

Języki i Paradygmaty Programowania

Przemysław Kobylański
Katedra Informatyki (W11/K2)
Politechnika Wrocławska

Na podstawie: Peter Van Roy. *Programowanie: Koncepcje, Techniki i Modele*
R. Kent Dybvig. *The Scheme Programming Language*
Graham Hutton. *Programming in Haskell*
Joe Armstrong. *Programming Erlang. Software for a Concurrent World*
William Clocksin, Christopher Mellish. *Prolog. Programowanie*



Politechnika Wrocławska

Plan wykładu

1. Koncepcje i paradygmaty (2w)
2. Przegląd paradygmatów na przykładzie języka Oz (4w)
3. Elementy języka Scheme (2w)
4. Elementy języka Haskell (2w)
5. Elementy języka Erlang (2w)
6. Elementy języka Prolog (2w)
7. Kolokwium

Literatura

- Peter Van Roy. **Programowanie: Koncepcje, Techniki i Modele**. Helion, 2005.
- R. Kent Dybvig. **The Scheme Programming Language**. The MIT Press, 2009.
- Graham Hutton. **Programming in Haskell**. Cambridge University Press, 2007.
- Joe Armstrong. **Programming Erlang. Software for a Concurrent World**. Pragmatic Bookshelf, 2013.
- William Clocksin, Christopher Mellish. **Prolog. Programowanie**. Helion, 2003.

Narzędzia programistyczne i materiały dostępne w sieci

- [Mozart/Oz](#)
- [MIT/GNU Scheme](#)
- [Haskell](#)
- [Erlang](#)
- [Prolog](#)

Koncepcje i paradygmaty

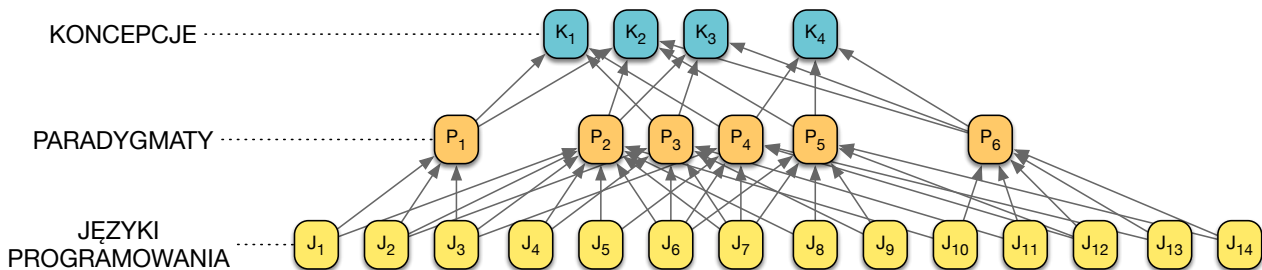
- języki — paradygmaty — koncepcje
- przegląd podstawowych koncepcji
- przegląd podstawowych paradygmatów

Języki — paradygmaty — koncepcje
Paradygmat

paradygmat [gr. παράδειγμα = przykład, wzór]
przyjęty sposób widzenia w danej dziedzinie, doktrynie itp.

paradygmat programowania
podstawowy styl programowania, służący jako sposób budowania struktury i elementów programu komputerowego

