

Programowanie funkcyjne

dr inż. Zdzisław Spławski

Katedra Inżynierii Oprogramowania

Wydział Informatyki i Zarządzania

Politechniki Wrocławskiej

&

Instytut Informatyki Uniwersytetu Wrocławskiego

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

Wybrana literatura

- E.Chailloux, P.Manoury, B.Pagano. *Développement d'Applications avec Objective Caml*. O'Reilly, 2000 <http://caml.inria.fr/pub/docs/oreilly-book/>
- X.Leroy. *The Objective Caml System. Documentation and user's manual*. INRIA <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>
- <http://ocaml.org/>
- <http://www.ffconsultancy.com/> F#
- J.B.Smith. *Practical OCaml*. Apress 2006
- J.D.Harrop. *Objective Caml for Scientists*. Flying Frog Consultancy Ltd. 2005
- G.Hutton. *Programming in Haskell*. Cambridge University Press 2007
- P.Hudak. *The Haskell School of Expression*. Cambridge University Press 2007
- <https://www.haskell.org/documentation>
- R.K.Dybvig. *The Scheme Programming Language*, fourth edition. The MIT Press 2009
<http://www.scheme.com/tspl4/>
- M.Felleisen, R.B.Findler, M.Flatt, S.Krishnamurti. *Projektowanie oprogramowania. Wstęp do programowania i techniki komputerowej*. Helion 2003 <http://www.htdp.org/>
- D.F.Friedman, M.Wand. *Essentials of Programming Languages*, third edition. The MIT Press 2008 <http://www.eopl3.com/>
- H.Abelson, G.J.Sussman, J.Sussman. *Struktura i interpretacja programów komputerowych*. WNT 2002 <http://mitpress.mit.edu/sicp/>
- J.Backfield. *Programowanie funkcyjne. Krok po kroku*. Helion 2015
<http://helion.pl/ksiazki/programowanie-funkcyjne-krok-po-kroku-joshua-backfield,pfukpk.htm>

Wykład 1

Rola abstrakcji w programowaniu.

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 na przykładach

Typy bazowe i typy strukturalne: listy i krotki

Literały funkcyjne i definicje funkcji

Postać zwinięta i rozwinięta funkcji

Operatory infiksowe

Wartościowanie gorliwe w języku OCaml

Reguły wartościowania jako reguły dedukcji

Abstrakcja i reprezentacja

- Tranzystory abstrahują od reguł elektrofizyki. Architektura mikroprocesorów abstrahuje od szczegółów budowy tranzystorów. Systemy operacyjne abstrahują od szczegółów architektury mikroprocesorów. Biblioteki abstrahują od szczegółów systemów operacyjnych. Aplikacje abstrahują od szczegółów bibliotek ...
- Abstrakcja jest stale obecna w działalności programisty. Specyfikacja systemu informatycznego abstrahuje od wielu szczegółów konkretnego wycinka rzeczywistości.
- Sposób reprezentacji problemu lub wycinka rzeczywistości jest kluczowy dla zrozumienia problemu i jego informatycznego rozwiązania.
- Czy używany język programowania pozwala na wyrażanie odpowiednich poziomów abstrakcji?
- Krótkie programy na wysokim poziomie abstrakcji są na ogół łatwiejsze do napisania i zrozumienia niż długie programy z wieloma szczegółami, co jest nie do uniknięcia na niskim poziomie abstrakcji.

