

Paradygmaty programowania

dr inż. Zdzisław Spławski

Katedra Inżynierii Oprogramowania

Wydział Informatyki i Zarządzania

Politechniki Wrocławskiej

p. 201/7a, D2

e-mail: zdzislaw.splawski@pwr.edu.pl

Materiały i informacje

- Wykład i ćwiczenia stanowią grupę kursów, z jedną oceną z egzaminu, zmodyfikowaną aktywnością na ćwiczeniach. Możliwe jest niedopuszczenie do egzaminu z powodu niedostatecznych postępów na ćwiczeniach!
- Drugim kursem jest laboratorium z oddzielną oceną.
- Materiały do wykładu i listy zadań na ćwiczenia będą umieszczane pod adresem:

<http://eportal.pwr.edu.pl/> (logowanie jak do JSOS)

Należy wybrać:

Kursy wydziałowe → Wydział Informatyki i Zarządzania
→ Programowanie → Paradygmaty programowania

Cel

- Poznanie:
 - podstawowych konstrukcji językowych, występujących w językach programowania
 - podstawowych paradygmatów programowania (modeli obliczeniowych)
- Zdobyć umiejętności wykorzystania technik programistycznych, właściwych dla stosowanego paradygmatu programowania
- Zdobyć umiejętności łączenia mechanizmów z różnych paradygmatów w jednym programie
- Nacisk jest położony na podstawowe pojęcia programowania, a nie na składnię i szczegóły implementacji w konkretnych językach

Materiały pomocnicze

Wiadomości, potrzebne do zaliczenia przedmiotu, będą podane na wykładach i na udostępnianych slajdach. Szczegóły, dotyczące wykorzystywanych języków programowania, można znaleźć w ich specyfikacjach:

- <http://www.scala-lang.org/files/archive/spec/2.11/> Scala
- <http://caml.inria.fr/pub/docs/manual-ocaml/index.html> OCaml
- <http://docs.oracle.com/javase/specs/jls/se8/html/index.html> Java

Wiele materiałów (lub informacje o nich) można znaleźć na stronach domowych wykorzystywanych języków.

- <http://www.scala-lang.org/> Scala
- <http://caml.inria.fr/ocaml/> Ocaml
- oraz <http://ocaml.org/>
- <http://www.oracle.com/technetwork/java/> Java

Na następnej stronie jest kilka wybranych pozycji literaturowych. Pierwsza z nich [PKTM] dobrze omawia najważniejsze paradygmaty i koncepcje programowania, ilustrując je za pomocą wieloparadygmatowego języka programowania Oz.

Wybrana literatura

- [PKTM] P.Van Roy, S.Haridi. *Programowanie. Koncepcje, techniki i modele*. Helion 2005
<http://www.info.ucl.ac.be/~pvr/book.html>
- M. Odersky, L. Spoon, B. Venners. *Programming in Scala (3rd ed.)*. Artima 2016
<http://www.artima.com/pins1ed/> (tu jest tylko pierwsze wydanie)
- G.Balcerek. *Język programowania Scala (wyd. 2)*. G. Balcerek 2016
- E.Chailloux, P.Manoury, B.Pagano. *Developing Applications with Objective Caml*.
<http://caml.inria.fr/pub/docs/oreilly-book/>
- <https://www.janestreet.com/technology/>
- <http://www.ffconsultancy.com/index.html>
- P.-Y.Saumont, *Java. Programowanie funkcyjne*. Helion 2017
<https://helion.pl/ksiazki/java-programowanie-funkcyjne-pierre-yves-saumont,javapf.htm>
- J.Backfield, *Programowanie funkcyjne. Krok po kroku*. Helion 2015
<http://helion.pl/ksiazki/programowanie-funkcyjne-krok-po-kroku-joshua-backfield,pfukpk.htm>

Wykład 1

Wprowadzenie do przedmiotu.

Podstawy programowania funkcyjnego w środowisku interakcyjnym.

Abstrakcja i reprezentacja

Paradygmaty programowania

Co to jest programowanie funkcyjne?

Co to jest OCaml?

Praca interakcyjna w cyklu REPL

Przegląd podstawowych konstrukcji języka OCaml i Scala na przykładach

Statyczne wyznaczanie zakresu

Typy bazowe i typy strukturalne: listy i krotki

Wyrażenia funkcyjne i deklaracje funkcji

Funkcje w języku Scala

Funkcje polimorficzne (polimorfizm parametryczny)

Dodatek: Złożoność obliczeniowa. Podstawowe pojęcia.

Paradygmat

Paradygmat <łac. *paradigma* z gr. *παράδειγμα* = przykład, wzór > **to przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie itp.; wzorzec, model.**

Pojęcie *paradygmatu programowania* nie jest ściśle zdefiniowane. Możliwe są różne klasyfikacje.

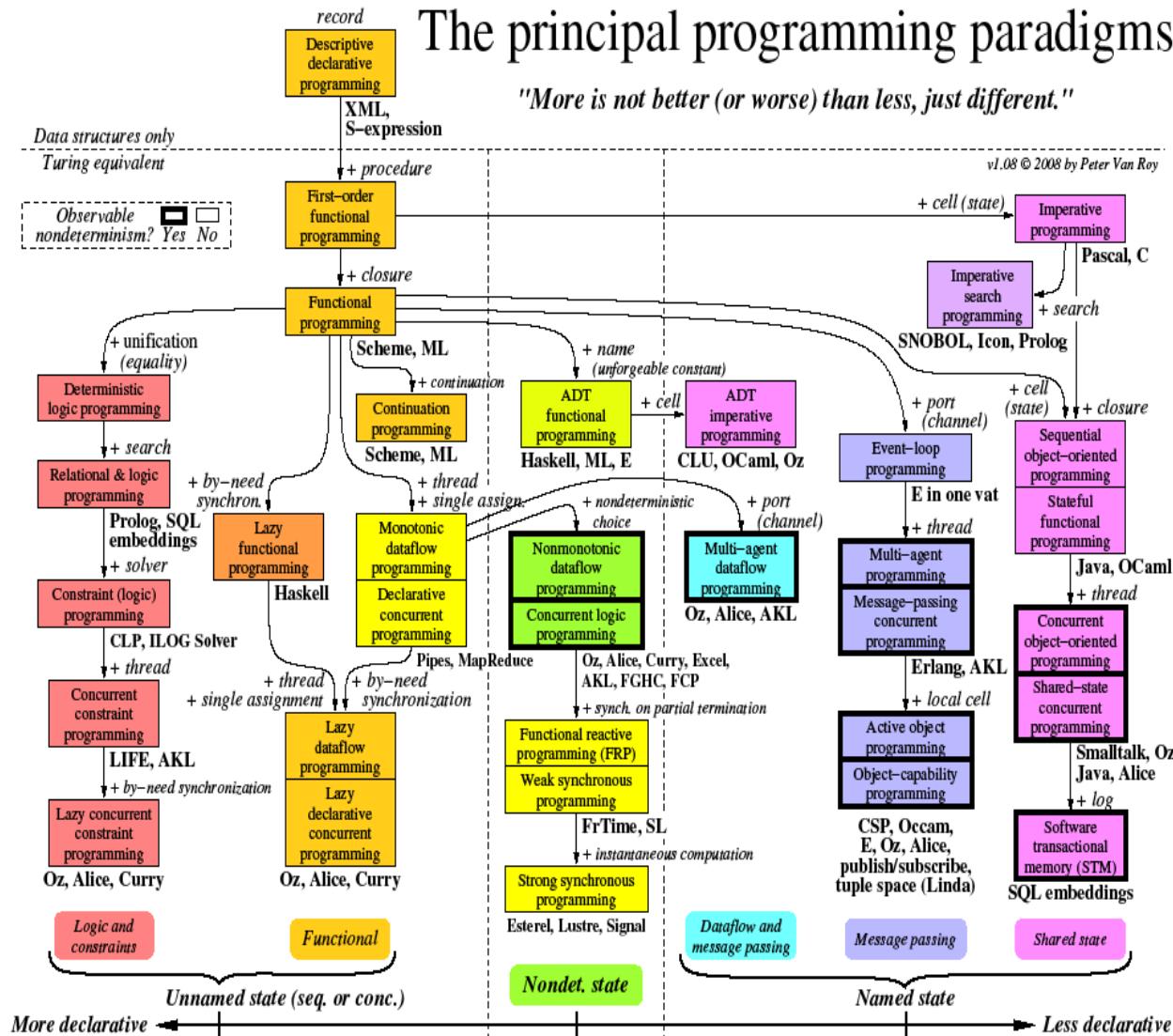
Paradygmaty programowania

- Programowanie deklaratywne (ang. declarative programming)
- Programowanie imperatywne (ang. imperative)
- Programowanie funkcyjne (ang. functional)
- Programowanie logiczne (ang. logic)
- Programowanie proceduralne (ang. procedural)
- Programowanie obiektowe (ang. object-oriented)
- Programowanie współbieżne (ang. concurrent)
- Programowanie rozproszone (ang. distributed)
- Programowanie z więzami (ograniczeniami) (ang. constrained)
- Programowanie sterowane zdarzeniami (ang. event-driven)
- ...