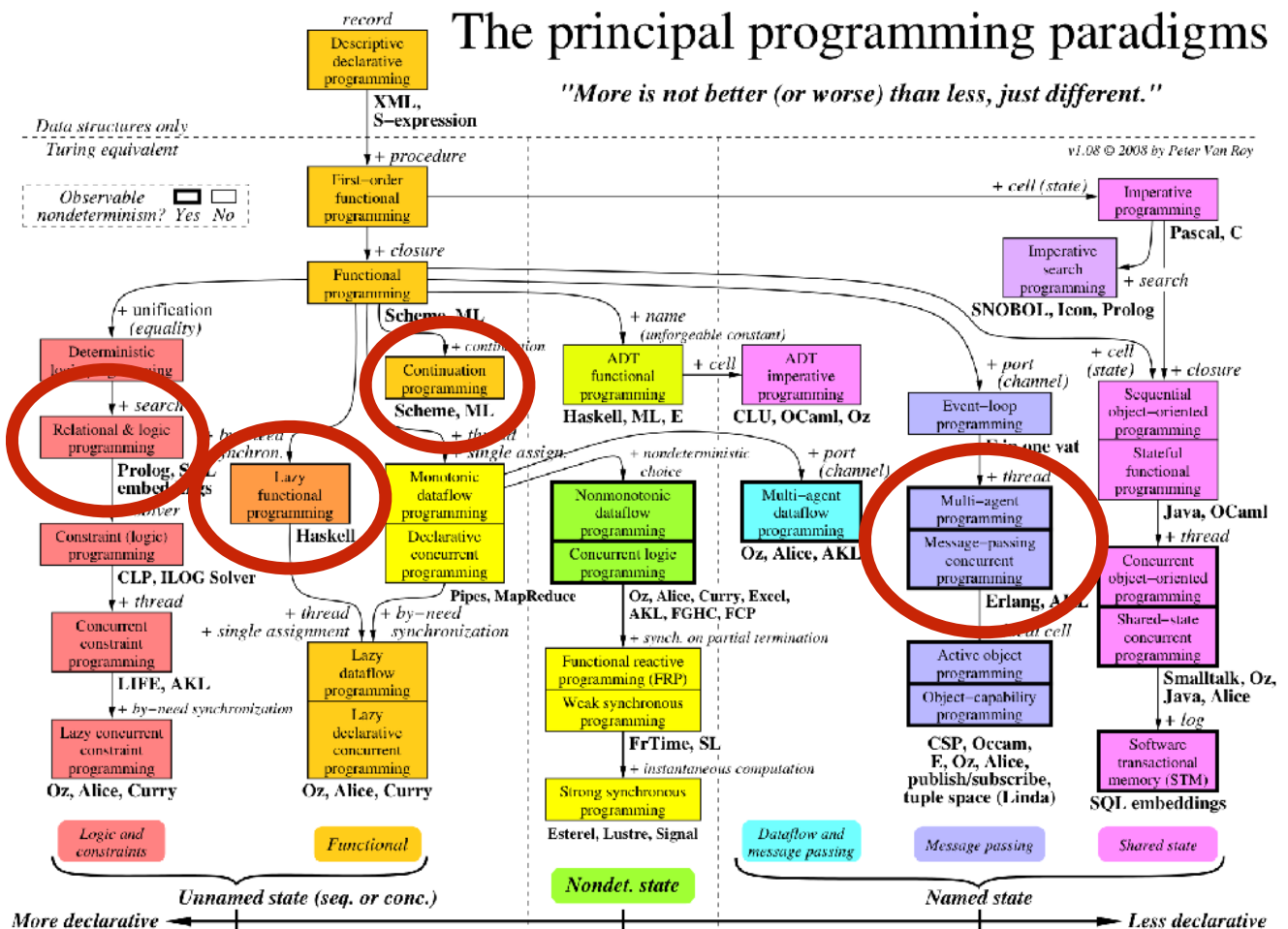
Języki — paradygmaty — koncepcje



koncepcja	paradygmat	język programowania
rekord, procedura, domknięcie, unifikacja, stan, wątek, jednokrotne przypisanie, ...	programowanie funkcyjne, programowanie imperatywne, programowanie logiczne, ...	Oz, Scheme, ML, Haskell, C, Pascal, SNOBOL, Erlang, Prolog, OCaml, Java, Curry, SQL, Alice, AKL, Icon, ...

The principal programming paradigms

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



Języki — paradygmaty — koncepcje
Zauważalny niedeterminizm

niedeterminizm (nondeterminism)

wykonanie programu nie jest w pełni zdeterminowane jego specyfikacją (np. w pewnych miejscach specyfikacja pozwala programowi wybrać co wykonywać dalej)

program planujący (scheduler)

podczas wykonywania wybór dalszych obliczeń dokonywany jest przez program planujący będący częścią run-time

Języki — paradygmaty — koncepcje
Zauważalny niedeterminizm

wyścig lub hazard (race)

kiedy wynik programu zależy od drobnych różnic w czasie wykonania różnych części programu

zauważalny niedeterminizm (observable nondeterminism)

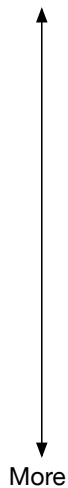
kiedy można zaobserwować różne wyniki przy uruchamianiu tego samego programu na tych samych danych

Języki — paradygmaty — koncepcje

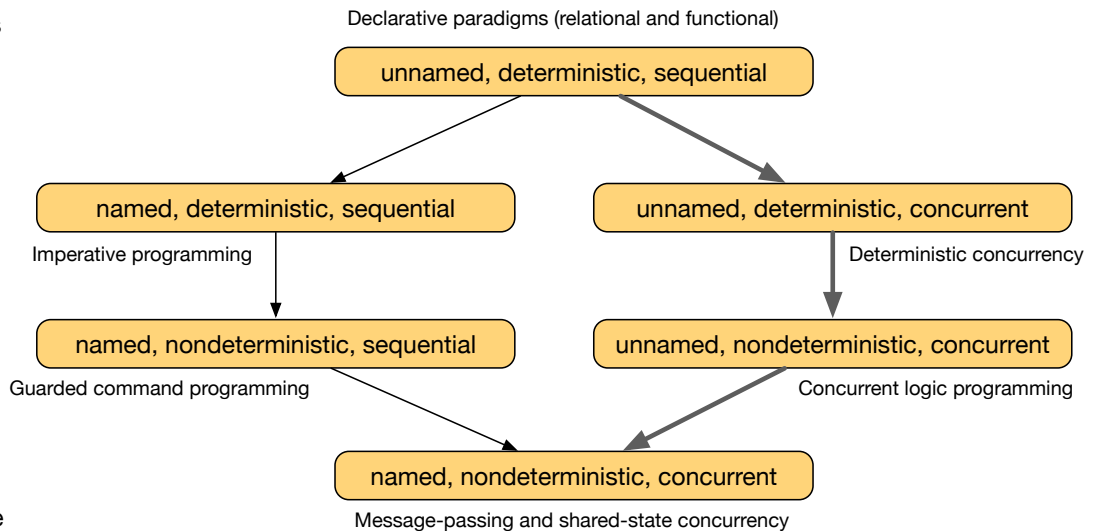
Nazwany stan

Expressiveness of state

Less



More



Koncepcja: **rekord** (record)