Paradygmat

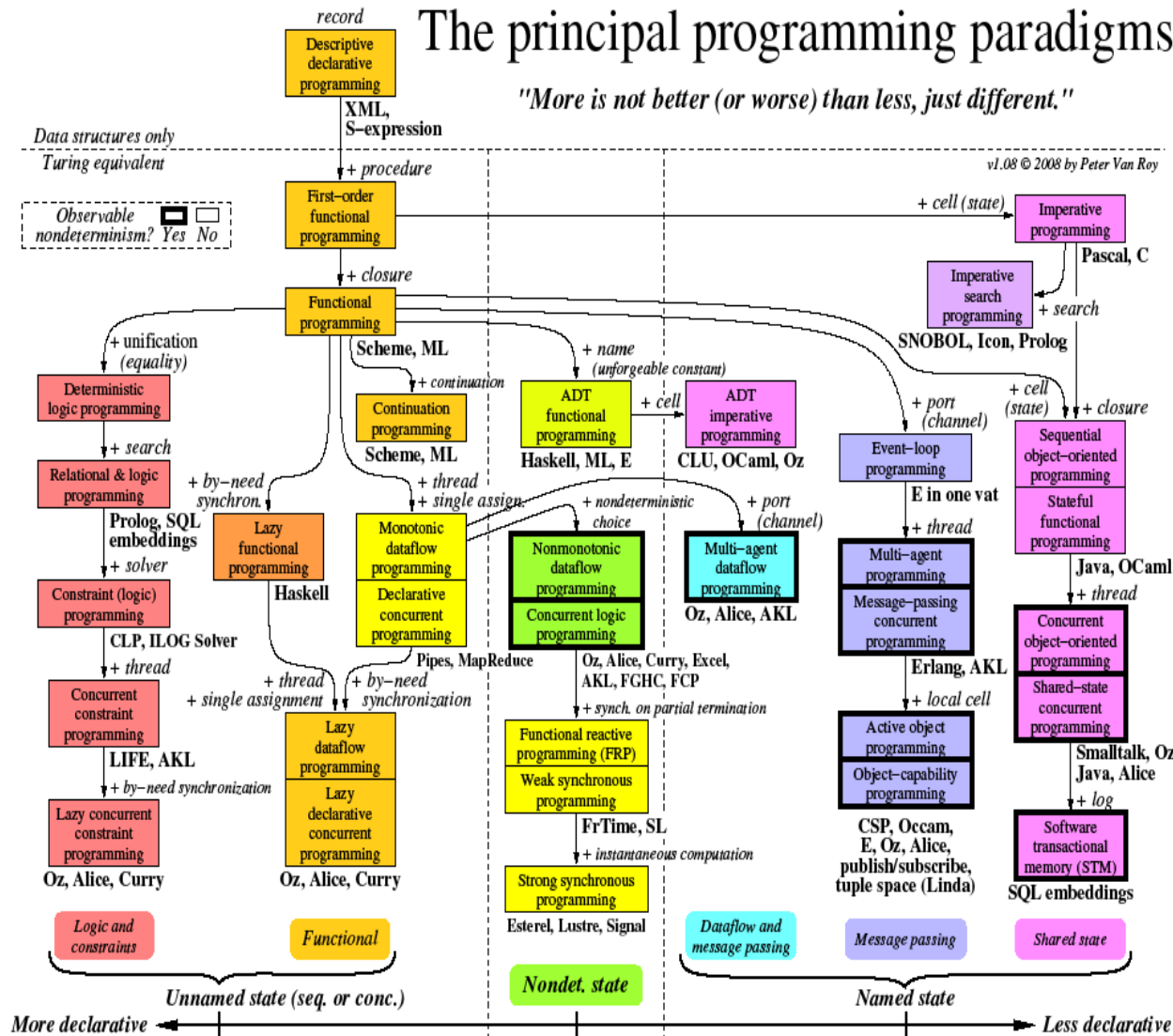
Paradygmat <łac. *paradigma* z gr. *παράδειγμα* = przykład, wzór > **to przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie itp.; wzorzec, model.**

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

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 „jedynego 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

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. πολυς = 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 można 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 i 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 parametry, 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 wyrażen, 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, mocna	statyczne	gorliwa	tak
Haskell	statyczna, mocna	statyczne	leniwa	nie

Co to jest OCaml?

OCaml (Objective Caml) jest językiem programowania funkcyjnego:

- z silną typizacją (statyczną),
- z parametrycznym polimorfizmem,
- z automatyczną inferencją typów,
- z mechanizmem wyjątków,
- z automatycznym odśmiecaniem pamięci,
- 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.

Garść informacji

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>). 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.

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 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.