The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.08 © 2008 by Peter Van Roy



Explanations

See "Concepts, Techniques, and Models of Computer Programming".

The chart classifies programming paradigms according to their kernel languages (the small core language in which all the paradigm's abstractions can be defined). Kernel languages are ordered according to the creative extension principle: a new concept is added when it cannot be encoded with only local transformations. Two languages that implement the same paradigm can nevertheless have very different "flavors" for the programmer, because they make different choices about what programming techniques and styles to facilitate.

When a language is mentioned under a paradigm, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment.

State is the ability to remember information, or more precisely, to store a sequence of values in time. Its expressive power is strongly influenced by the paradigm that contains it. We distinguish four levels of expressiveness, which differ in whether the state is unnamed or named, deterministic or nondeterministic, and sequential or concurrent. The least expressive is functional programming (threaded state, e.g., DCGs and monads: unnamed, deterministic, and sequential). Adding concurrency gives declarative concurrent programming (e.g., synchrocells: unnamed, deterministic, and concurrent). Adding nondeterministic choice gives concurrent logic programming (which uses stream mergers: unnamed, nondeterministic, and concurrent). Adding ports or cells, respectively, gives message passing or shared state (both are named, nondeterministic, and concurrent). Nondeterminism is important for real-world interaction (e.g., client/server). Named state is important for modularity.

Axes orthogonal to this chart are typing, aspects, and domain-specificity. Typing is not completely orthogonal: it has some effect on expressiveness. Aspects should be completely orthogonal, since they are part of a program's specification. A domain-specific language should be definable in any paradigm (except when the domain needs a particular concept).

Metaprogramming is another way to increase the expressiveness of a language. The term covers many different approaches, from higher-order programming, syntactic extensibility (e.g., macros), to higher-order programming combined with syntactic support (e.g., meta-object protocols and generics), to full-fledged tinkering with the kernel language (introspection and reflection). Syntactic extensibility and kernel language tinkering in particular are orthogonal to this chart. Some languages, such as Scheme, are flexible enough to implement many paradigms in almost native fashion. This flexibility is not shown in the chart.

<http://www.info.ucl.ac.be/~pvr/paradigms.html>

Język programowania jest ważny

- Język programowania służy do precyzyjnego opisywania zadań, które mają być wykonane przez komputer.
- Język programowania służy *również* do przekazywania idei między ludźmi. Jak każdy język dostarcza on środków do werbalizowania idei, dotyczących najrozmaitszych dziedzin życia.
- Dobrze zaprojektowany język programowania wysokiego poziomu ułatwia obydwie te zadania, prowadząc do programów odpornych, szybkich i łatwych do zrozumienia.

Język programowania nie jest ważny

- Język programowania można porównać do skomplikowanego narzędzia: jego efektywne wykorzystanie wymaga znajomości wielu szczegółów.
- Jednak narzędzia we współczesnym świecie zmieniają się bardzo szybko. Programistę, znającego jeden język można porównać do kompozytora, który potrafi komponować utwory tylko na altówkę...
- Języki programowania ewoluują, podczas gdy podstawowe koncepcje i techniki programowania pozostają (względnie) stabilne.

Motto

Profesjonalista zna kilka języków programowania i w razie potrzeby uczy się nowych. Nie ma „jedyne go najlepszego języka” dla wszystkich ludzi i dziedzin. W istocie wszystkie znane nam większe systemy zostały zbudowane przy użyciu więcej niż jednego języka programowania.

Bjarne Stroustrup

Programowanie : Teoria i praktyka z wykorzystaniem C++

Helion 2010

Języki programowania

- Języki programowania w różnym stopniu wspierają różne paradygmaty programowania, np.
 - Smalltalk: programowanie obiektowe
 - Haskell: programowanie funkcyjne
 - Prolog: programowanie logiczne
 - ...
- Chcemy *szybko* poznać przynajmniej najważniejsze paradygmaty. W jaki sposób?
- Poznać kilka języków (nowa składnia, nowa semantyka ...)
- Wykorzystać wieloparadygmatyczny język programowania, np. Scala, OCaml, Oz, ...

Kilka ważnych pojęć

- *Statyczna* cecha języka jest związana z fazą kompilacji.
- *Dynamiczna* cecha języka jest związana z fazą wykonywania.
- *Słaba typizacja* (kontrola typów) : wewnętrzna reprezentacja typu może podlegać manipulacjom.
- *Silna (ściśła) typizacja* (ang. strong typing): błędy typów są zawsze wykrywane.
- *Koercja*: operacja semantyczna, przekształcająca argument do oczekiwanego typu. Szczególnym przypadkiem koercji jest *promocja*.
- *Polimorfizm* <gr. πολυζ = liczny + μορφη = postać > ogólnie oznacza wielopostaciowość i umożliwia przypisanie różnych typów temu samemu programowi.

Zmienna

- Składniowo: *identyfikator*
- Semantycznie: *zmienna składowana* (ang. store variable)
 1. zmienna w sensie matematycznym (skrót dla wartości) (ang. immutable variable)
 - używana w czystym programowaniu funkcyjnym
 2. komórka pamięci, w której można umieścić dowolną zawartość (ang. mutable variable)
 - używana w programowaniu imperatywnym
 3. zmienna jednokrotnego przypisania (zmienna logiczna, zmienna przepływu danych)
 - umożliwia obliczenia na wartościach częściowych

W języku OCaml i Haskell używane są zmienne w sensie 1. W językach imperatywnych zmienne są rozumiane w sensie 2. W językach Prolog i Oz wykorzystywane są zmienne jednokrotnego przypisania. W języku Scala możemy deklarować zmienne w sensie 1 i 2.

Wartość pierwszej kategorii

Element języka programowania jest wartością *pierwszej kategorii* (ang. first class value, first class citizen) jeśli:

- może być przypisany do zmiennej
- może być przekazywany jako argument do funkcji (metody, procedury)
- może być zwracany jako wynik funkcji (metody, procedury)

Przykłady: wartości numeryczne we wszystkich językach programowania, funkcje w OCamlu lub Haskellu, referencje do obiektów w Javie, ...

Statyczne i dynamiczne określanie typów

- Typizacja statyczna (ang. static typing)
 - wykrywanie większej liczby błędów logicznych w czasie kompilacji
 - efektywniejszy kod programu
 - czytelniejszy program
- Typizacja dynamiczna (ang. dynamic typing)
 - większa elastyczność programowania
 - mniej efektywny kod programu

Różne rodzaje polimorfizmu zwiększają elastyczność języków z typizacją statyczną.

Najważniejsze rodzaje polimorfizmu

- Polimorfizm parametryczny
- Polimorfizm inkluzyjny
 - Podtypowanie
 - Dziedziczenie
- Polimorfizm ograniczeniowy
 - = polimorfizm parametryczny
 - + polimorfizm podtypowy

Deklaratywny model obliczeniowy

- Jeden z najbardziej podstawowych modeli obliczeniowych.
 - powiedz, *co* chcesz zrobić
 - niech komputer zdecyduje, *jak* to zrobić
- Uwzględnia kardynalne idee dwóch głównych paradygmatów deklaratywnych: funkcyjnego i logicznego
- Programowanie bezstanowe
 - struktury danych są *niemodyfikowalne*

Funkcyjny model obliczeniowy

(Ścisły) funkcyjny model obliczeniowy otrzymujemy go przez nałożenie dwóch ograniczeń na model deklaratywny, tak aby funkcje były zawsze obliczane na wartościach pełnych:

- deklaracja zmiennej jest połączona z wiązaniem z wartością;
- używana jest składnia funkcji, a nie procedur.

Program funkcyjny jest funkcją w sensie matematycznym. Ta własność jest bardzo pożądana w programowaniu współbieżnym.