`R=nazwa(pole1:wartość1 pole2:wartość2 ... polen:wartośćn)`

`R.pole1 R.pole2 ... R.polen`

`R=nazwa(1:wartość1 2:wartość2 ... n:wartośćn)`

`R.1 R.2 ... R.n`

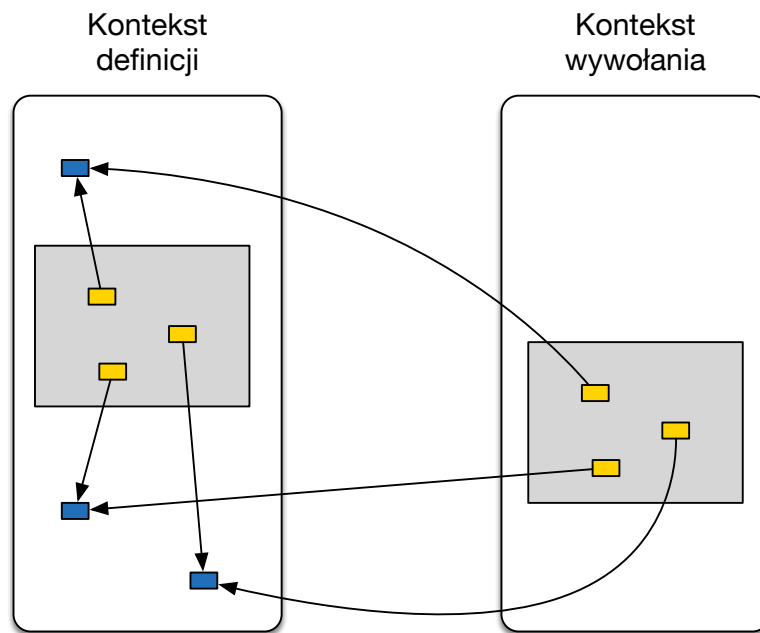
`R=nazwa(wartość1 wartość2 ... wartośćn)`

`R.1 R.2 ... R.n`

`C=circle(x:10 y:20 radius:5)`

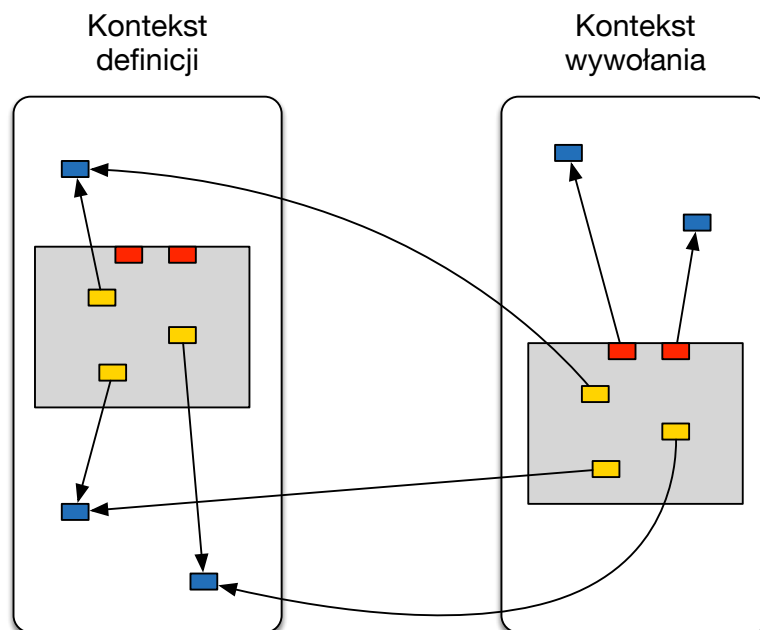
`R=rectangle(x:100 y:150 height:20 width:200)`

Koncepcja: **domknięcie** (closure)



Procedura bez parametrów

Koncepcja: **domknięcie** (closure)



Procedura z parametrami

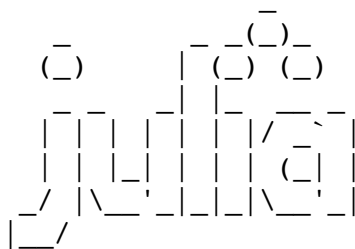
Koncepcja: **domknięcie** (closure)

```
declare  
fun {CreateAdder X}  
    fun {Adder Y}  
        X+Y  
    end  
in  
    Adder  
end
```

```
declare F G  
F = {CreateAdder 2}  
G = {CreateAdder ~5}  
{Show {F 10}#{G 20}}
```

12#15

Koncepcja: **domknięcie** (closure)



A fresh approach to technical computing
Documentation: <http://docs.julialang.org>
Type "help()" for help.

Version 0.3.6 (2015-02-17 22:12 UTC)
Official <http://julialang.org/> release
x86_64-apple-darwin13.4.0

```
julia> function create_adder(x)  
    function adder(y)  
        x + y  
    end  
    adder  
end  
create_adder (generic function with 1 method)  
  
julia> add_10 = create_adder(10)  
adder (generic function with 1 method)  
  
julia> add_10(3)  
13
```


Koncepcja: **niezależność, współbieżność**
(independence, concurrency)

współbieżność (concurrent)

gdy dwa fragmenty programu nie są od siebie zależne¹⁾

sekwencja (sequence)

gdy zadana jest kolejność wykonywania dwóch fragmentów programu

równoległość (parallel)

gdy dwa fragmenty programu są wykonywane jednocześnie na wielu procesorach

¹⁾ Formalnie: wykonanie programu składa się z częściowo uporządkowanych zdarzeń przejść między stanami obliczeń i dwa zdarzenia są współbieżne jeśli nie ma porządku między nimi.

Koncepcja: **niezależność, współbieżność**
(independence, concurrency)

Trzy poziomy współbieżności:

- System rozproszony: zbiór komputerów połączonych siecią. Współbieżnym działaniem jest **komputer**.
- System operacyjny: oprogramowanie zarządzające komputerem. Współbieżnym działaniem jest **proces**.
- Działania wewnątrz procesu. Współbieżnym działaniem jest **wątek** (thread).