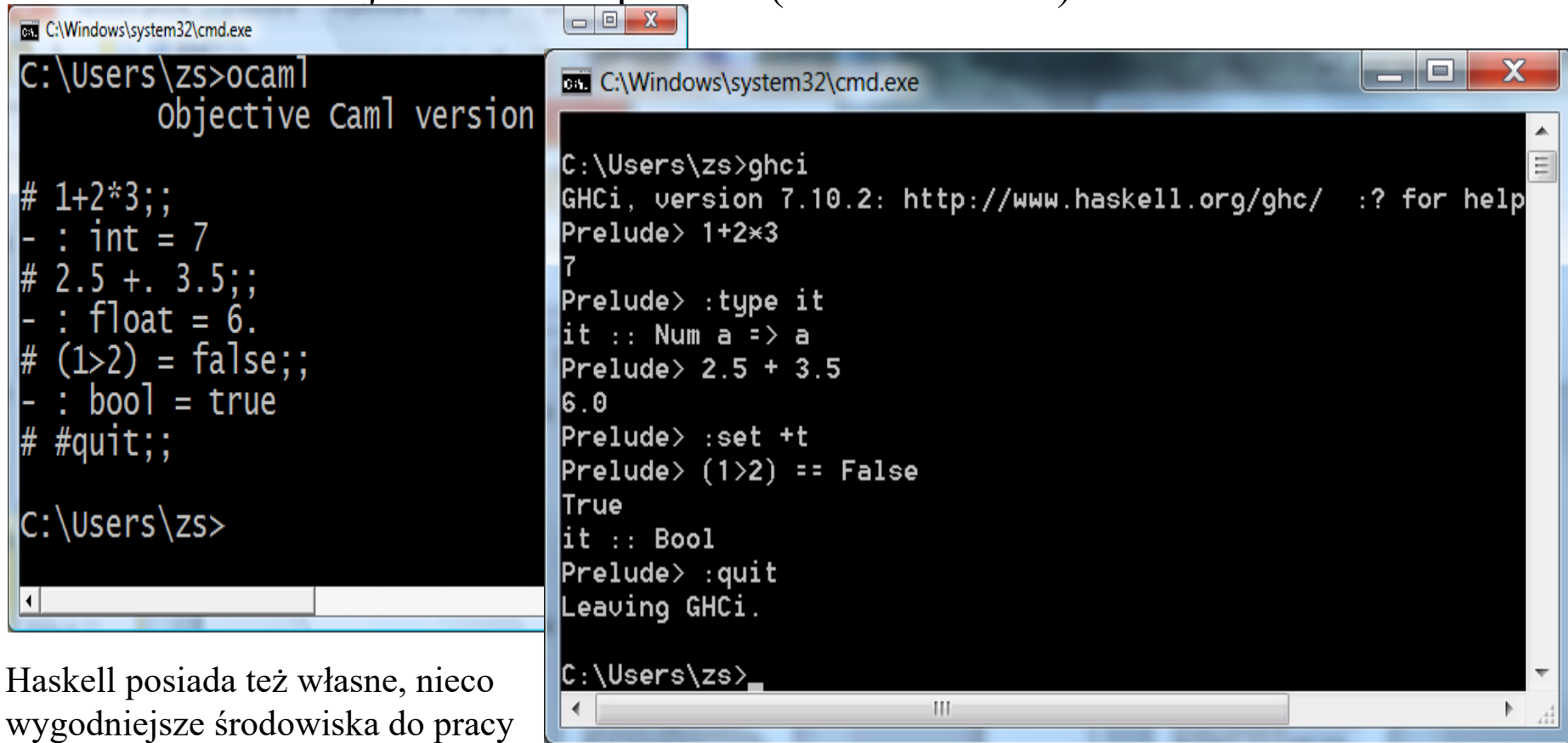
System typów w języku OCaml

- 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).



```
C:\Windows\system32\cmd.exe
C:\Users\zs>ocaml
objective Caml version

# 1+2*3;;
- : int = 7
# 2.5 +. 3.5;;
- : float = 6.
# (1>2) = false;;
- : bool = true
# #quit;;

C:\Users\zs>
```

```
C:\Windows\system32\cmd.exe
C:\Users\zs>ghci
GHCi, version 7.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> 1+2*3
7
Prelude> :type it
it :: Num a => a
Prelude> 2.5 + 3.5
6.0
Prelude> :set +t
Prelude> (1>2) == False
True
it :: Bool
Prelude> :quit
Leaving GHCi.

C:\Users\zs>
```

Haskell posiada też własne, nieco wygodniejsze środowiska do pracy interakcyjnej: WinGHCi.