Co to jest programowanie funkcyjne?

Programowanie funkcyjne (ang. functional programming) to sposób konstruowania programów, w których:

- jedyną akcją jest wywołanie funkcji,
- jedynym sposobem podziału programu na części jest wprowadzenie nazwy dla funkcji,
- jedyną regułą kompozycji jest składanie funkcji,
- każde wyrażenie może w danym kontekście być zastąpione przez swoją wartość (*przezroczystość referencyjna*, ang. referential transparency).

Czyli w językach funkcyjnych:

- każdy program jest wyrażeniem,
- każde wyrażenie wyznacza wartość,
- funkcje są wartościami pierwszej kategorii (ang. first-class citizens) – mogą być przekazywane jako argumenty, otrzymywane jako wyniki lub stanowić część struktur danych.

W przeciwieństwie do programowania imperatywnego, opartego na modyfikacjach stanu programu przez instrukcje, programowanie funkcyjne (aplikatywne) jest stylem programowania opartym na obliczaniu wartości wyrażeń, a w “czystych” językach programowania funkcyjnego nie ma instrukcji przypisania.

Zalety programowania funkcyjnego

Modularność, tzn. czyste funkcje są łatwiejsze w:

- testowaniu
- wykorzystywaniu
- zrównoleglaniu
- uogólnianiu
- dowodzeniu poprawności (zgodności ze specyfikacją)

Model funkcyjny nie nakłada ograniczeń na to, *co* można wyrazić w języku programowania; ograniczenie dotyczy wyłącznie tego, *jak* to wyrazić.

Bez znajomości tego paradygmatu nie można w pełni wykorzystać możliwości żadnego współczesnego języka programowania (C#, C++, Java ...). Jest on teraz powszechnie wykorzystywany w programowaniu współbieżnym, LINQ (Language Integrated Query) itd.

Najważniejsze języki funkcyjne

Język	Typizacja	Wyznaczanie zakresu	Ewaluacja	Efekty uboczne
Lisp	dynamiczna	dynamiczne	gorliwa	tak
Scheme	dynamiczna	statyczne	gorliwa + leniwa (kontynuacje)	tak
Clojure (platforma JVM)	dynamiczna	statyczne lub dynamiczne	gorliwa	tak
Standard ML OCaml	statyczna, silna	statyczne	gorliwa	tak
Haskell	statyczna, silna	statyczne	leniwa	nie

Scala

Naszym głównym narzędziem będzie wieloparadygmatyczny język Scala z implementacją na platformie JVM. Scala jest językiem skalowalnym, tzn. ten sam mechanizm językowy równie dobrze opisuje małe, jak i duże komponenty programowe.

Najważniejsze cechy języka Scala są przedstawione na stronie:

<http://www.scala-lang.org/node/25>

- Scala jest językiem z mocną typizacją statyczną, łączącym programowanie obiektowe i funkcyjne.
- Scala jest językiem obiekowym, w którym każda wartość jest obiektem, a każda operacja jest metodą.
- Scala jest językiem funkcyjnym, w którym każda funkcja jest wartością pierwszej kategorii.
- Scala umożliwia użycie mechanizmu dopasowania do wzorca.
- Scala wspiera programowanie współbieżne za pomocą wątków i aktorów.
- Scala pozwala na „bezszwowe” wykorzystywanie kodu, napisanego w języku Java.

Prace nad językiem Scala trwały od roku 2001 w École Polytechnique Fédérale de Lausanne (EPFL). Pierwsze wersje stały się publicznie dostępne w styczniu 2004r. na platformie JVM i w czerwcu 2004r. na platformie .NET.

OCaml

Na wykładzie będą też podawane przykłady w języku OCaml.
OCaml (Objective Caml) jest językiem programowania funkcyjnego:

- z silną typizacją (statyczną),
- z parametrycznym polimorfizmem,
- z automatyczną inferencją typów,
- z automatycznym odświeżaniem pamięci,
- z mechanizmem wyjątków,
- z parametryzowanymi modułami,
- z klasami i obiektami,
- umożliwiającym programowanie współbieżne (wątki),
- w bezpieczny sposób włączający mechanizmy programowania imperatywnego,
- z efektywną implementacją.

Język F# z platformy .Net jest w znacznym stopniu kompatybilny z językiem OCaml.

OCaml

OCaml (Objective Caml = Categorical Abstract Machine Language) powstał w 1984 roku (jako Caml, potem Caml Light) i jest ciągle rozwijany w INRIA (Institut National de Recherche en Informatique et en Automatique) we Francji (<http://caml.inria.fr>). OCaml należy do rodziny języków, wywodzących się z ML (Meta Language). Sztandarowym reprezentantem tej rodziny jest Standard ML, z wieloma implementacjami (<http://sml-family.org/> <http://www.smlnj.org/sml97.html>). OCaml różni się od pozostałych języków z tej rodziny głównie tym, że jako jedyny umożliwia również programowanie obiektowe. Języki programowania funkcyjnego od „zawsze” były stosowane w projektach naukowych (pierwsze zastosowania Lispu, który powstał prawie równocześnie z językiem Fortran, miały związek ze sztuczną inteligencją), lecz obecnie znajdują coraz więcej zastosowań przemysłowych (<http://homepages.inf.ed.ac.uk/wadler/realworld/>).

Teoretycznym modelem programowania funkcyjnego jest λ - rachunek (rachunek lambda), wprowadzony przez Alonzo Churcha w 1932 roku.

OCaml - wykorzystanie przemysłowe

- Jane Street (<https://www.janestreet.com/technology/>)
OCaml jest używany w tej międzynarodowej firmie jako przemysłowy język programowania. Firma jest też zaangażowana w jego rozwój. Została tam stworzona m.in. bogata biblioteka Core (<http://janestreet.github.io/>), będąca (niekompatybilną!) alternatywą dla standardowej biblioteki OCaml'a oraz biblioteka Async, wspierająca programowanie współbieżne.
Biblioteki te są wykorzystywane w kodzie z książki:
Y.Minsky, A.Madhavapeddy, J.Hickey, *Real World OCaml*. O'Reilly 2013
<https://realworldocaml.org/>
- OCamlPro (<http://www.ocamlpro.com/>).
Ta firma również wykorzystuje język Ocaml i jest zaangażowana w jego rozwój.