Współbieżność procesów opiera się na **konkurencyjności** o zasoby.

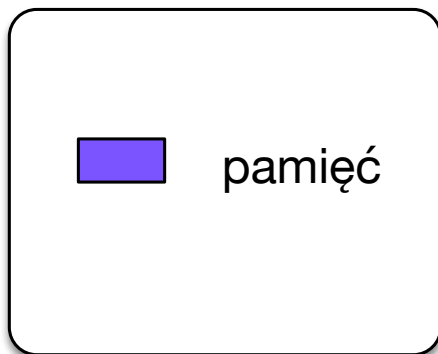
Współbieżność wątków opiera się na **współdziałaniu i współdzieleniu** zasobów.

Koncepcja: **nazwany stan** (named state, cell)

nazwany stan (named state)

sekwencja wartości w czasie jaką posiadała pojedyncza nazwa

Komponent A



Komponent B



Koncepcja: **nazwany stan** (named state, cell)

```
fun {ModuleMaker}
  fun {F ...}
    ...      % Definition of F
  end
  fun {G ...}
    ...      % Definition of G
  end
in
  themodule(f:F g:G)
end

M={ModuleMaker}  % Creation of M
{M.f ...}
{M.g ...}
```

Koncepcja: **nazwany stan** (named state, cell)

Jeśli funkcja musi policzyć ile razy była wykonana, to bez nazwanego stanu trzeba zmienić jej interfejs:

```
fun {F ... Fin Fout}
    Fout=Fin+1
    ...
end
```

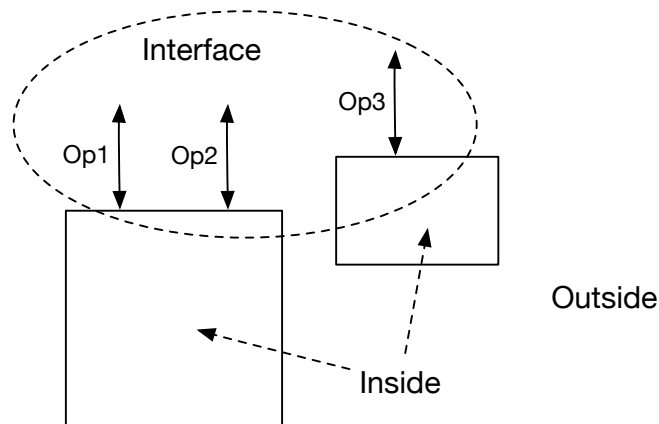
```
A={F ... F1 F2}
B={F ... F2 F3}
C={F ... F3 F4}
```

Koncepcja: **nazwany stan** (named state, cell)

```
fun {ModuleMaker}
    X={NewCell 0} % Create cell referenced by X
    fun {F ...}
        X:=@X+1    % New content of X is old plus 1
        ...        % Original definition of F
    end
    fun {G ...}
        ...        % Definition of G
    end
    fun {Count} @X end % Return content of X
in
    themodule(f:F g:G c:Count)
end

M={ModuleMaker} % Creation of M
```

Abstrakcja danych
(korzystanie z danych bez zajmowania się ich implementacją)



1. Interfejs gwarantuje, że abstrakcja danych zawsze działa poprawnie.
2. Program jest prostszy do zrozumienia.
3. Umożliwia rozwijanie bardzo dużych programów.

typ abstrakcyjny (ADT *abstract data type*)

zbiór wartości połączony ze zbiorem operacji na tych wartościach

CLU
B. Liskov et al.
1974

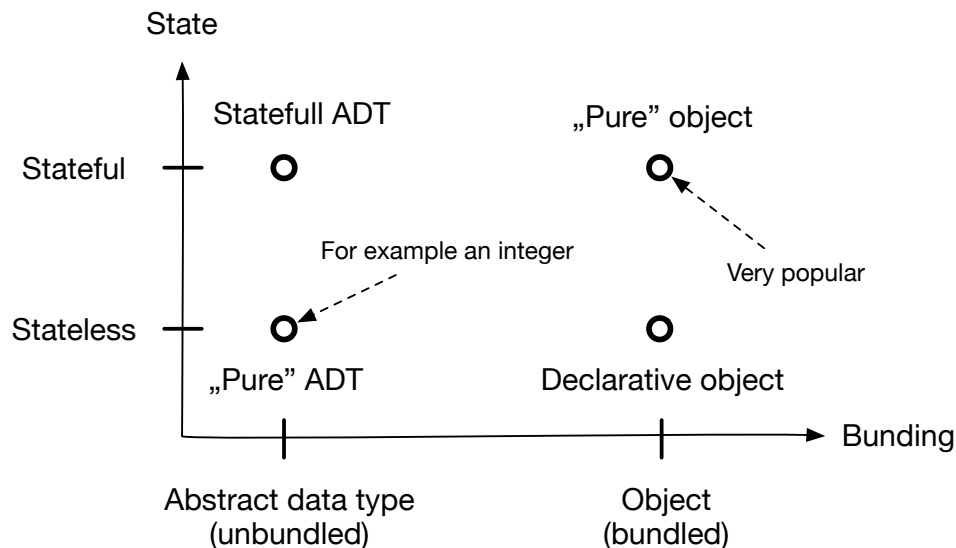
```
fun {NewStack}: <Stack T>
fun {Push <Stack T> T}: <Stack T>
fun {Pop <Stack T> T}: <Stack T>
fun {IsEmpty <Stack T>}: <Bool>
```