Przegląd podstawowych konstrukcji języka

Składniowo program w języku Ocaml jest ciągiem fraz (wyrażeń i definicji), które mogą być bezpośrednio wykonane. Koniec frazy w pracy intrykcyjnej jest oznaczany przez podwójny średnik (;). Istnieją definicje wartości (`let`), wyjątków (`exception`) oraz typów (`type`).

```
# 1+2*3;;
- : int = 7
# 2.5 +. 3.5;;
-: float = 6.
# 2.0 + 3.5;;
Characters 0-3:
  2.0 + 3.5;;
  ^^^
Error: This expression has type float but an expression was expected of type
      int
# not(1 > 2);;
- : bool = true
# 'a';;
- : char = 'a'
# let x = 3+2;;      (* wiązanie identyfikatora z wartością *)
val x : int = 5
# let s1 = "ciekawe" and s2 = "To ";;
val s1 : string = "ciekawe"
val s2 : string = "To "
```

Przegląd podstawowych konstrukcji języka

```
# let s1 = s2 and s2 = s1;;  
                                (* wartościowane są najpierw wszystkie prawe strony *)  
val s1 : string = "To "  
val s2 : string = "ciekawe"  
# let l1 = s1 :: s2 :: [];;  
val l1 : string list = ["To "; "ciekawe"]  
# let p = (x+1, s1 ^ s2);;  
val p : int * string = (6, "To ciekawe")  
# snd p;;  
-: string = "To ciekawe"
```

Uwaga! W przeciwieństwie do języków imperatywnych jak Pascal, C++ czy Java, w „czystych” językach funkcyjnych nie ma instrukcji przypisania. Identyfikator jest związany z wartością tak jak w matematyce), a nie z miejscem w pamięci operacyjnej! Poniższa deklaracja nie powoduje błędu typu.

```
# let p = -3.14;;  
val p : float = -3.14
```

Definicje wartości

let *identyfikator* = *wyrażenie*;;

```
# let a = 2 + 3  let b = "A" ^ "la";;  
val a : int = 5  
val b : string = "Ala"
```

Równoczesne definicje wartości

let *ident1* = *wyrl* **and** *ident1* = *wyrl* ... **and** *ident1* = *wyrl*;;

```
# let a = b and b = a;;          (* najpierw są wartościowane prawe strony *)  
val a : string = "Ala"         (* wszystkich deklaracji *)  
val b : int = 5
```

Definicje lokalne (są wyrażeniami)

```
# (let x = b*b in x+x) + 1;;    (* zwiększają czytelność i/lub efektywność *)  
- : int = 51
```

Równoczesne definicje lokalne

```
# (let x=b mod 2 and y=b-3 in y+x) + 1;;  
- : int = 4
```

Funkcje

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

Funkcje rekurencyjne

```
# let rec silnia n = if n=0 then 1 else n*silnia(n-1);;  
val silnia : int -> int = <fun>  
# silnia 4;;  
- : int = 24
```

Funkcje i polimorfizm

```
# 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"
```

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`.

`^` konkatencja napisów

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

typ `float`

`+. -. *. /. **`

`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)

Listy

```
# [];;          (* lista pusta jest polimorficzna, tzn.*)
- : 'a list = [] (* jest listą elementów dowolnego typu *)
# let l1 = 1::2::3::[];; (* pełna notacja *)
val l1 : int list = [1; 2; 3]
# let l2 = [1;2;3];;      (* notacja skrócona *)
val l2 : int list = [1; 2; 3]
# l1 = l2;;              (* listy można porównywać *)
- : bool = true
# [1;2]@[2;3];;          (* konkatencja list; ma złożoność liniową *)
- : int list = [1; 2; 2; 3] (* względem pierwszego argumentu *)
# List.hd [1;2;3];;      (* patrz moduł List *)
- : int = 1
# List.tl [1;2;3];;
- : int list = [2; 3]
# List.rev [1;2;3];;
-: int list = [3; 2; 1]
# List.length [5;6;7];;  (* ma złożoność liniową *)
- : int = 3
# [[]];;
-: 'a list list = [[]]
```