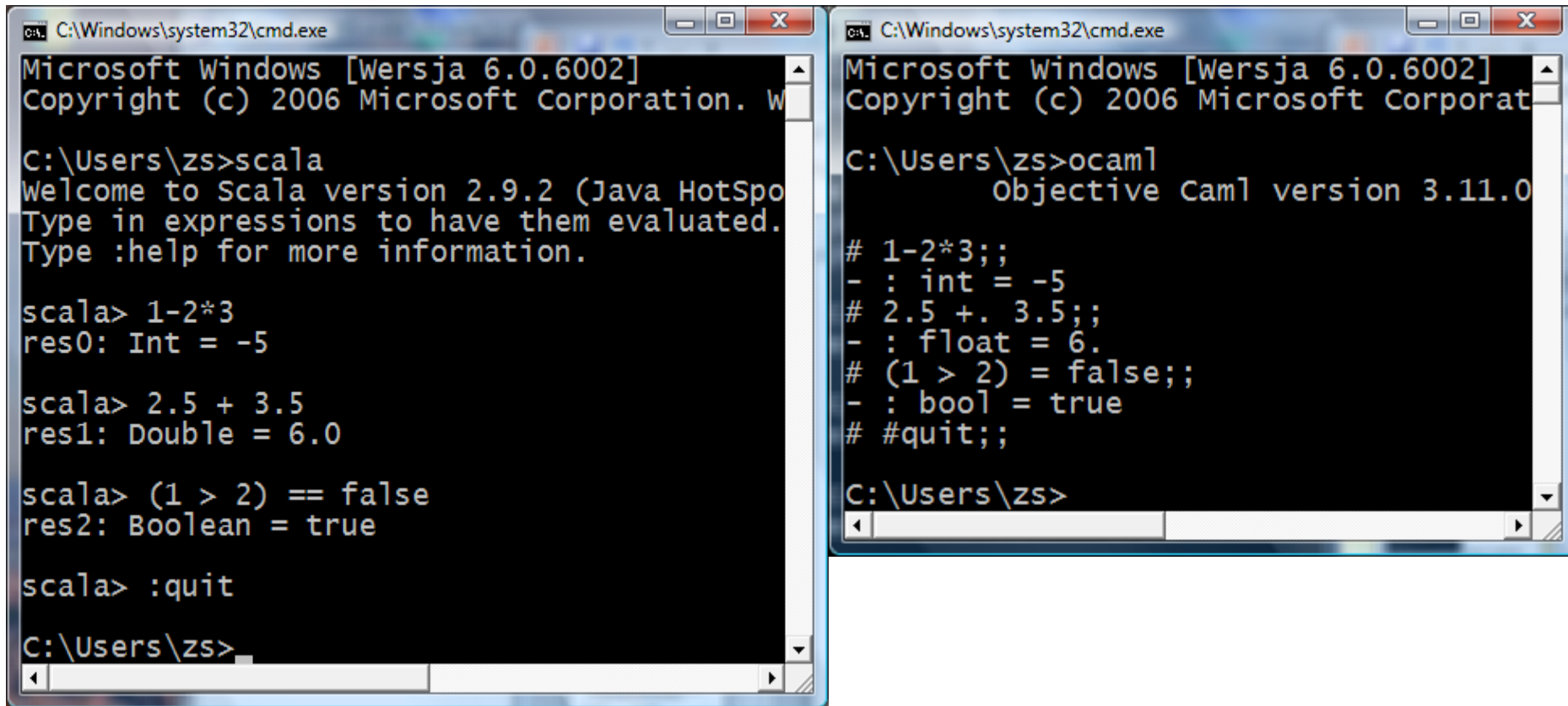
System typów w języku OCaml

- OCaml jest językiem silnie (ściśle) typizowanym (ang. strongly typed), tzn. w jego systemie typów nie ma żadnych luk.
- Jest on typizowany statycznie (ang. statically typed), tzn. łączenie typów z identyfikatorami i sprawdzanie poprawności typów jest przeprowadzane tylko raz w czasie kompilacji, co gwarantuje poprawność programu w czasie wykonania (jeśli chodzi o typy).
- OCaml jest typizowany niejawnie (ang. implicitly typed), tzn. programista nie jest zmuszony do jawnego wprowadzania informacji o typach – kompilator rekonstruuje najogólniejszy typ z kontekstu. Informacje o typach można jednak podawać, jeśli jest to konieczne do lepszego zrozumienia programu.

Silna statyczna typizacja umożliwia znalezienie większości błędów logicznych już w czasie kompilacji i generowanie efektywniejszego kodu. Funkcje polimorficzne usuwają wiele ograniczeń związanych ze statyczną typizacją w konwencjonalnych imperatywnych językach programowania.

Praca interakcyjna w cyklu REPL

Każdy funkcyjny język programowania umożliwia pracę interakcyjną w cyklu REPL (REPL = Read-Evaluate-Print Loop). Najprostszym środowiskiem jest tu wiersz poleceń (okno terminala).



The image shows two side-by-side terminal windows, both titled "C:\Windows\system32\cmd.exe".

The left window shows a Scala REPL session. It starts with the prompt "C:\Users\zs>scala", followed by the Scala welcome message: "Welcome to Scala version 2.9.2 (Java HotSpot(TM) 6.0.6002) Type in expressions to have them evaluated. Type :help for more information." The user then enters several expressions: "scala> 1-2*3" (result: "res0: Int = -5"), "scala> 2.5 + 3.5" (result: "res1: Double = 6.0"), "scala> (1 > 2) == false" (result: "res2: Boolean = true"), and "scala> :quit". The prompt returns to "C:\Users\zs>".

The right window shows an OCaml REPL session. It starts with the prompt "C:\Users\zs>ocaml", followed by the OCaml welcome message: "Objective Caml version 3.11.0". The user then enters several expressions: "# 1-2*3;;" (result: "- : int = -5"), "# 2.5 +. 3.5;;" (result: "- : float = 6."), "# (1 > 2) = false;;" (result: "- : bool = true"), and "# #quit;;". The prompt returns to "C:\Users\zs>".

Skrypty

Nasz kurs zaczniemy od tworzenia i wykonywania skryptów funkcyjnych w językach Scala i OCaml.

Skrypt (ang. script) jest plikiem z ciągiem poleceń, przeznaczonych do sekwencyjnego wykonania przez jakiś program. Skrypt ma też dostęp do argumentów z wiersza poleceń.

Plik z kodem źródłowym w języku Scala powinien mieć rozszerzenie `.scala` (np. `foo.scala`), a w języku OCaml – rozszerzenie `.ml` (np. `foo.ml`).

Skrypt z pliku `foo.scala` można załadować i wykonać w konsoli interaktywnej za pomocą polecenia `:load`, np.

```
scala> :load foo.scala
```

Skrypt z pliku `foo.ml` można załadować i wykonać w konsoli interaktywnej za pomocą polecenia `#use`, np.

```
# #use "foo.ml";;
```

Środowisko interakcyjne (Eclipse)

Scala

- Wybierz perspektywę Scali. W wybranym projekcie utwórz plik (*File>>New>>File*), który będzie zawierał skrypt w języku Scala, np. *W1.scala*
- W oknie edytora wprowadź treść skryptu, np.
1-2*3
2.5 + 3.5
(1 > 2) == false
- Ustaw kursor na odpowiedni wiersz skryptu (lub zaznacz myszą kilka wierszy) i na pasku narzędzi kliknij *Run Selection in Scala Interpreter*. Na dole w oknie interpretera pojawi się wynik.

OCaml

- Wybierz perspektywę OCaml. W wybranym projekcie utwórz plik (*File>>New>>File*), który będzie zawierał skrypt w języku OCaml, np. *W1.ml*
- W oknie edytora wprowadź treść skryptu, np.
1-2*3;;
2.5+. 3.5;;
(1>2) = false;;
- Ustaw kursor na odpowiedni wiersz skryptu (lub zaznacz myszą kilka wierszy) i na pasku menu kliknij *Ocaml>>Eval in Toplevel*. Na dole w oknie interpretera pojawi się wynik.
- Innym wygodnym IDE do pracy w OCamlu jest Emacs z wtyczką np. Tuareg.

Kalkulator

Scala

```
scala> 1-2*3  
res0: Int = -5  
scala> 2.5 + 3.5  
res1: Double = 6.0  
scala> !(1 > 2)  
res2: Boolean = true  
scala> (1 > 2) != false  
res3: Boolean = false  
scala> res2  
res4: Boolean = true
```

OCaml

```
# 1-2*3;;  
- : int = -5  
# 2.5 +. 3.5;;  
- : float = 6.  
# not (1 > 2);;  
- : bool = true  
# (1>2) <> false;;  
- : bool = false
```

Składniowo program w języku OCaml jest ciągiem fraz (wyrażeń i definicji), które mogą być bezpośrednio wykonane. Koniec frazy w pracy interakcyjnej jest oznaczany przez podwójny średnik (;).

Identyfikatory zmiennych

Scala

```
scala> val x1 = 3 + 2  
x1: Int = 5  
scala> val napis = "Ala "+"ma "+"kota"  
napis: String = Ala ma kota
```

OCaml

```
# let x1 = 3+2;;  
val x1 : int = 5  
# let napis = "Ala "^"ma "^"kota";;  
val napis : string = "Ala ma kota"
```

W językach funkcyjnych deklarowana zmienna **musi** być od razu związana z wartością, która **nie może** być zmieniana.

```
scala> x1 = 6  
<console>: x1: 11: error: reassignment to val  
      x1 = 6  
      ^
```

```
# x1 = 6;;  
- : bool = false (* to jest porównanie! *)
```

Potem tę samą zmienną można związać z inną wartością (być może innego typu), co pokazuje następny przykład.

Identyfikatory zmiennych

Scala

```
scala> val x1 = 7.5  
x1: Double = 7.5  
scala> x1 + 6.0  
res5: Double = 13.5  
scala> x1 + 6  
res6: Double = 13.5
```

OCaml

```
# let x1 = 7.5;;  
val x1 : float = 7.5  
# x1 +. 6. ;;  
- : float = 13.5  
# x1 +. float_of_int 6;;  
- : float = 13.5
```

Uwaga. W języku OCaml nie są przeprowadzane automatycznie żadne koercje typów. Przy obliczeniach numerycznych użyteczne mogą być biblioteczne funkcje koercji *float_of_int* i *int_of_float*.

Funkcja *int_of_float* obcina część ułamkową (= *truncate*).

W języku Scala koercje przeprowadzane są podobnie jak w języku Java.