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of X|S1 then E=X S1 end end
fun {IsEmpty S} S==nil end
```

```
fun {NewStack} stackEmpty end
fun {Push S E} stack(E S) end
fun {Pop S E} case S of stack(X S1) then E=X S1 end end
fun {IsEmpty S} S==stackEmpty end
```

proceduralna abstrakcja danych (PDA *procedural data abstraction*)

łączy w postaci obiektu pojęcia wartości i operacji
(operacje wykonuje się poprzez wywołanie obiektu i
poinformowanie go jaką czynność powinien wykonać)



Abstrakcja danych: **polimorfizm**

```
class Figure
```

```
...
```

```
end
```

```
class Circle
```

```
  attr x y r
```

```
  meth draw ... end
```

```
...
```

```
end
```

```
class Line
```

```
  attr x1 y1 x2 y2
```

```
  meth draw ... end
```

```
...
```

```
end
```

```
class CompoundFigure
```

```
  attr figlist
```

```
  meth draw
```

```
    for F in @figlist do  
      {F draw}
```

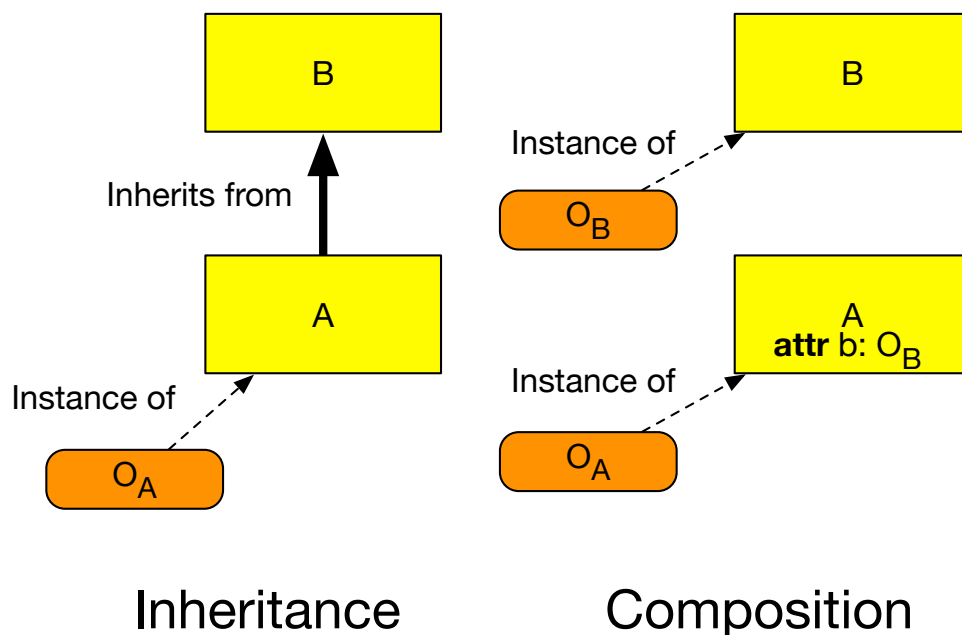
```
    end
```

```
  end
```

```
...
```

```
end
```

Abstrakcja danych: **dziedziczenie** a **złożenie**



Paradygmat: **deterministyczne programowanie** **współbieżne**

Przykład wyścigu: komórka **C** może zawierać 1 albo 2 po wykonaniu obu wątków.

```
declare C={NewCell 0}  
thread C:=1 end  
thread C:=2 end
```

niedeterminizm!

Należy unikać niedeterminizmu w językach współbieżnych:

1. Ograniczać zauważalny niedeterminizm tylko do tych części programu, które faktycznie go wymagają.
2. Definiować języki tak aby było w nich możliwe pisanie współbieżnych programów bez zauważalnego niedeterminizmu.

Concurrent paradigm	Races possible?	Inputs can be nondeterm.?	Example languages
Declarative concurrency	No	No	Oz, Alice
Constraint programming	No	No	Gecode, Numerica
Functional reactive programming	No	Yes	FrTime, Yampa
Discrete synchronous programming	No	Yes	Esterel, Lustre, Signal
Message-passing concurrency	Yes	Yes	Erlang, E

Paradygmat: **deklaratywna współbieżność**

Wątki mają tylko jedną operację:

- **{NewThread P}**: tworzy nowy wątek wykonujący bezargumentową procedurę P.

Zmienne przepływu danych służą synchronizacji, mogą być podstawiane tylko jeden raz i mają następujące proste operacje:

- **X={NewVar}**: tworzy nową zmienną przepływu danych.
- **{Bind X V}**: wiąże X z V, gdzie V jest wartością lub inną zmienną przepływu danych.
- **{Wait X}**: bieżący wątek czeka aż X zostanie związana z wartością.

Używając prostych operacji rozszerzamy operacje języka tak by czekały na dostępność swoich argumentów:

```
proc {Add X Y Z}  
  {Wait X} {Wait Y}  
  local R in {PrimAdd X Y R} {Bind Z R} end  
end
```

Tworzymy deklaratywną współbieżność leniwą przez dodanie nowej koncepcji synchronizacji przez-potrzebę (*by-need synchronization*):

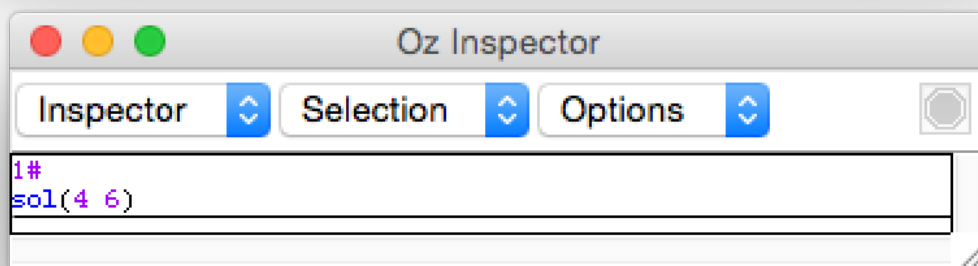
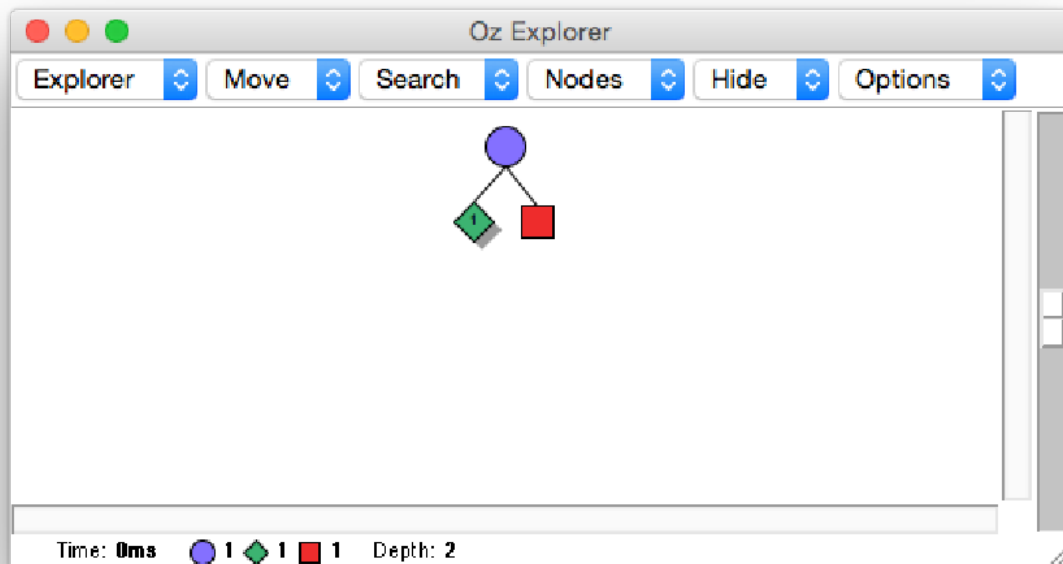
- {WaitNeeded X}: bieżący wątek czeka aż któryś z wątków wykona {Wait X}.

```
proc {LazyAdd X Y Z}  
  thread {WaitNeeded Z} {Add X Y Z} end  
end
```

Paradygmat: **programowanie z ograniczeniami**

Czy istnieje prostokąt o powierzchni 24 jednostek kwadratowych i obwodzie 20 jednostek?

```
declare  
proc {Rectangle ?Sol}  
  sol(X Y)=Sol  
in  
  X::1#9   Y::1#9  
  X*Y=:24   X+Y=:10   X=<:Y  
  {FD.distribute naive Sol}  
end  
  
