwracany jako wartość wywołania `factIt(n,ak)`.

Dodatkowy parametr `ak` nazywany jest akumulatorem, ponieważ służy do „akumulowania” wyniku.

Gorliwe wartościowanie rekursji - OCaml

```
let  rec factorial n =  if n=0  then 1 else n*factorial (n-1);;
```

factorial 4 $\Rightarrow$ if 4=0 then 1 else 4*factorial (4-1)

$\Rightarrow$ 4*factorial 3

$\Rightarrow$ 4*(if 3=0 then 1 else 3*factorial (3-1))

$\Rightarrow$ 4*(3* factorial 2)

$\Rightarrow$ 4*(3*(if 2=0 then 1 else 2*factorial (2-1)))

$\Rightarrow$ 4*(3*(2*factorial 1))

$\Rightarrow$ 4*(3*(2*(if 1=0 then 1 else 1*factorial (1-1))))

$\Rightarrow$ 4*(3*(2*(1*factorial 0)))

$\Rightarrow$ 4*(3*(2*(1*(if 0=0 then 1 else 0*factorial (0-1)))))

$\Rightarrow$ 4*(3*(2*(1*1)))

$\Rightarrow$ 4*(3*(2*1))

$\Rightarrow$ 4*(3*2)

$\Rightarrow$ 4*6

$\Rightarrow$ 24

Analogicznie przebiega obliczenie w języku Scala.

Rekursja ogonowa - Scala

```
def succ(n: Int): Int = if (n==0) 1 else 1 + succ(n-1)
succ: (n: Int)Int

succ(1000000)
java.lang.StackOverflowError
at .succ(<console>:7)
at .succ(<console>:7)
at  …
def succTail(n: Int) = {
   def succIter(n:Int, accum:Int): Int =
      if (n==0) accum else succIter(n-1, accum+1)
   succIter(n,1)
}
succTail: (n: Int)Int

succTail(1000000)
res13: Int = 1000001
```

Ograniczenia rekursji ogonowej w języku Scala

W języku Scala rekursja ogonowa jest optymalizowana jeśli funkcja wywołuje ogonowo samą siebie. W innych przypadkach taka optymalizacja nie jest przeprowadzana, ponieważ na maszynie wirtualnej Javy (JVM) jest to bardzo trudne.

W definicji funkcji evenR i oddR mamy do czynienia z rekursją ogonową.

```
def evenR(n: Int): Boolean =
        if (n==0) true else oddR(n-1)
def oddR(n: Int) :Boolean =
        if (n==0) false else evenR(n-1)
evenR: (n: Int)Boolean
oddR: (n: Int)Boolean
```

Niestety, Scala nie przeprowadza tu oczekiwanej optymalizacji:

```
evenR(1000000)
java.lang.StackOverflowError
at .oddR(<console>:10)
at .evenR(<console>:8)
at .oddR(<console>:10)
at   …
```

W języku OCaml (i wszystkich językach funkcyjnych) rekursja ogonowa jest zawsze optymalizowana:

```
# evenR 1000000;;
- : bool = true
```

Adnotacja @tailrec w języku Scala

W celu upewnienia się, że kompilator Scali zoptymalizował rekursję ogonową, warto przed odpowiednią metodą umieścić adnotację @tailrec z pakietu scala.annotation. Jeśli optymalizacja nie jest przeprowadzona to błąd kompilacji zawiera wyjaśnienia.

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> :paste
// Entering paste mode (ctrl-D to finish)
def succTail(n: Int) = {
  @tailrec
  def succIter(n:Int, accum:Int): Int =
    if (n==0) accum else succIter(n-1, accum+1)
  succIter(n,1)
}
// Exiting paste mode, now interpreting.
succTail: (n: Int)Int

scala> :paste
// Entering paste mode (ctrl-D to finish)
@tailrec
def succ(n: Int): Int = if (n==0) 1 else 1 + succ(n-1)
// Exiting paste mode, now interpreting.