Statyczne wyznaczanie zakresu identyfikatora

Scala

```
scala> { val x = x1+x1
      |   val y = 2
      |   x + x +
      |   { val x = 10.0
          |     x + y
          |   }
      | } + 1.0
res7: Double = 43.0
```

OCaml

```
# (let x = x1+.x1 and y = 2.0 in
   x +. x +.
   (let x=10.0 in x +. y)
) +. 1.0;;
- : float = 43.
```

Blok jest fragmentem programu źródłowego, zawierającym deklaracje i instrukcje, wykorzystywanym do kontroli widoczności identyfikatorów.

Fragment programu, w którym identyfikator jest *widoczny* (można się do niego odwołać) nazywamy *zakresem* (ang. scope) zmiennej. Zmienna *nielokalna* (*globalna*) bloku jest w tym bloku widoczna, ale nie jest w nim zadeklarowana.

Statyczne wyznaczanie zakresu identyfikatora (ang. static scoping, lexical scoping) polega na prostym zbadaniu tekstu programu. *Dynamiczne wyznaczanie zakresu* identyfikatora (ang. dynamic scoping) jest bardzo rzadko stosowane (np. w języku Lisp).

Pary i krotki

Scala

```
scala> val k3 = (3+4, 2.0, 2<4)
k3: (Int, Double, Boolean) = (7,2.0,true)
/* W języku Scala krotki są instancjami jednej
 * z klas TupleN, gdzie 2<= N <= 22, które
 * mają zdefiniowane wszystkie selektory
 */
scala> k3._1
res8: Int = 7
scala> k3._3
res9: Boolean = true
// krotki można porównywać
scala> k3 == (8-1, 2.0, 2==2)
res10: Boolean = true
```

OCaml

```
# let k3=(3+4, 2. , 2<4);;
val k3 : int * float * bool = (7, 2., true)
(* W języku funkcyjnym z typizacją statyczną
   można mieć tylko selektory krotek
   o ustalonej liczbie elementów,
   np.w OCamlu fst, snd dla dowolnych par.
 *)
# let k3_1 (x,y,z) = x;;
val k3_1 : 'a * 'b * 'c -> 'a = <fun>
# k3_1 k3;;
- : int = 7
(* krotki można porównywać *)
# k3=(8-1,2.,2=2);;
- : bool = true
```

Pary i krotki c.d.

Scala

```
scala> val k2 = (3+4, (2.0 , 2<4))
k2: (Int, (Double, Boolean)) = (7,(2.0,true))

scala> k2._1
res11: Int = 7

scala> k2._2
res12: (Double, Boolean) = (2.0,true)

scala> (k2._2)._1
res13: Double = 2.0
```

OCaml

```
# let k2=(3+4, (2. , 2<4));;
val k2 : int * (float * bool) = (7, (2., true))

# fst k2;;
- : int = 7

# snd k2;;
- : float * bool = (2., true)

# fst(snd k2);;
- : float = 2.
```

Listy

Scala

```
scala> val xs = 1.0 :: x1 :: 2.5 :: Nil
xs: List[Double] = List(1.0, 7.5, 2.5)
// listy można porównywać
scala> xs == List(1.0 , 7.5, 2.0+0.5)
res14: Boolean = true
scala> xs.head    // lub xs head
res15: Double = 1.0
//UWAGA. Dla xs head pojawi się ostrzeżenie
warning: there was one feature warning: re-run with -feature for
details
W celu uniknięcia takich ostrzeżeń należy użyć klauzuli import
(wyjaśnienie będzie na wykładzie 3)
scala> import scala.language.postfixOps
scala> xs.tail    // lub xs tail
res16: List[Double] = List(7.5, 2.5)
```

OCaml

```
# let xs = 1. :: x1 :: 2.5 :: [];;
val xs : float list = [1.; 7.5; 2.5]
(* listy można porównywać *)
# xs=[1.; 7.5; 2. +. 0.5];;
- : bool = true
# List.hd xs;;
- : float = 1.
# List.tl xs;;
- : float list = [7.5; 2.5]
```

Listy c.d.

Scala

```
scala> xs.length // lub xs length
res17: Int = 3

scala> xs ++ List(9.0,10.0)
res18: List[Double] = List(1.0, 7.5, 2.5, 9.0, 10.0)

scala> List(1,2,3).reverse
res19: List[Int] = List(3, 2, 1)

scala> val xss = List(List(4.0,5.0), xs,
                      1.0 :: 2.0 :: 3.0 :: Nil)
xss: List[List[Double]] =
  List(List(4.0, 5.0), List(1.0, 7.5, 2.5),
        List(1.0, 2.0, 3.0))

scala> xss.length
res20: Int = 3
```

OCaml

```
# List.length xs;;
- : int = 3

# xs@[9.; 10.];;
- : float list = [1.; 7.5; 2.5; 9.; 10.]

# List.rev [1; 2; 3];;
- : int list = [3; 2; 1]

# let xss=[[4.;5.]; xs; 1. :: 2.:: 3. :: []];;

val xss : float list list =
  [[4.; 5.]; [1.; 7.5; 2.5]; [1.; 2.; 3.]]

# List.length xss;;
- : int = 3
```

Funkcje `length`, `reverse` oraz `rev` mają złożoność liniową względem długości listy.
Funkcje `append` (`@` oraz `++`) mają złożoność liniową względem długości pierwszej listy.

Funkcje

Scala

```
scala> val double = (x:Int) => 2*x
double: Int => Int = <function1>
scala> double(6)
res21: Int = 12

/* funkcja anonimowa
 * = literal funkcyjny */

scala> ((x:Int) => 2*x)
res22: Int => Int = <function1>

scala> ((x:Int) => 2*x) (6)
res23: Int = 12
```

Scala zwykle wymaga jawnego podania typów argumentów funkcji i ujęcia ich w nawiasy.

OCaml

```
# let double = fun x -> 2*x;;
val double : int -> int = <fun>

# double 6;;
- : int = 12

(* funkcja anonimowa
 * = literal funkcyjny *)

# fun x -> 2*x;;
- : int -> int = <fun>

# (fun x -> 2*x) 6;;
- : int = 12
```


Funkcje w języku Scala

Scala jest językiem obiektowym, więc każda wartość jest obiektem (w odróżnieniu od Javy nie ma tu typów pierwotnych), np.

```
scala> val s = 2.toString  
s: String = 2
```

Scala jest też językiem funkcyjnym, więc funkcje są wartościami pierwszej kategorii, a więc są obiektami (patrz wyżej). Funkcja w języku Scala jest instancją klasy, rozszerzającej jedną z cech (ang. trait) `FunctionN` z pakietu `scala`, gdzie $0 \leq N \leq 22$. Cechę `Function0` rozszerzają funkcje bezargumentowe, `Function1` rozszerzają funkcje z jednym argumentem, `Function2` rozszerzają funkcje z dwoma argumentami itd. Każda cecha `FunctionN` ma metodę `apply`, służącą do wywoływania funkcji. Szczegóły będą wyjaśnione później.

W językach funkcyjnych wszystkie funkcje są jednak jednoargumentowe!

W Scali jest inaczej, ponieważ jest ona językiem obiektowo-funkcyjnym, który ma być kompatybilny z językiem Java.

Funkcje c.d.

Scala

```
scala> def twice (x:Int) = 2*x
twice: (x: Int)Int
scala> twice (6)
res24: Int = 12
scala> twice (2+3)
res25: Int = 10
/* aplikacja funkcji do argumentu
 * wiąże najmocniej
 */
scala> twice (2)+3
res26: Int = 7
```

OCaml

```
(* to jest wygodny skrót notacyjny *)
# let twice x = x+x;;
val twice : int -> int = <fun>
# twice 6;;
- : int = 12
# twice (2+3);;
- : int = 10
(* aplikacja funkcji do argumentu
   wiąże najmocniej (lewostronnie)!
 *)
# twice 2+3;;
- : int = 7
```

Funkcje w języku Scala c.d.

W OCamlu `twice` jest funkcją z takim samym typem i taką samą reprezentacją wewnętrzną jak `double`. W Scali `twice` jest metodą, porównaj typy:

```
double: Int => Int = <function1>  
twice: (x: Int)Int
```

Metody nie są wartościami. Nie można ich wobec tego związać ze zmienną czy przekazywać jako argumenty (bez koercji typu, zwykle automatycznej). Metody można jednak łatwo przekształcić w funkcje.

```
scala> val twiceF = twice _  
twiceF: Int => Int = <function1>
```

Z tym zastrzeżeniem możemy metody traktować jak funkcje. Będzie to zwłaszcza potrzebne w przypadku funkcji polimorficznych. Szczegóły będą wyjaśnione później.