{ExploreAll Rectangle}
```

przekazywanie ograniczeń (*propagate step*)

jak najbardziej ogranicza rozmiar dziedzin zmiennych za pomocą propagatora

propagator

współbieżny agent implementujący ograniczenia. Jest aktywowany gdy zmieni się dziedzina którejkolwiek zmiennej i stara się zawęzić dziedziny zmiennych aby nie były naruszone implementowane ograniczenia.

Propagatory aktywują się nawzajem poprzez współdzielone argumenty. Działają aż do osiągnięcia punktu stałego (nie są możliwe dalsze redukcje). Prowadzi to do trzech możliwości: rozwiązanie, niepowodzenie (brak rozwiązania) lub niepełne rozwiązanie.

strategia podziału (*distribute step*)

dokonuje wyboru pomocniczego ograniczenia C dzielącego rozwiązywany problem P na podproblemy $(P \wedge C)$ oraz $(P \wedge \neg C)$ w konsekwencji decydując o kształcie drzewa poszukiwań. Wspomniany podział dokonywany jest rekurencyjnie w każdym węźle drzewa.

Zagadka kryptoarytmetyczna:

	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

$$\begin{cases} s, e, n, d, m, o, r, y \in 0..9, \\ s > 0, m > 0, \\ -10000 \cdot m + 1000 \cdot (s + m - o) + 100 \cdot (e + o - n) + 10 \cdot (n + r - e) + d + e - y = 0 \end{cases}$$

Co można powiedzieć o zakresie dla zmiennej s ?

$$1000 \cdot s = 9000 \cdot m + 900 \cdot o + 90 \cdot n - 91 \cdot e + y - 10 \cdot r - d$$

Niech $min..max$ będzie zakresem prawej strony a $s_0..s_1$ zakresem zmiennej s .