<pastie>:13: error: could not optimize @tailrec annotated method succ: it contains a recursive call not in tail position
       def succ(n: Int): Int = if (n==0) 1 else 1 + succ(n-1)
                                                   ^
```


scala> :paste

Wzorce - definicja

Wzorce pozwalają na dekompozycję wartości strukturalnych i związanie identyfikatorów z wartościami ich komponentów.

Wzorzec (ang. pattern) w języku OCaml jest zbudowany ze zmiennych i stałych za pomocą konstruktorów wartości. **Zmienne we wzorcach nie są związane z żadnymi wartościami - związanie nastąpi w wyniku dopasowania do wzorca.** Dopasowanie do wzorca (ang. pattern matching) ma sens tylko dla wartości, nie będących funkcjami.

- Każda wartość dopasowuje się do zmiennej, tworząc związanie.
- Każda wartość dopasowuje się do wieloznacznika (bez tworzenia związania).
- Konstruktory wartości dopasowują się tylko do identycznych konstruktorów.
- Do wzorca, posiadającego strukturę, dopasowuje się tylko wartość o takiej samej strukturze.

Dopasowanie do wzorca w definicjach zmiennych

OCaml

```
# let x = (false, 10);;
val x : bool * int = (false, 10)

# let (z,y) = x;;
val z : bool = false
val y : int = 10
```

Scala

```
val x = (false, 10)
x: (Boolean, Int) = (false,10)

val (z,y) = x
z: Boolean = false
y: Int = 10
```

Dopasowanie do wzorca w definicjach zmiennych

OCaml

```
# let (false, y) = x;;
Characters 4-14:
  let (false, y) = x;;
      ^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(true, _)
val y : int = 10
```

Scala

```
val (false,y) = x
y: Int = 10
```

Dopasowanie do wzorca w definicjach zmiennych

OCaml

```
# let (true, y) = x;;
Characters 4-13:
  let (true, y) = x;;
      ^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(false, _)
Exception: Match_failure ("//toplevel//", 1, 4).
```

Scala

```
val (true,y) = x
scala.MatchError: (false,10) (of class scala.Tuple2)
 ... 53 elided
```

Dopasowanie do wzorca w definicjach zmiennych

OCaml

```
# let xs = ["Ala"; "ma"; "kota"];;
val xs : string list = ["Ala"; "ma"; "kota"]

# let [x1; x2; x3] = xs;;
Characters 4-16:
  let [x1; x2; x3] = xs;;
      ^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x1 : string = "Ala"
val x2 : string = "ma"
val x3 : string = "kota"
```

Scala

```
val xs = List("Ala", "ma", "kota")
xs: List[String] = List(Ala, ma, kota)

val List(x1, x2, x3) = xs
x1: String = Ala
x2: String = ma
x3: String = kota
```

Dopasowanie do wzorca w definicjach zmiennych

OCaml

```
# let h::t = xs;;
Characters 4-8:
  let h::t = xs;;
      ^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val h : string = "Ala"
val t : string list = ["ma"; "kota"]
```

Scala

```
val h::t = xs
h: String = Ala
t: List[String] = List(ma, kota)
```

Wzorzec z wieloznacznikiem (z dżokerem, uniwersalny)

Wieloznacznik (lub *dżoker*, ang. wildcard) podobnie jak zmienna dopasowuje się z każdą wartością, ale nie tworzy związania.

OCaml

```
# let (z, _ ) = (false, 10);;
val z : bool = false
```

Scala

```
val (z,_) = (false, 10)
z: Boolean = false
```

Wzorce nie muszą być „płaskie”

OCaml

```
# let x = (("Smith", 25), true);;
val x : (string * int) * bool = (("Smith", 25), true)

# let ((n,w),b) = x;;
val n : string = "Smith"
val w : int = 25
val b : bool = true
```

Scala

```
val x = (("Smith", 25), true)
x: ((String, Int), Boolean) = ((Smith,25),true)

val ((n,w),b) = x
n: String = Smith
w: Int = 25
b: Boolean = true
```

Parametry funkcji są wzorcami

OCaml

```
# let f = fun ((x,y),z) -> (x,y,(x,y),z);;
val f : ('a * 'b) * 'c -> 'a * 'b * ('a * 'b) * 'c = <fun>
# f((2, 5.5), [1;2]);;
- : int * float * (int * float) * int list = (2, 5.5, (2, 5.5), [1; 2])
```

Scala

Niestety, w języku Scala parametry muszą być zmiennymi.