Funkcje c.d.

Scala

```
scala> def silnia(n:Int):Int =  
    if (n==0) 1 else n*silnia(n-1)  
  
silnia: (n: Int)Int  
scala> silnia (4)  
res27: Int = 24  
scala> silnia (-4)  
java.lang.StackOverflowError  
at .silnia(<console>:8)  
at .silnia(<console>:8)  
at .silnia(<console>:8)  
at ...
```

Dla funkcji rekurencyjnych w Scali należy jawnie podać typ wyniku.

OCaml

```
# let rec silnia n =  
    if n=0 then 1 else n*silnia(n-1);;  
  
val silnia : int -> int = <fun>  
  
# silnia 4;;  
- : int = 24  
  
# silnia (-4);;  
Stack overflow during evaluation  
(looping recursion?).
```

Wyjątki

- Sytuacje wyjątkowe w czystym modelu deklaratywnym wymagają często uciążliwego sprawdzania.
- *Wyjątki* (ang. exceptions) stanowią rozszerzenie modelu deklaratywnego o mechanizm obsługi błędów.
- Większość języków programowania funkcyjnego pozwala na wykorzystanie mechanizmu wyjątków. Używa się go w implementacji funkcji częściowych, których wynik nie jest określony dla wszystkich wartości argumentów.
- Scala. Wyjątki są zgłaszane tak jak w Javie.
- OCaml. O wyjątkach będzie mowa później. Na razie w sytuacjach wyjątkowych należy zgłaszać wyjątek Failure (patrz następny slajd) lub krócej: failwith "wyjaśnienie wyjątku".

Funkcje c.d.

Scala

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
def silnia(n:Int):Int =
  if (n==0) 1
  else if (n>0) n*silnia(n-1)
  else throw new Exception("ujemny argument")
// naciśnij ctrl-D , ten komentarz jest mój
// Exiting paste mode, now interpreting

silnia: (n: Int)Int
scala> silnia (4)
res0: Int = 24
scala> silnia (-4)
java.lang.Exception: ujemny argument
    at .silnia(<pastie>:10)
    ... 32 elided
```

OCaml

```
# let rec silnia n =
  if n=0 then 1
  else if n>0 then n*silnia(n-1)
  else raise (Failure "ujemny argument");;

val silnia : int -> int = <fun>

# silnia 4;;
- : int = 24

# silnia (-4);;
Exception: Failure "ujemny argument".
```

Funkcje c.d.

W językach funkcyjnych wszystkie funkcje są jednoargumentowe. Kilka wartości można przekazać do funkcji w postaci krotki. Jak wspomniano wcześniej, w Scali jest niestety inaczej (z powodu kompatybilności z językiem Java).

Scala

```
scala> (p:(Int,Int)) => p._1 + p._2
res0: ((Int, Int)) => Int = <function1> // jeden argument

scala> (x:Int, y:Int) => x+y
res1: (Int, Int) => Int = <function2> // dwa argumenty

// mimo to wywoływać można identycznie

scala> ((p:(Int,Int)) => p._1 + p._2)(4,5)
res2: Int = 9

scala> ((x:Int, y:Int) => x+y) (4,5)
res3: Int = 9

// ale uwaga!

scala> ((p:(Int,Int)) => p._1 + p._2)((4,5))
res4: Int = 9

scala> ((x:Int, y:Int) => x+y) ((4,5))
<console>:8: error: not enough arguments for method apply:
(v1: Int, v2: Int)Int in trait Function2.
Unspecified value parameter v2.
```

OCaml

```
# fun p -> fst p + snd p;;
- : int * int -> int = <fun>

# fun (x,y) -> x+y;;
- : int * int -> int = <fun>

# (fun p -> fst p + snd p) (4,5);;
- : int = 9

# (fun (x,y) -> x+y) (4,5);;
- : int = 9
```

Statyczne wiązanie zmiennych globalnych funkcji

Scala

```
scala> val p = 10  
p: Int = 10  
scala> def f (x:Int) = (x, p, x+p)  
f: (x: Int)(Int, Int, Int)  
scala> f(p)  
res3: (Int, Int, Int) = (10,10,20)  
scala> val p = 1000  
p: Int = 1000  
scala> f(p)  
res4: (Int, Int, Int) = (1000,10,1010)
```

OCaml

```
# let p = 10;;  
val p : int = 10  
# let f x = (x, p, x+p);;  
val f : int -> int * int * int = <fun>  
# f p;;  
- : int * int * int = (10, 10, 20)  
# let p = 1000;;  
val p : int = 1000  
# f p;;  
: int * int * int = (1000, 10, 1010)
```

Zmienne globalne funkcji są wiązane statycznie z wartościami ze środowiska w momencie definiowania funkcji. Wiązanie dynamiczne jest bardzo rzadko stosowane (np. w Lispie).

Wiązanie zmiennych globalnych funkcji

OCaml – wiązanie statyczne

```
# let a = 3
  in let f = fun x -> x + a
    and a = 5
    in a * f 2;;
- : int = 25

(* 5 * (2 + 3) => 25 *)
```

Lisp – wiązanie dynamiczne

```
➤ let a = 3
  in let f = proc (x) + (x,a)
    a = 5
    in *(a, f(2))
➔ 35

;; *(5, +(2, 5)) => 35
;; czyli 5 * (2 + 5) => 35
```

W OCamlu zmienne globalne funkcji są wiązane statycznie z wartościami ze środowiska w momencie **definiowania** funkcji.

W Lispie zmienne globalne funkcji są wiązane dynamicznie z wartościami ze środowiska w czasie **wykonania** funkcji.

Polimorfizm parametryczny

Scala

```
scala> def id[A](x:A) = x
id: [A](x: A)A
scala> id (5)
res5: Int = 5
scala> id (3+4, "siedem")
res6: (Int, String) = (7,siedem)
scala> :paste
def last[A](xs:List[A]):A =
if (xs == Nil) throw new Exception("pusta lista")
else (xs.reverse).head
// naciśnij ctrl-D , ten komentarz jest mój
last: [A](xs: List[A])A
scala> last(List(1,3,4,2))
res7: Int = 2
scala> last(Nil)
java.lang.Exception: pusta lista
    at .last(<pastie>:8)
    ... 32 elided
```

OCaml

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 5;;
- : int = 5
# id (3+4, "siedem");;
-   : int * string = (7, "siedem")
# let last xs =
    if xs=[] then failwith "pusta lista"
    else List.hd( List.rev xs);;
val last : 'a list -> 'a = <fun>
# last [1;3;4;2];;
- : int = 2
# last [];;
Exception: Failure "pusta lista".
```

Kod pośredni (bajtowy)

Kod bajtowy (ang. byte code) to nazwa reprezentacji (zwykle niezależnej od platformy) kodu programu będącego kodem pośrednim między kodem źródłowym a kodem maszynowym.

OCaml stosuje dynamiczne ładowanie modułów z kodem bajtowym zarówno w czasie wykonywania programu, jak i pracy interakcyjnej w cyklu REPL. Podczas pracy w cyklu REPL moduł Foo z kodem bajtowym można załadować za pomocą dyrektywy `#load`:

```
# #load "Foo.cma";;
```

Kod źródłowy z pliku "foo.ml" można załadować i wykonać za pomocą dyrektywy `#use`:

```
# #use "foo.ml";;
```

Dodatek - przypomnienie

Złożoność obliczeniowa. Podstawowe pojęcia.

Rząd wielkości funkcji, notacja asymptotyczna, równania rekurencyjne, najważniejsze klasy asymptotycznej złożoności obliczeniowej.

Podstawowe pojęcia

Złożoność obliczeniową algorytmu definiuje się jako ilość zasobów komputerowych (np. czas działania lub ilość zajmowanej pamięci), potrzebnych do jego wykonania i przedstawia w postaci funkcji *rozmiaru danych wejściowych* $n=|d|$.

Złożoność czasowa $T(n)$ powinna być własnością samego algorytmu, niezależnie od jego implementacji. Osiąga się to, wyróżniając w algorytmie *operacje charakterystyczne (dominujące)*, i definiując $T(n)$ jako ilość operacji charakterystycznych, wykonywanych przez algorytm dla danych o rozmiarze n ; czyli za *jednostkę złożoności czasowej* przyjmuje się wykonanie jednej operacji charakterystycznej. Najczęściej rozważa się *pesymistyczną złożoność czasową* $W(n)$, czyli najdłuższy czas działania dla każdego danych określonego rozmiaru, czasem bada się też *złożoność oczekiwaną* $A(n)$.