$$1000 \cdot s_1 > max \rightarrow s \in s_0..\lfloor max/1000 \rfloor$$

$$1000 \cdot s_0 < min \rightarrow s \in \lceil min/1000 \rceil..s_1$$

Na tym etapie:

$$min = 8082, max = 89919, s_0 = 1 s_1 = 9 \rightarrow s \in 9..9$$

	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

81,000,000



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

9,000,000



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

1,000,000




	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

200,000

	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								




118,098



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

32,768



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

16,307



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

14,406



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

12,348



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

10,290



	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

3,575



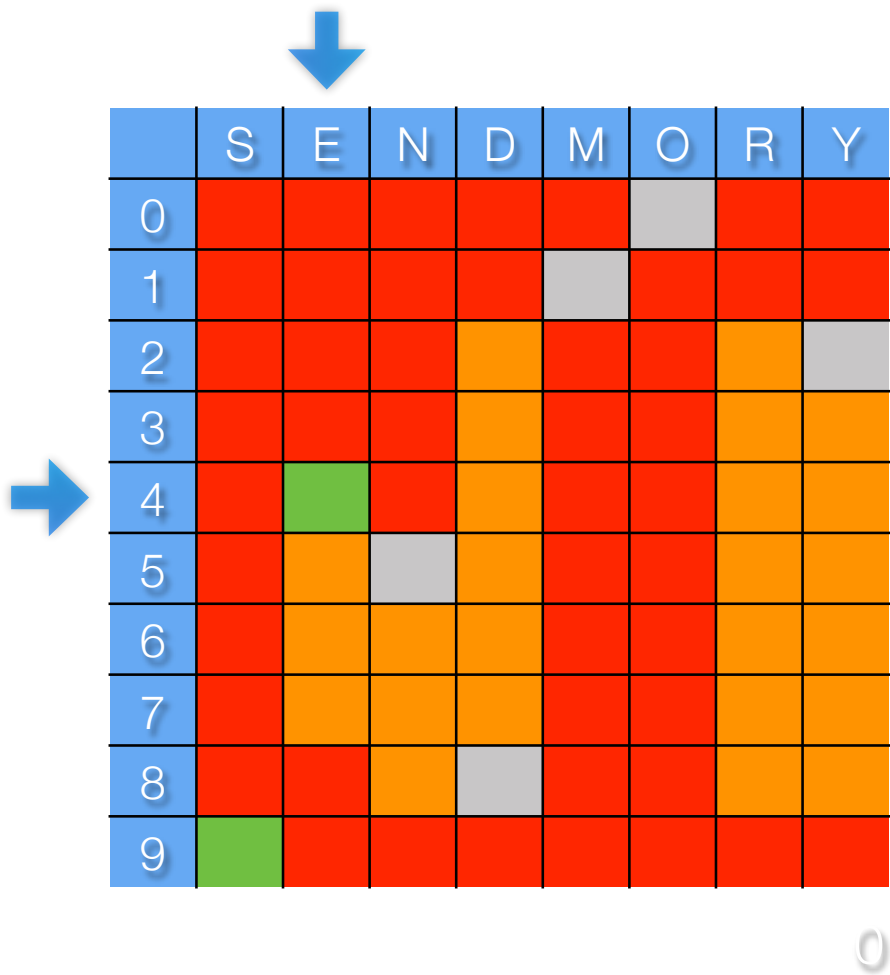
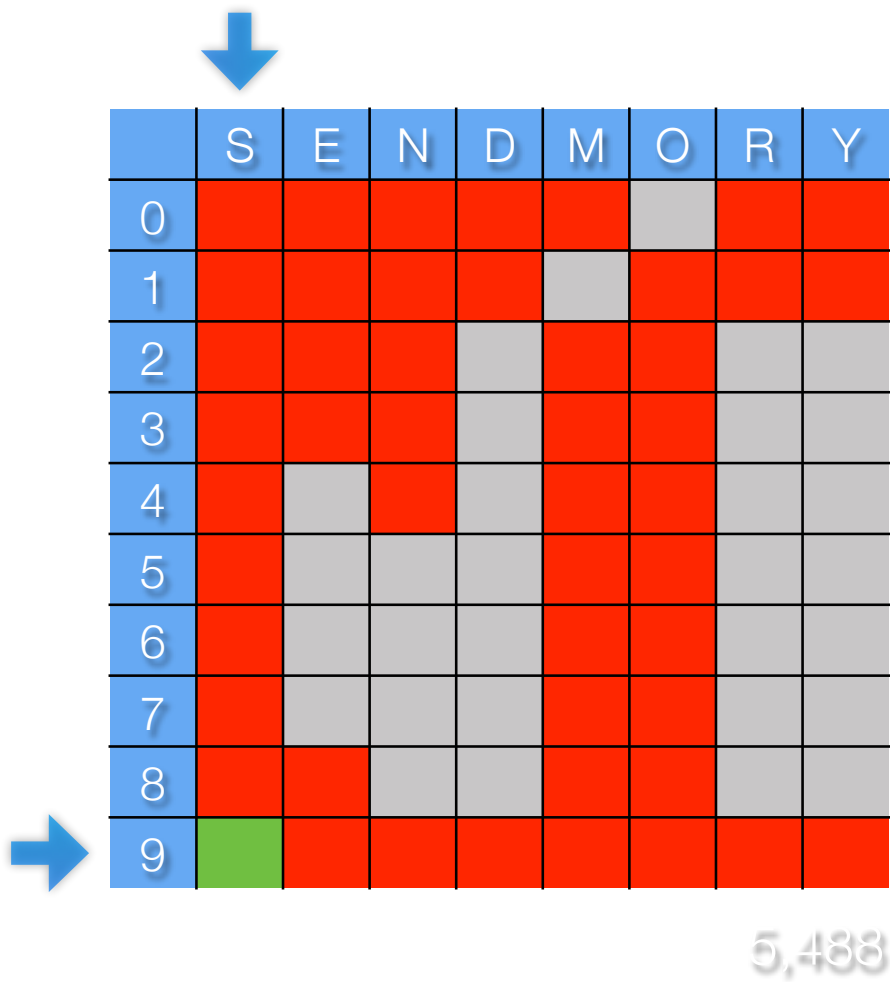
	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

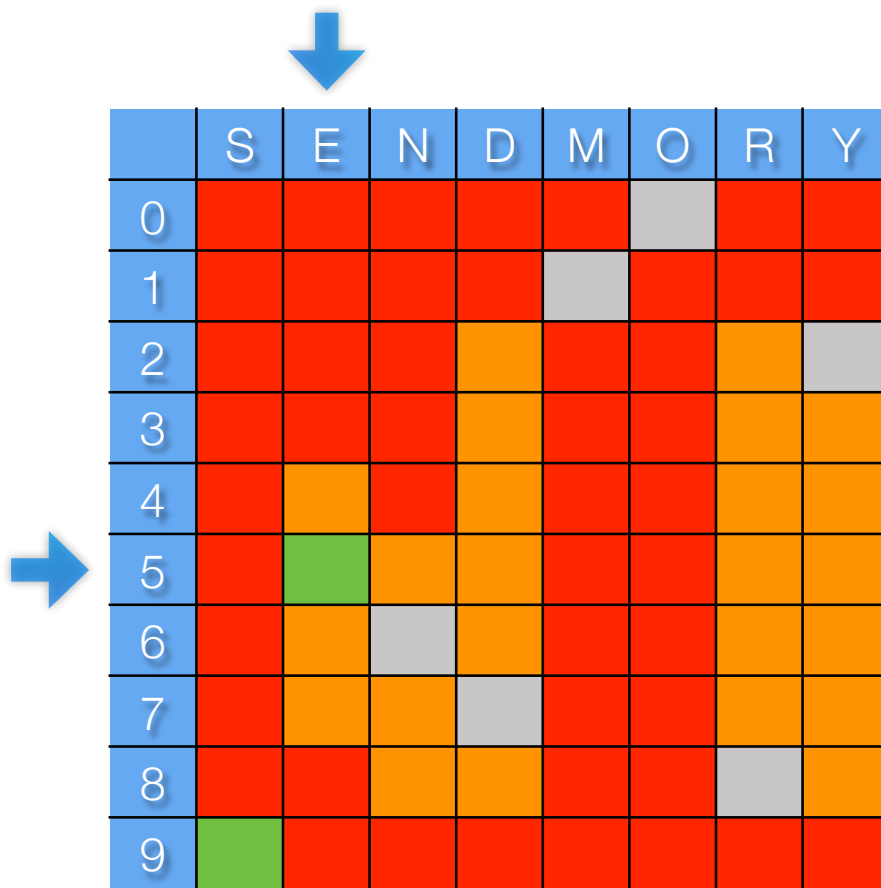
6,360



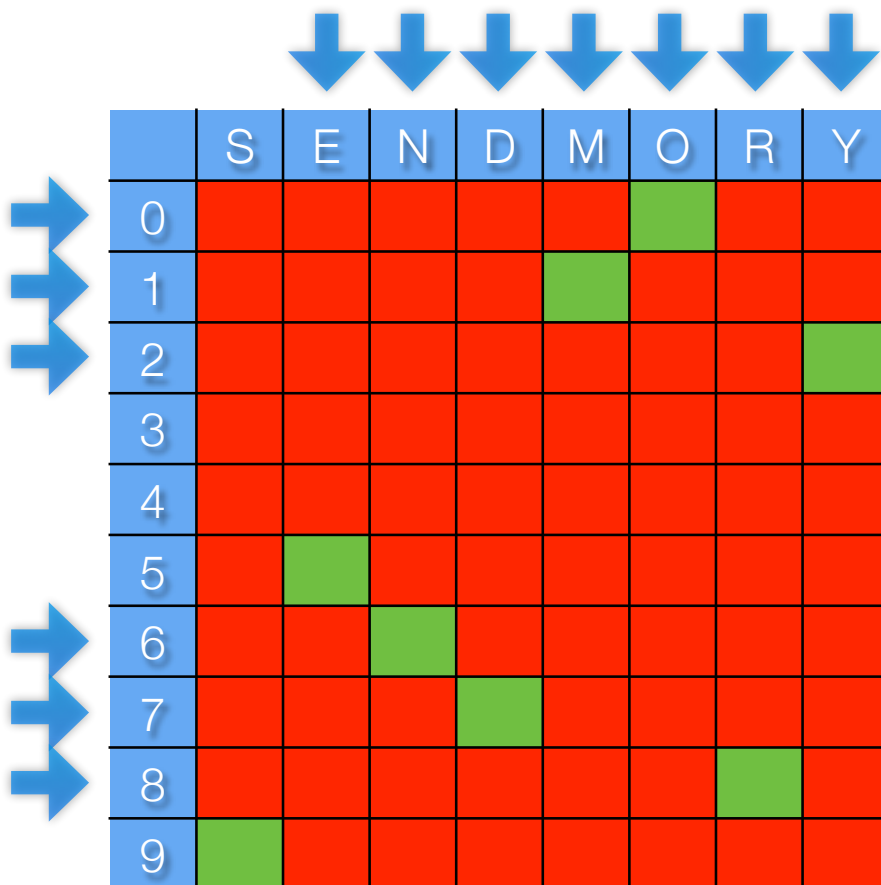
	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

5,488





1



$$\begin{array}{r}
 9567 \\
 + 1085 \\
 \hline
 10652
 \end{array}$$

1