```
def f[A,B,C]((x:A,y:B),z:C) = (x,y,(x,y),z)
<console>:1: error: identifier expected but '(' found.
      def f[A,B,C]((x:A,y:B),z:C) = (x,y,(x,y),z)
```

Możemy jednak wykorzystać dopasowanie do wzorca w treści funkcji (w definicjach zmienych):

```
def f[A,B,C](k:((A,B),C)) =  {
   val ((x,y),z) = k
   (x,y,(x,y),z)
}
f: [A, B, C](k: ((A, B), C))(A, B, (A, B), C)
f((2, 5.5), List(1,2))
res0: (Int, Double, (Int, Double), List[Int]) = (2,5.5,(2,5.5),List(1, 2))
```

Wyrażenie match

Scala

```
def imply1(pb:(Boolean,Boolean)) =
   pb match {
         case (false,false) => true
         case (false,true)  => true
         case (true,false)  => false
         case (true,true)   => true
}

imply1: (pb: (Boolean, Boolean))Boolean

def imply2(pb:(Boolean,Boolean)) =
  pb match {
        case (true,false) => false
        case       _      => true
}
imply2: (pb: (Boolean, Boolean))Boolean

imply2(1>2,true)
res1: Boolean = true
```

OCaml

```
# let imply1 pb =
  match pb with
      (false, false) -> true
    | (false, true)  -> true
    | (true,  false) -> false
    | (true,  true)  -> true
;;
val imply1 : bool * bool -> bool = <fun>

# let imply2 pb =
  match pb with
      (true,  false) -> false
    |         _      -> true
;;
val imply2 : bool * bool -> bool = <fun>

# imply2(1>2,true);;
- : bool = true
```

Przykład: funkcja „zip”

OCaml

```
# let rec zip (xs,ys)  =              (* zip ([x1, ... ,xn], [y1, ... ,yn]) =  [(x1,y1), ... ,(xn,yn)]  *)
     match (xs,ys) with
          (h1::t1,h2::t2) -> (h1,h2)::zip(t1,t2)
        |          _          -> [];;
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>
# zip ([1;2;3], ['a';'b';'c']);;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
# zip ([1;2;3], ['a';'b']);;
- : (int * char) list = [(1, 'a'); (2, 'b')]
```

Scala

```
def zip[A,B](xs:List[A],ys:List[B]): List[(A,B)]
    (xs,ys) match {
       case (h1::t1, h2::t2) => (h1,h2)::zip(t1,t2)
       case          _           => Nil
}
zip: [A, B](xs: List[A], ys: List[B])List[(A, B)]
zip (List(1,2,3), List('a','b','c'))
res2: List[(Int, Char)] = List((1,a), (2,b), (3,c))
zip (List(1,2,3), List('a','b'))
res3: List[(Int, Char)] = List((1,a), (2,b))
```

Przykład: funkcja „unzip”

OCaml

```
# let rec unzip ps =       (* unzip [(x1,y1); ... ;(xn,yn)]  = ([x1; ... ;xn], [y1; ... ;yn]) *)
    match ps with
      []            -> ([], [])
    | (h1,h2)::t  -> let (l1,l2) = unzip t in (h1::l1, h2::l2);;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

# unzip [(1,2);(3,4);(5,6);(7,8)];;
- : int list * int list = ([1; 3; 5; 7], [2; 4; 6; 8])
```

Scala

```
def unzip[A,B](ps:List[(A,B)]): (List[A],List[B]) =
  ps match {
        case Nil          => (Nil, Nil)
        case (h1,h2)::t => {val (xs1,xs2)=unzip(t); (h1::xs1, h2::xs2)}
}
unzip: [A, B](ps: List[(A, B)])(List[A], List[B])

unzip(List((1,2),(3,4),(5,6),(7,8)))
res4: (List[Int], List[Int]) = (List(1, 3, 5, 7),List(2, 4, 6, 8))
```

Kombinacja wzorców

w1 | ... | *wn* *wi muszą zawierać te same zmienne tych samych typów!*

```
# let isLatinVowel v =
   match v with
       'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
   |                  _                  -> false;;
val isLatinVowel : char -> bool = <fun>
# isLatinVowel 's';;
- : bool = false
# isLatinVowel 'i';;
- : bool = true
```

Wzorce dozorowane - OCaml