Dla sekwencji operacji (np. wykonywanych na stosie czy kolejce) bardzo przydatne jest pojęcie *złożoności amortyzowanej*.

Przykład

1 **PRIME**(n) { $n > 0$ }

2 $p \leftarrow 2$

3 $b \leftarrow \text{true}$

4 **while** $(p * p \leq n) \wedge b$ **do**

$\{ b \Leftrightarrow \forall q. (2 \leq q \leq p-1) \Rightarrow n \bmod q \neq 0 \}$

5 **if** $n \bmod p = 0$

6 **then** $b \leftarrow \text{false}$

7 $p \leftarrow p+1$

$\{ b \Leftrightarrow \forall q. (2 \leq q \leq n-1) \Rightarrow n \bmod q \neq 0 \}$

8 **return** b

.

dokładnie 1 operacja

dokładnie 1 operacja

co najwyżej $3 \lfloor \sqrt{n} \rfloor$ operacji

co najwyżej $2(\lfloor \sqrt{n} \rfloor - 1)$ operacji

co najwyżej 1 operacja

co najwyżej $2(\lfloor \sqrt{n} \rfloor - 1)$ operacji

dokładnie 1 operacja

Razem: co najwyżej $7 \lfloor \sqrt{n} \rfloor$ operacji

Gdy n jest liczbą pierwszą wykonuje się dokładnie $7 \lfloor \sqrt{n} \rfloor - 1$ operacji.

Gdy n jest kwadratem liczby pierwszej wykonuje się dokładnie $7 \lfloor \sqrt{n} \rfloor$ operacji.

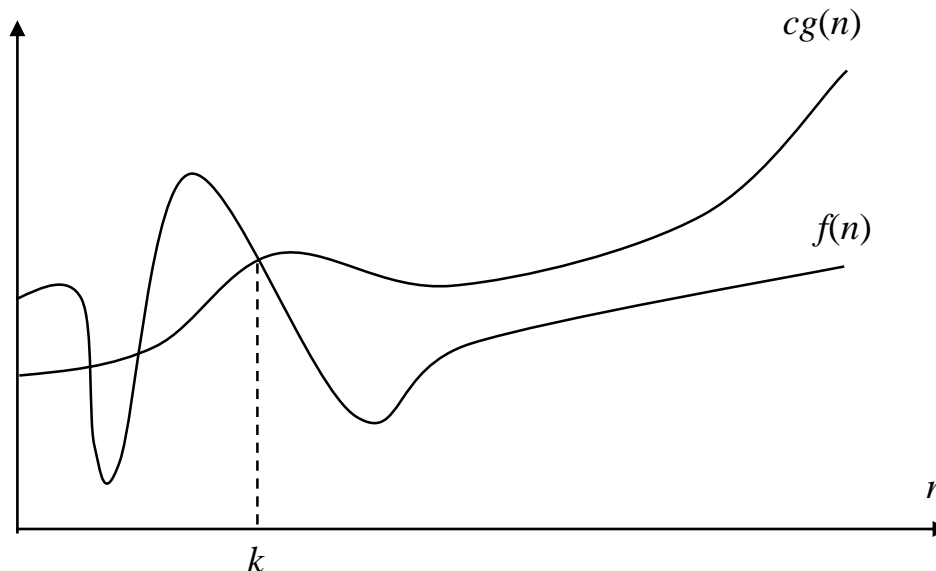
Gdy n jest liczbą parzystą wykonuje się dokładnie 14 operacji.

Rząd wielkości funkcji i notacja asymptotyczna

Niech $f, g: N \rightarrow R_+ \cup \{0\}$.

$O(g(n)) = \{f(n) \mid \text{istnieją stała rzeczywista } c > 0 \text{ i stała naturalna } k \text{ takie, że } 0 \leq f(n) \leq cg(n) \text{ zachodzi dla każdego } n \geq k\}$

Mówimy, że f jest co najwyżej rzędu g (lub f jest asymptotycznie zdominowana przez g), co zapisujemy $f(n) = O(g(n))$, gdy f jest elementem zbioru $O(g(n))$. Np. $n^2 + 2n = O(n^2)$, ponieważ $n^2 + 2n \leq 3n^2$ dla każdego $n \geq 0$. Taka notacja jest używana z powodów historycznych. Matematycznie poprawne jest $f(n) \in O(g(n))$.

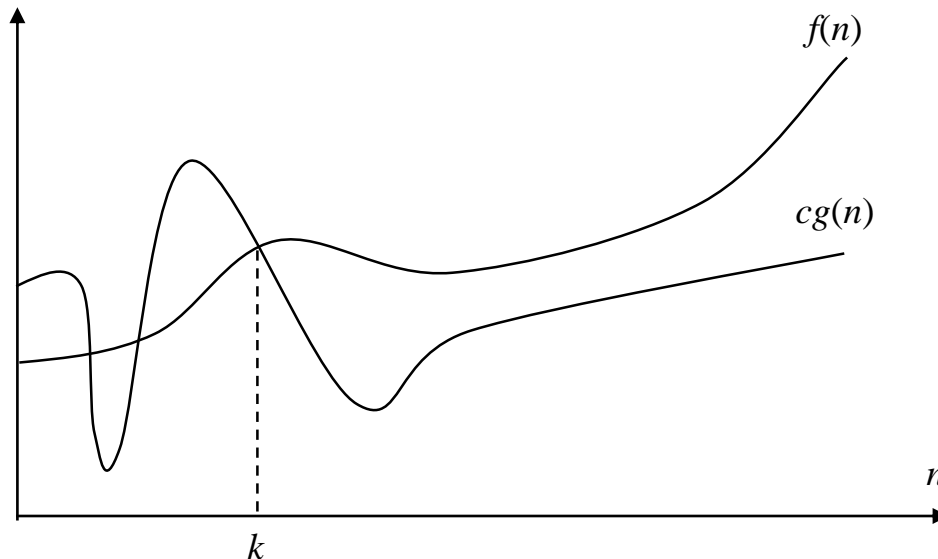


Rząd wielkości funkcji i notacja asymptotyczna

Niech $f, g: \mathbb{N} \rightarrow \mathbb{R}_+ \cup \{0\}$.

$\Omega(g(n)) = \{f(n) \mid \text{istnieją stała rzeczywista } c > 0 \text{ i stała naturalna } k \text{ takie, że } 0 \leq cg(n) \leq f(n) \text{ zachodzi dla każdego } n \geq k\}$

Mówimy, że f jest co najmniej rzędu g , co zapisujemy $f(n) = \Omega(g(n))$, gdy f jest elementem zbioru $\Omega(g(n))$. Np. $n^2 + 2n = \Omega(n^2)$, ponieważ $n^2 \leq n^2 + 2n$ dla każdego $n \geq 0$.

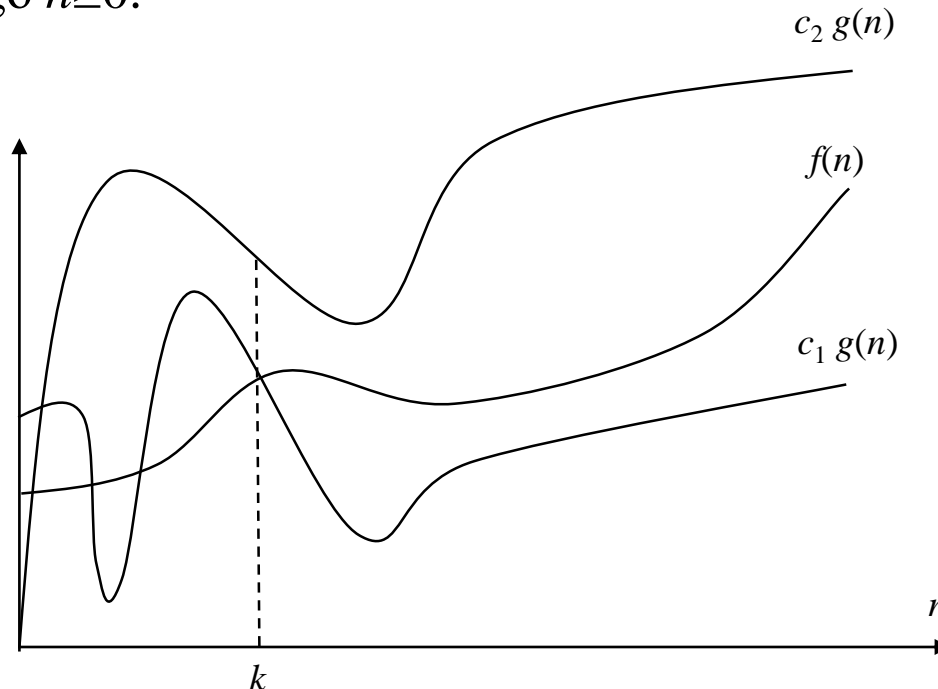


Rząd wielkości funkcji i notacja asymptotyczna

Niech $f, g: N \rightarrow R_+ \cup \{0\}$.