Pary i krotki

Infiksowym konstruktorem wartości tego typu jest przecinek (,), a konstruktorem typu jest gwiazdka (*).

```
# (8, "osiem");;      (* nawiasy tylko zwiększają czytelność *)
- : int * string = (8, "osiem")
# (1,1.0,"jeden");;      (* to jest krotka *)
- : int * float * string = (1, 1., "jeden")
# (1,(1.0,"jeden"));;      (* to jest para *)
- : int * (float * string) = (1, (1., "jeden"))
# fst (8, "osiem");;
- : int = 8
# snd;;
- : 'a * 'b -> 'b = <fun>
```

Warunkowa struktura kontrolna

Wyrażenie `if w1 then w2 else w3` zwraca wartość `w2` jeśli `w1` daje w wyniku wartościowania `true` i wartość `w3` jeśli `w1` daje w wyniku wartościowania `false`.

```
# ( if 2=3 then 4 else 5) + 5;;
- : int = 10
```


Literały funkcyjne (funkcje anonimowe)

function *arg* -> *wyr*

```
# (function x -> x*x) 5;;  
- : int = 25
```

wyr w wyrażeniu funkcyjnym może być dowolnym wyrażeniem
(także wyrażeniem funkcyjnym)

```
# let f = function y -> function x -> x*x+y;;  
val f : int -> int -> int = <fun>  
# f 2 5;;  
- : int = 27  
# let fdwa = f 2;;  
val fdwa : int -> int = <fun>  
# fdwa 5;;  
- : int = 27  
# let f5 = function z -> f z 5;;  
val f5 : int -> int = <fun>  
# f5 2;;  
- : int = 27
```

Strzałka wiąże w prawo `int -> (int -> int)`

Aplikacja wiąże w lewo `(f 2) 5`

fdwa jest funkcją wyspecjalizowaną

f5 jest funkcją wyspecjalizowaną

Lukier syntaktyczny dla funkcji

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
```

Wszystkie funkcje są jednoargumentowe

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

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

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

postać zwinięta

postać rozwinięta

Niepoprawne funkcje rekurencyjne

Rozważ poniższą „definicję” indukcyjną:

$$\begin{array}{ll} \text{nonsens}(n) = 1 & \text{dla } n=0 \\ \text{nonsens}(n) = n * \text{nonsens}(n+1) & \text{w przeciwnym przypadku} \end{array}$$

i odpowiadającą tej definicji funkcję rekurencyjną:

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

Matematyk powie, że funkcja jest „źle zdefiniowana” dla liczb dodatnich, a informatyk powie, że funkcja „zapętla się”. Ponieważ każde wywołanie rekurencyjne pobiera pamięć ze stosu, to po wyczerpaniu pamięci program kończy się z odpowiednim komunikatem.

Wyjątki

Większość języków funkcyjnych zawiera mechanizm wyjątków. Dokładnie będziemy mówili o nich później. Na razie w sytuacjach wyjątkowych należy zgłaszać wyjątek Failure (patrz następny slajd) lub krócej: failwith "wyjaśnienie wyjątku".

```
# 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?).
```

```
# let rec silnia n =
  if n=0 then 1
  else if n>0 then n*silnia(n-1)
  else failwith "ujemny argument";;
val silnia : int -> int = <fun>
# silnia 4;;
- : int = 24
# silnia (-4);;
Exception: Failure "ujemny argument".
```

Operatory infiksowe

Każdy operator infiksowy może być zamieniony na funkcję dwuargumentową w postaci rozwiniętej przez umieszczenie go w nawiasach: *(op)*

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

Można definiować swoje operatory infiksowe i prefiksowe

```
# let (++) c1 c2 = (fst c1) + (fst c2), (snd c1) + (snd c2);;  
val ( ++ ) : int * int -> int * int -> int * int = <fun>  
# let c = (2,3) in c ++ c;;  
- : int * int = (4, 6)
```

ident ::= (litera | _) { litera | 0 ... 9 | _ | ' }

*operator ::= ! | \$ | % | & | * | + | - | . | / | : | < | = | > | ? | @ | ^ | | | ~*

*op-infiksowy ::= (= | < | > | @ | ^ | | | & | + | - | * | / | \$ | %) { operator }*

op-prefiksowy ::= (! | ? | ~) { operator }

Połączenie zbiorów identyfikatorów alfanumerycznych i symbolicznych spowodowałoby konieczność oddzielania identyfikatorów spacjami, np. wyrażenie $x+1$ byłoby traktowane jako identyfikator, a nie jako $x + 1$.

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 **v₁**, 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 ”.

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

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

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

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

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

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

Przykłady wartościowań

$(3+4) * (5 + 6) \Rightarrow (3 + 4) * 11$
 $\Rightarrow 7 * 11$
 $\Rightarrow 77$