```
# let srednia p =
   match p with
     (x,x) -> x                          (* zmienna x występuje we wzorcu dwukrotnie *)
   | (x,y) -> (x +. y) /. 2.0;;
 Characters 40-41:
    (x,x) -> x                          (* zmienna x występuje we wzorcu dwukrotnie *)
         ^
Error: Variable x is bound several times in this matching
(* Można jednak użyć wzorców dozorowanych (ang. guarded matches) *)
# let srednia p =
  match p with
      (x,y) when x=y -> x                (* guarded match *)
    | (x,y)                -> (x+.y)/.2.0;;
      val srednia : float * float -> float = <fun>
# srednia (5.,5.);;
-    : float = 5.
-    (* W tym przykładzie najlepiej tak:  *)
let srednia(x,y) = if x=y then x else (x+.y)/.2.0;;
```

Wzorce dozorowane - Scala

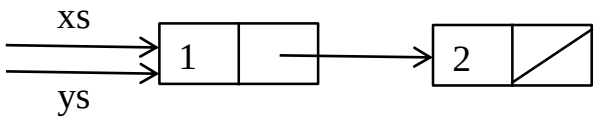
```
def średnia(p:(Double,Double)) =
  p match {
  case (x,x) => x                  // zmienna x występuje we wzorcu dwukrotnie
  case (x,y) => (x+y)/2.0
}
<console>:12: error: x is already defined as value x
        case (x,x) => x           // zmienna x występuje we wzorcu dwukrotnie
//  Można jednak użyć wzorców dozorowanych (ang. guarded matches)
def średnia(p:(Double,Double)) =
  p match {
  case (x,y) if x == y => x          // pattern guard
  case (x,y)          => (x+y)/2.0
}   średnia: (p: (Double, Double))Double
średnia (5.0, 5.0)
res0: Double = 5.0
def średnia(p:(Double,Double)) = {   // można też tak
  val (x,y) = p
  if (x==y) x else (x+y)/2.0
}
```

Współdzielenie danych niemodyfikowalnych

W językach funkcyjnych współdzielone są wszystkie dane, które można bezpiecznie współdzielić przy założeniu, że są one niemodyfikowalne, np.

```
# let xs = [1;2];;
val xs : int list = [1; 2]
# let ys = xs;;
val ys : int list = [1; 2]
```

Jak wygląda reprezentacja wewnętrzna tych list?



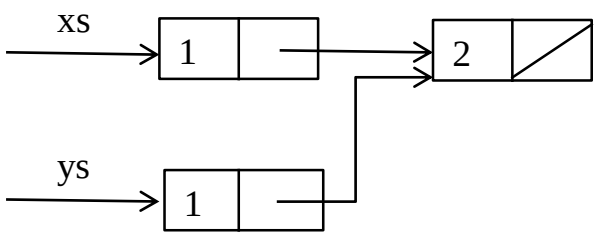
Na wykładzie 6 dowiemy się, jak można to sprawdzić.

Współdzielenie i kopiowanie danych niemodyfikowalnych

W języku OCaml **konstruktor wartości z argumentem zawsze tworzy nową wartość**, np.

```
# let ys = let h::t = xs in h::t;;
Characters 13-17:
  let ys = let h::t = xs in h::t;;
               ^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val ys : int list = [1; 2]
```

Jak teraz wygląda reprezentacja wewnętrzna tych list?

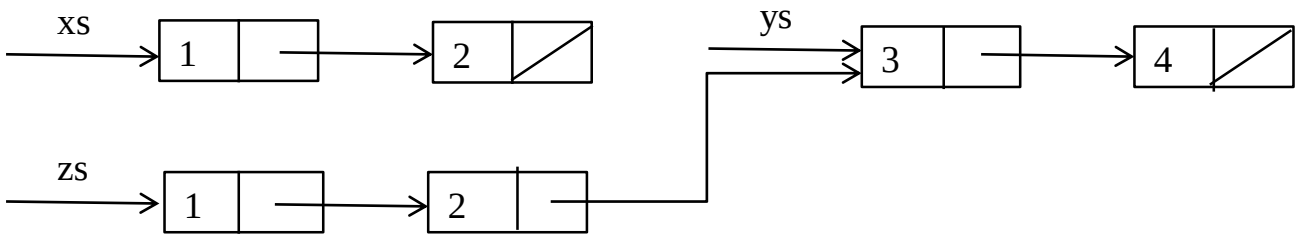


Współdzielenie i kopiowanie danych niemodyfikowalnych

W języku OCaml funkcja konkatenacji list jest zdefiniowana następująco:

```
let rec ( @ ) l1 l2 =   (* identyfikator symboliczny, można używać infiksowo  (patrz wykład 3) *)
   match l1 with
     [] -> l2
   | hd :: tl -> hd :: (tl @ l2);;