$\Theta(g(n)) = \{f(n) \mid \text{istnieją stałe rzeczywiste } c_1 > 0 \text{ i } c_2 > 0, \text{ oraz stała naturalna } k \text{ takie, że } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ zachodzi dla każdego } n \geq k\}$

Mówimy, że f jest dokładnie rzędu g , co zapisujemy $f(n) = \Theta(g(n))$, gdy f jest elementem zbioru $\Theta(g(n))$. Np. $n^2 + 2n = \Theta(n^2)$, ponieważ $n^2 \leq n^2 + 2n \leq 3n^2$ dla każdego $n \geq 0$.



Porównywanie rzędów funkcji

Niech $f, g, h: N \rightarrow R_+ \cup \{0\}$.

- Przechodniość:

Jeśli $f(n) = O(g(n))$ i $g(n) = O(h(n))$ to $f(n) = O(h(n))$

Jeśli $f(n) = \Omega(g(n))$ i $g(n) = \Omega(h(n))$ to $f(n) = \Omega(h(n))$

Jeśli $f(n) = \Theta(g(n))$ i $g(n) = \Theta(h(n))$ to $f(n) = \Theta(h(n))$

- Zwrotność: $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, $f(n) = \Theta(f(n))$

- Symetria: $f(n) = \Theta(g(n))$ wtw, gdy $g(n) = \Theta(f(n))$

- Symetria transpozycyjna: $f(n) = O(g(n))$ wtw, gdy $g(n) = \Omega(f(n))$

- Własności domknięcia:

Jeśli $f(n) = O(g(n))$, to $mf(n) = O(g(n))$ dla dowolnego $m \in R_+$

Jeśli $f(n) = O(g(n))$ i $h(n) = O(g(n))$ to $f(n) + h(n) = O(g(n))$

- $\log_a n = O(\log_b n)$ dla każdej liczby rzeczywistej $a, b > 1$

Równania rekurencyjne

Często złożoność obliczeniową algorytmu daje się łatwo wyrazić za pomocą *równania rekurencyjnego*. W metodzie iteracyjnej rozwiązywania równań rekurencyjnych rekurencja jest przekształcana w sumę, której wartość jest szacowana.

```
1 for i ← 1 to n do  
2   I = O(1)
```

$$\begin{aligned}T(0) &= 0 \\T(n) &= T(n-1) + c \\T(n) &= T(n-1) + c \\&= T(n-2) + c + c \\&\dots \\&= cn \\&\text{czyli } T(n) = \Theta(n)\end{aligned}$$

```
1 for i ← 1 to m do  
2   for j ← 1 to n do  
3     I = O(1)
```

$$\begin{aligned}T(0) &= 0 \\T(m) &= T(m-1) + cn \\T(m) &= T(m-1) + cn \\&= T(m-2) + cn + cn \\&\dots \\&= cmn \\&\text{czyli } T(m) = \Theta(mn)\end{aligned}$$

Równania rekurencyjne

Wyeliminowanie jednego elementu danych wymaga analizy wszystkich elementów.

```
1 for i ← 1 to n do  
2   for j ← 1 to i do  
3     I=O(1)
```

$$T(0) = 0$$

$$T(n) = T(n-1) + cn$$

$$T(n) = T(n-1) + cn$$

$$= T(n-2) + c(n-1) + cn$$

...

$$= c + c2 + \dots + c(n-2) + c(n-1) + cn$$

$$= cn(n+1)/2$$

$$\text{czyli } T(n) = \Theta(n^2 + n)$$

$$\text{lub } T(n) = \Theta(n^2)$$

Rozmiar problemu jest zmniejszany o połowę stałym kosztem.

```
1 r ← n;  
2 while r > 1 do  
3   I=O(1);  
4   r ← ⌊r/2⌋;
```

$$T(1) = 0$$

$$T(n) = T(\lfloor n/2 \rfloor) + c \text{ dla } n > 1$$

Niech $n = 2^k$. Wtedy

$$T(2^k) = T(2^{k-1}) + c$$

$$= T(2^{k-2}) + c + c$$

...

$$= T(2^0) + kc = c \lg n$$

$$\text{czyli } T(n) = \Theta(\lg n)$$

Równania rekurencyjne

W celu zmniejszenia rozmiaru zadania o połowę trzeba przejrzeć wszystkie elementy.

```
1  $r \leftarrow n$ ;  
2 while  $r > 1$  do  
3   for  $i \leftarrow 1$  to  $r$  do  
4      $I = O(1)$ ;  
5    $r \leftarrow \lfloor r/2 \rfloor$ ;  
       $T(1) = 0$   
       $T(n) = T(\lfloor n/2 \rfloor) + cn$  dla  $n > 1$ 
```

Niech $n = 2^k$. Wtedy

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + c2^k \\ &= T(2^{k-2}) + c2^{k-1} + c2^k \\ &\dots \\ &= T(2^0) + c2^0 + c2^1 + \dots + c2^{k-1} + c2^k \\ &= c2(2^k - 1)/(2 - 1) = 2cn - 2c \end{aligned}$$

czyli $T(n) = \Theta(n)$
Zdzisław Spławski

Redukcja zadania do dwóch podzadań rozmiaru $n/2$ kosztem liniowej liczby operacji (algorytmy „divide-and-conquer”).

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + cn \text{ dla } n > 1 \end{aligned}$$

Niech $n = 2^k$. Wtedy

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + c2^k \\ &= 2(2T(2^{k-2}) + c2^{k-1}) + c2^k \\ &= 2^2T(2^{k-2}) + c2^k + c2^k \\ &\dots \\ &= 2^k T(2^0) + kc2^k \\ &= 0 + cn \lg n \end{aligned}$$

czyli $T(n) = \Theta(n \lg n)$

Najważniejsze klasy asymptotycznej złożoności obliczeniowej

1. $f = O(1)$ – złożoność stała; koszt algorytmu nie zależy od rozmiaru danych
2. $f = O(\lg n)$ – złożoność logarytmiczna (zadanie rozmiaru n zostaje sprowadzone do zadania rozmiaru $n/2$ + stała liczba działań)
3. $f = O(n)$ – złożoność liniowa (wykonywana jest stała liczba działań dla każdego z n elementów danych)
4. $f = O(n \lg n)$ – złożoność $n \log n$; liniowo-logarytmiczna (zadanie rozmiaru n zostaje sprowadzone do dwóch zadań rozmiaru $n/2$ kosztem liniowej liczby operacji)
5. $f = O(n^2)$ – złożoność kwadratowa (wykonywana jest stała liczba działań dla każdej pary danych, np. podwójna pętla)
6. $f = O(2^n)$ – złożoność wykładnicza (wykonywana jest stała liczba działań dla każdego podzbioru danych)
7. $f = O(n!)$ – złożoność $n!$ (wykonywana jest stała liczba działań dla każdej permutacji danych)

Twierdzenie. $O(1) \subset O(\lg n) \subset O(n) \subset O(n \lg n) \subset O(n^2) \subset O(2^n) \subset O(n!)$

Ograniczenia oszacowania asymptotycznego

Oszacowanie asymptotyczne funkcji kosztów pozwala porównywać algorytmy dla „dostatecznie dużych” rozmiarów danych. Dla mniejszych rozmiarów trzeba przeprowadzić dokładniejszą analizę.

Założmy, że algorytmy F i G mają następujące funkcje kosztów: $f(n)=O(n^2)$, $g(n)=O(n)$. Oczywiście $O(g(n)) \subset O(f(n))$, ale dla $f(n)=n^2-n+550$, $g(n)=59n+50$ w przedziale $10 < n < 50$ algorytm F wykonuje się szybciej niż algorytm G .