```
let sqr n = n*n;;
```

$\text{sqr} (\text{sqr } 2) \Rightarrow \text{sqr } (2*2)$
 $\Rightarrow \text{sqr } 4$
 $\Rightarrow 4*4$
 $\Rightarrow 16$

Gorliwe wartościowanie rekursji

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

```
silnia 4  ⇒ if 4=0 then 1 else 4*silnia (4-1)
          ⇒ 4*silnia 3
          ⇒ 4*(if 3=0 then 1 else 3*silnia (3-1))
          ⇒ 4*(3* silnia 2)
          ⇒ 4*(3*(if 2=0 then 1 else 2*silnia (2-1)))
          ⇒ 4*(3*(2*silnia 1))
          ⇒ 4*(3*(2*(if 1=0 then 1 else 1*silnia (1-1))))
          ⇒ 4*(3*(2*(1*silnia 0)))
          ⇒ 4*(3*(2*(1*(if 0=0 then 1 else 0*silnia (0-1)))))
          ⇒ 4*(3*(2*(1*1)))
          ⇒ 4*(3*(2*1))
          ⇒ 4*(3*2)
          ⇒ 4*6
          ⇒ 24
```

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";;
```

Definicja języka programowania (wymagania)

Opisy języków programowania powinny z jednej strony ułatwiać programiście pisanie programów (*generowanie* napisów należących do języka), z drugiej zaś umożliwiać automatyczne sprawdzanie (*rozpoznawanie*), czy program istotnie należy do języka.

Definicja języka programowania zawiera:

- opis *składni* (*syntaktyki*), czyli definicję zbioru napisów, będących poprawnymi programami;
- opis *semantyki*, czyli znaczenia programów;
- ewentualnie *system wnioskowania*, służący do dowodzenia poprawności programów, tj. ich zgodności ze specyfikacją.

Podane wyżej reguły wartościowania zadają semantykę dla poznanego przez nas fragmentu języka OCaml.

Zadania kontrolne

1. Podaj typy poniższych funkcji (zrekonstruuj je samodzielnie, a potem wykorzystaj OCamla do sprawdzenia):
 - a) **function** $(x, y) \rightarrow x + y$
 - b) **function** $(x, y, z) \rightarrow \text{if } x \text{ then } y \text{ else } z$
 - c) **function** $(x, (y, z)) \rightarrow ((x, y), z)$
2. Zdefiniuj polimorficzną funkcję *flatten*, która dla argumentu będącego listą list tworzy listę złożoną z elementów wszystkich podlist z zachowaniem ich kolejności, np. *flatten* `[[5;6];[1;2;3]]` zwraca `[5; 6; 1; 2; 3]`. Jaki jest typ funkcji *flatten*?
3. Zdefiniuj polimorficzną funkcję *count* obliczającą ile razy dany obiekt występuje w danej liście, np. *count* `('a', ['a'; 'l'; 'a'])` zwraca 2. Jaki jest typ tej funkcji?
4. Zdefiniuj polimorficzną funkcję *duplicate* powtarzającą dany obiekt określoną liczbę razy i zwracającą wynik w postaci listy, np. *duplicate* `("la", 3) → ["la"; "la"; "la"]`. Jaki jest typ tej funkcji?
5. Zdefiniuj funkcję *sqr_list* : *int list* → *int list* podnoszącą do kwadratu wszystkie elementy danej listy liczb całkowitych, np. *sqr_list* `[1; 2; -3] → [1; 4; 9]`.
6. Zdefiniuj polimorficzną funkcję *palindrome* : *'a list* → *bool* sprawdzającą, czy dana lista jest palindromem, tj. równa się sobie samej przy odwróconej kolejności elementów.
7. Napisz dowolną funkcję typu *'a* → *'b*.