# let xs = [1;2] and ys = [3;4]    let zs = xs @ ys;;
val xs : int list = [1; 2]
val ys : int list = [3; 4]
val zs : int list = [1; 2; 3; 4]
```

Jak wygląda reprezentacja wewnętrzna tych list?



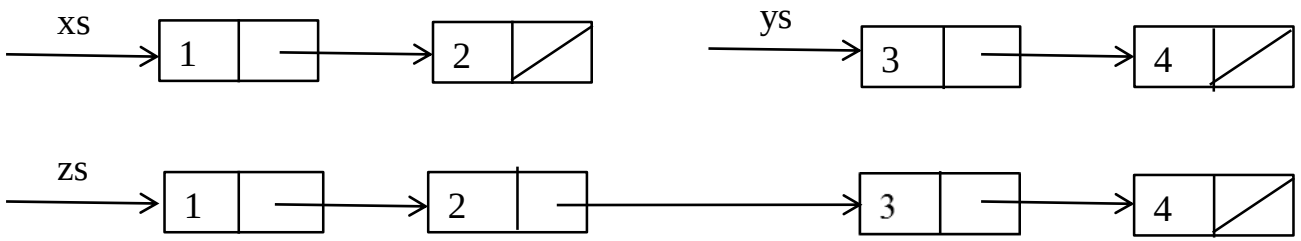
Kopiowane jest tylko to, co konieczne; współdzielone jest to, co możliwe (przy założeniu, że dane są niemodyfikowalne). Złożoność obliczeniowa (czasowa i pamięciowa) jest liniowa względem długości pierwszej listy.

Współdzielenie i kopiowanie danych niemodyfikowalnych

Funkcję konkatenacji można też zdefiniować za pomocą rekursji po obu listach:

```
let rec badAppend l1 l2 =
        match (l1, l2) with
          ([], []) -> []
        | ([], h2::t2) -> h2:: badAppend [] t2
        | (h1 :: t1, [])  -> h1 :: badAppend t1 []
        | (h1 :: t1, h2::t2)  -> h1 :: badAppend t1 l2;;
# let xs = [1;2] and ys = [3;4]    let zs = badAppend xs ys;;
val xs : int list = [1; 2]
val ys : int list = [3; 4]
val zs : int list = [1; 2; 3; 4]
```

Jak teraz wygląda reprezentacja wewnętrzna tych list?



Skopiowane zostały obie listy. Złożoność obliczeniowa (czasowa i pamięciowa) jest liniowa względem sumy długości obu list. Jednak jeśli dopuszczalna byłaby modyfikacja wartości elementów list, to takie rozwiązanie może być preferowane.

Wzorce warstwowe (ang. as-patterns , layered patterns)

pat **as** *ident*

OCaml

```
# let ((n,w) as d, b) = x;;
val n : string = "Smith"
val w : int = 25
val d : string * int = ("Smith", 25)
val b : bool = true

# let f1 = fun ((x,y),z) -> (x,y,(x,y),z);;                    (* patrz str. 30 *)
val f1 : ('a * 'b) * 'c -> 'a * 'b * ('a * 'b) * 'c = <fun>

# let f2 = fun ((x,y) as p,z) -> (x,y,p,z);;
val f2 : ('a * 'b) * 'c -> 'a * 'b * ('a * 'b) * 'c = <fun>
```

Wykorzystanie wzorców warstwowych może polepszyć czytelność oraz efektywność czasową i pamięciową. Funkcja f2 jest nie tylko czytelniejsza, ale też efektywniejsza niż f1. Będzie o tym mowa na wykładzie 6.

Wzorce warstwowe (ang. as-patterns, layered patterns)

ident @ pat **Scala**

```
val (d@(n,w),b) = (("Smith", 25), true)
d: (String, Int) = (Smith,25)
n: String = Smith
w: Int = 25
b: Boolean = true

def f[A,B,C](k:((A,B),C)) = {  // porównaj str. 30
  val (p@(x,y),z) = k
  (x,y,p,z)
}                                  // lub tak
def f[A,B,C](k:((A,B),C)) =
  k match {
  case (p@(x,y),z) => (x,y,p,z)
}
```

Dopasowanie do wzorca - korzyści

```
let rec sqr_list l =
     match l with
         [] -> []
       | h::t -> h*h :: sqr_list t;;
```

- Czytelność, porównaj:

```
let rec sqr_list l =
    if l = [] then []
    else let h = List.hd l in h*h :: sqr_list (List.tl l);;
```

- Efektywność, porównaj:

```
let rec sqr_list l =
    if l = [] then []
    else (List.hd l)*(List.hd l) :: sqr_list (List.tl l);;
```

- Bezpieczeństwo – kompilator statycznie sprawdza kompletność wzorca.

Dodatek

Arytmetyka liczb całkowitych i zmiennopozycyjnych w języku Java

W językach Scala i OCaml arytmetyka jest implementowana analogicznie (mutatis mutandis).

Kodowanie liczb całkowitych

Liczby całkowite nieujemne są kodowane w naturalnym zapisie dwójkowym:

$$w = \sum_{i=0}^{n-1} 2^i b_i$$

Dla liczb całkowitych ujemnych stosowany jest zapis w kodzie uzupełnień do dwóch:

$$w = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

Otrzymywanie uzupełnienia do dwóch: dodaj 1 do uzupełnienia liczby do 1, czyli reprezentacja (-X) ≡ ~X+1

Np. 3 ≡ 0011 -3 ≡ 1100 + 0001 = 1101 (dla reprezentacji czterobitowej).

Dodawanie:

0010 (= 2)	0010 (= 2)	1110 (= -2)	1110 (= -2)
0011 (= 3)	1101 (= -3))	0011 (= 3)	1101 (= -3)
------	-------	-------	-------
0101 (= 5)	1111 (= -1)	0001 (= 1)	1011 (= -5)

Odejmowanie: Y - X = Y + (-X)

Arytmetyka liczb całkowitych

Arytmetyka liczb całkowitych jest arytmetyką uzupełnień do dwóch; jeśli wartość wyrażenia przekracza zakres swojego typu, to jest pobierana modulo ten zakres. Maksymalne i minimalne wartości są dostępne przez stałe `MIN_VALUE` i `MAX_VALUE` klas opakowujących. Obliczenia w arytmetyce całkowitej nie powodują nigdy przekroczenia zakresu, lecz się „zawijają”.

Dzielenie całkowite obcina wynik w kierunku zera, to znaczy odrzucana jest część ułamkowa wyniku, np. `7/2` daje `3`, a `-7/2` daje `-3`. Dla typów całkowitych (i zmiennoprzecinkowych) dzielenie i dzielenie modulo spełniają równanie:

```
(x/y)*y + x%y == x
```

Zatem `7%2` daje w wyniku `1`, a `-7%2` daje `-1`. Dzielenie oraz dzielenie modulo przez zero są zakazane i powodują zgłoszenie wyjątku ArithmeticException.

Operacje arytmetyczne na znakach są wykonywane w arytmetyce całkowitoliczbowej, przy czym znaki są niejawnie przekształcane do typu `int`.

Arytmetyka liczb zmiennopozycyjnych

Arytmetyka zmiennopozycyjna w Javie stanowi podzbiór standardu IEEE-754-1985. Dodawanie dwóch wartości nieskończonych o jednakowych znakach daje w wyniku taką samą nieskończoność, a wartość NaN (Not a Number), jeśli ich znaki były różne. Odejmowanie dwóch wartości nieskończonych o jednakowych znakach daje w wyniku NaN, a nieskończoność takiego samego znaku jak lewy argument, jeśli ich znaki były różne, np. $\infty - (-\infty)$ daje ∞ . Operacje arytmetyczne, w których wystąpi wartość NaN, dają w wyniku NaN. Nadmiar daje w wyniku nieskończoność odpowiedniego znaku, a niedomiar - zero odpowiedniego znaku. `+0.0` i `-0.0` są sobie równe, jednak `1f/0f` daje $+\infty$, a `1f/-0f` daje $-\infty$. Stałe $\pm\infty$ oraz NaN typów `float` oraz `double` są zdefiniowane w klasach opakowujących `Float` i `Double` jako `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN`, np. `Double.POSITIVE_INFINITY`. Do sprawdzania równości użyj metody `isNaN`. Operator `%` dla liczb zmiennopozycyjnych zachowuje się analogicznie jak dla liczb całkowitych. Inną metodą jest `Math.IEEERemainder`.

x	y	x/y	x%y
Wartość skończona	±0.0	±∞	NaN
Wartość skończona	±∞	±0.0	x
±0.0	±0.0	NaN	NaN
±∞	Wartość skończona	±∞	NaN
±∞	±∞	NaN	NaN